Chess-AI Parallel Programing

Dynamic parallel distribution

Cheng-En, Cai 310551068[†]
Department of Computer Science and Engineering
University of National Yang-Ming
Chiao Tung, Taiwan
s40042488@gmail.com

Min-Hua, Tsai
Department of Computer Science and Engineering
University of National Yang-Ming
Chiao Tung, Taiwan
leoleo752175@gmail.com

1 ABSTRACT

In a chess match in 1997, the parallel chess supercomputer Deep Blue defeated the world champion Garry Kasparov making a milestone in the history of computer science. In this final project, we would like to study the secret behind Deep Blue and implement different parallel ways to improve the Chess AI engine and analyze the performance by time, scalability, and portability.

In the following report, we will introduce the parallelization program from different perspectives to speed up the AI chess program's performance. Two versions of parallel programs are implemented: PVS-search, and Root Splitting Search. In addition, we will investigate the performance of parallelization by different methods and compare the results of PVS-search and Root Splitting Search, and find out the reasons for the differences from repeated tests.

2 Introduction

In this Chess AI engine, commonly used methods such as Alpha-Beta Pruning, because, in actual problems, the number of searches required is quite large, which will lead to poor performance, so we use Alpha-Beta Pruning to remove some meaningless Search. The method of Alpha-Beta Pruning is based on a min-max search, where alpha is the highest value found by any node on the Max path, and beta is the lowest value found by any node on the Min path. When a node is found to be worse than the current alpha or beta, the subtree where the current node is located is removed, and the rest of the nodes are removed, and the entire subtree is deleted, so the performance will be effectively improved.

Here we use Root Splitting Search and Principal Variation Search (PVS), which also use Alpha-Beta Pruning but different algorithms. And our biggest goal is to use Alpha-Beta Pruning as our search method and compare the differences between the two.

We use OpenMP to parallelize our programs and improve performance. Since the Alpha-Beta Pruning method is based on min-max search, the search and calculation scores of each node are very suitable for parallelization when used. Finally, we compare the results of parallelization with the results without parallelization and compare whether the performance can rise as expected when the thread increases. And we will also sort each layer of ply to test the improvement in performance and the difference in scalability. Finally, we will also display the entire board to ensure the correctness of our answers.

3 Proposed Solution

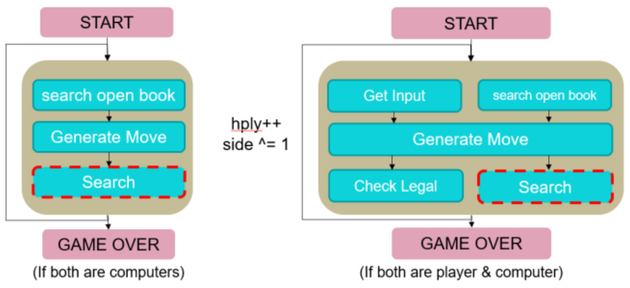

Figure 1: Flow Chart of entire Program

Figure 1 shows the flow chart of our main program, each element is used to be as a module, and all modules are composed to complete the entire program. START is to the initialize board's state and initialize all parameters. Generate Move is to generate all the legal moves, the pieces that can be moved by that side (white or black). Check Move is to check if the player's input is a legal move from Generate Move. Search is to search all the pieces that can be moved, and recursively go down to search until depth is enough, and evaluate the current board state, and finally return the score. Game Over is to check if the board is finished or not. If the game is over, the program would wait for the player's next command. If the game is not over, the program would switch the side and go to Get Input or Search open Book.

3.1 Alpha-Beta Pruning

Alpha-Beta Pruning algorithm is an improvement on minimax search, which is one of the majority of search algorithms on board games. Minimax search is searching all nodes and each layers would choose a best(worst) score from child node. Alpha-Beta prunes leave that cannot affect the outcome, which reduce searching nodes' time. The following the pseudo code is one version of the Alpha-Beta Pruning algorithm.

```
int AlphaBetaSearch(int depth, int alpha, int beta){
  if(depth ==0){
    return Evaluate(board);
    foreach(move in GenerateMove(board) ){
      score = -AlphaBetaSearch(depth-1, -beta, -alpha);
        if(score > alpha){
          if(score > beta)  return beta // cutoff
          else alpha = score
        } } } }
```

Figure2: Nega max pseudo code.

Alpha-Beta Pruning algorithm uses a lower and upper bound (alpha and beta), trying to cut unnecessary nodes. In the beginning, the bounds of the root node are initialized at -99999 and 99999. When the maximum value is found and smaller than the upper bound value, the upper bound value would be updated as the maximum value. On the other side, if the minimum value is found and larger than the lower bound, the lower bound would be updated as a new value at a minimizing level.

Nega max algorithm is another version of the alpha-beta prune algorithm, which is much easy to implement. In this algorithm, we always assume it should find the minimum score which larger than the lower bound and update the lower bound. If we evaluate the score as larger than the upper bound, we can just prune it, since it's too good to move the piece. The opponent would never let us go this way.

3.2 Parallel Search Method I

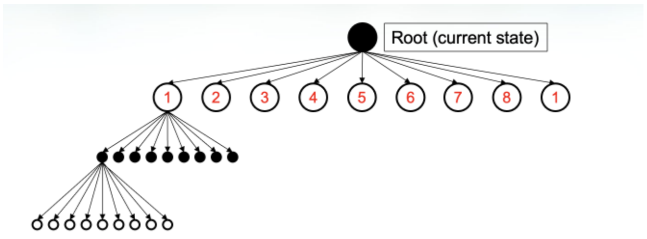

Figure 3: Root Splitting search and Distribute thread in Method I

Of all the programs, Search is the one that takes up the most time. On average, all legal moves on each level are about 30 steps long. In other words, each time depth is increased by one, the number of the child nodes to be searched would grow by an exponential factor of 30.

In order to divide the area under each thread equally, we take the nodes under root as the area under one thread and continue to search down. The hope is that by doing this, the workload will be shared evenly and the workload will be balanced.

3.3 Parallel Search Method II (PVS)

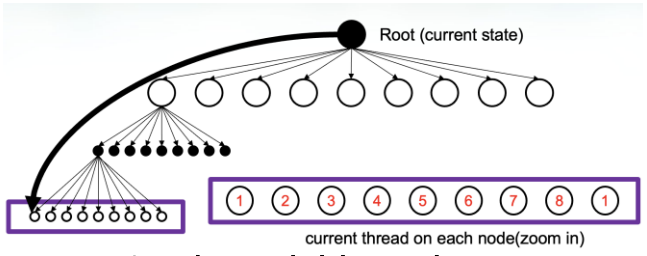

Figure 4: PVS search start at the leftmost node

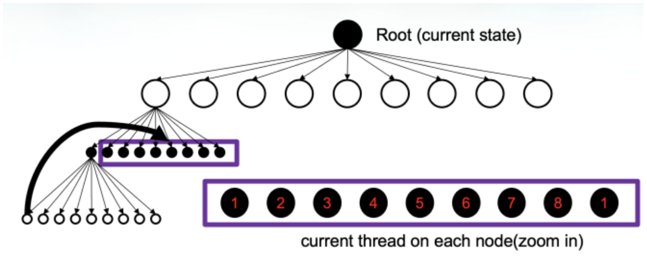

Figure 5: Search would recursively distribute remaining subtrees

The Principle Variant Splitting searches the same tree as the minimax algorithm. It does the search in a parallel fashion with multiple threads. In Principle Variant Splitting, the first thread would start and go to the leftmost child node. The remaining child nodes are searched later with other threads (Figure 4). When depth 3 is a search done, all the threads would distribute depth 2 remaining nodes to search (Figure 5). It would be a recursive loop until all nodes have been found.

The difference between this and Method 1 is the way of work is allocated. Since we use the alpha-beta pruning algorithm, it is not possible to know exactly how many times a node's child node has been cut off and how often. Therefore, PVS search is used, so that the average cutoff could be spread evenly across the threads.

3.4 Sort

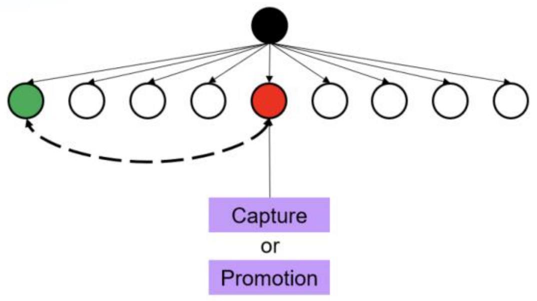

Figure 6: sort nodes

Although it is not possible to determine exactly where and how often a cutoff occurs. It is possible to know what might cause a cutoff to occur. A cutoff is generated when one side scores more points than the other. The possible causes are as follows: 1. Exchange. 2. promotion.

Once we know the reason for the cutoff, we can add weight to the node so that we can see the status of each node. Whenever we search, we will check the child nodes, and if it is an exchange or promotion node. If It is, we will move the node to the leftmost side, hoping to generate the cutoff condition in advance (Figure 6).

4 Experimental Methodology

4-1 Our platform and devices :

Window 10, 64bits

Intel® Core™ I7-6700 CPU @ 3.40GHz (4CPUs 8Threads)

RAM 16GB

4-2 Input sets and All tests :

Input sets: hply == 5 (the computers play each other five moves.)

1. Search with Alpha-Beta Pruning or not.
2. Compare Method I and Method II without sort
3. Compare Method I with(out) sort
4. Compare Method II with(out) sort
5. Compare Method I and Method II.

5 Experimental Results

At **first,** we used method 1 to assign work to different threads in a dynamic way and found two things. Firstly, when the number of threads exceeded 4, the speed up did not continue to rise. Secondly, when the number of threads went up, the speed up did not continue to increase in a proportional way and performance dropped significantly.

5-1 with(out) cutoff

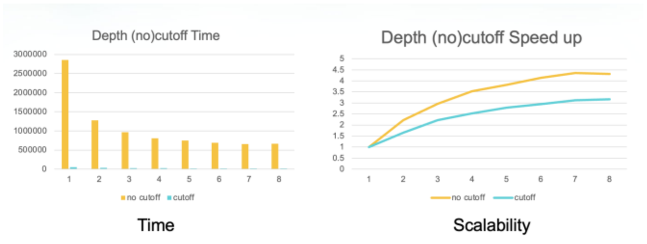

Figure 7: The difference between cutoff and without cutoff

Our guess is that the cutoff will cause the workload of each thread to vary, resulting in an unbalanced workload. We further investigated and removed the cutoff function. We found that when the thread was up, performance was significantly higher than with a cutoff. However, there was a huge difference in the time taken compared to the cutoff (Figure 7). Therefore, in the following experiments, we continued to use cutoff as a base and **parallelized** the search in different ways.

5-2 Compare Method I and Method II without sort

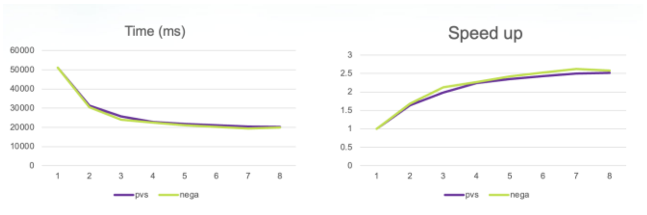

Figure 8: Compare Method I and Method II

We will then allocate the workload to different threads according to the method I and method II, and compare the time difference between method I and method II. To be fair, method I and method II will both start searching at the beginning of the board, and hply will be set at 5 (in other words, the computers will play each other five moves)

As we can see from Figure 8 above, there is no significant difference in the time spent on method I and method II with different numbers of threads allocated to them, and the speedup is almost the same. Our guess is that PVS is indeed able to distribute the cutoff possibilities evenly across the different threads, however, after assigning work to the threads, PVS has to wait until all the threads have done all the nodes at that level before it can continue to assign work to the higher levels. This will result in hidden barriers which will negate the advantage of an evenly distributed cutoff.

5-3 Compare Method I with(out) sort

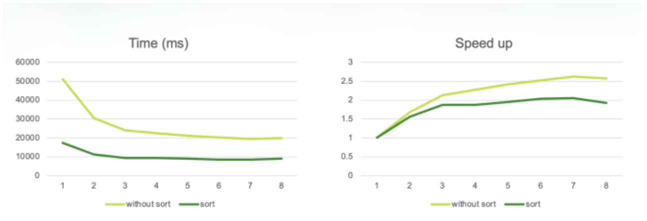

Figure 9: method I spend time and speed with(out) sort

In order to speed up the search, we look at our program. If we can anticipate when a cutoff will occur, we may be able to improve the speed further. The possible factors for a cutoff are mentioned in the method section. Firstly, exchange, and secondly, promotion. If these nodes are searched and cut off earlier, we can reduce the number of nodes to be searched and further reduce the time spent.

We sorted the node searched in method I. In Figure 9, we can clearly see that after nodes in method I was sorted, the search speed was much faster than without the sort, but the speed up did not increase according to the number of threads (performance decreased). Our conjecture is that the load balance of method I is not more evenly balanced because the cutoff occurs earlier. Instead, it is more likely to reflect the unbalanced workload across the different threads, thus causing a drop in performance.

5-4 Compare Method II with(out) sort

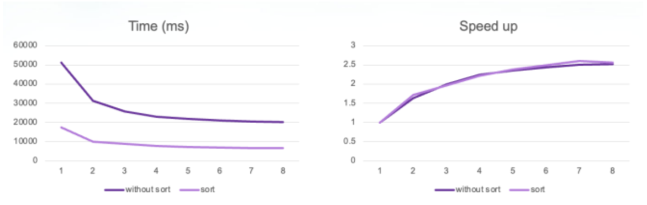

Figure 10: PVS spend time and speed up with(out) sort

In Figure 10, we can see that after sorting the node, the time spent is significantly reduced because it is cut off earlier, and the performance is still the same as without sort because the thread rises.

5-5 Compare Method I and Method II

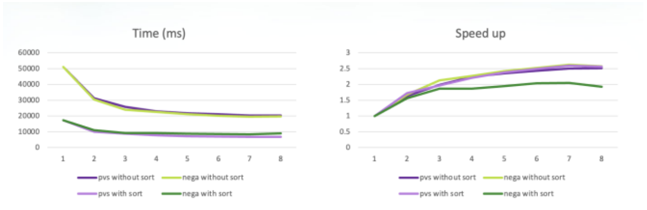

Figure 11: compare Method I and Method II with(out) sort

After comparing all the methods, the search by sort will reduce the number of nodes and the time required to search by cutting off earlier. By using Method II (PVS search), the number of cutoffs can be spread evenly over different threads, making the workload of each thread more even than that of Method I. Method I, on the other hand, can advance the cutoff due to sort, but it still does not solve the unevenness of the workload and is more pronounced.

6 Conclusion

After our implementation and analysis, we will find that using the Alpha-Beta Pruning algorithm, its performance will be greatly improved.

When we compare PVS Search and Nega Search, we find that there is no significant difference in the time spent and performance improvement between the two. We guess that because PVS Search allocates threads, it will need to wait until all threads have completed the calculation of the current layer. The next distribution will be carried out, which will offset its advantage of evenly distributed cut-off.

And in our implementation, we have also added sort to know that there will be cutoffs as soon as possible, and the performance improvement obtained in this respect is also significantly different. We also found that sort has different effects on our two different algorithms. For Nega Sort, its efficiency will decrease with the increase of threads. We mainly speculate that it is caused by an unbalanced load. But on the contrary, using PVS Search does not have this problem, and the efficiency is still the same as the unsorted one.

After these implementations, we learned that proper parallelization can speed up the entire calculation speed very effectively. Although the extremely deep situation has not been tested due to the time factor, the speed and intensity of the calculations by the program we made It is enough to cope with a normal level of the duel, I believe that deepening the intensity will have a greater improvement.

7 RELATIVE WORK

PVSplit introduces the design and implementation of a multithreaded distributed system for parallel game search. It provides a relatively high-level and machine independent programming system that greatly simplifies the efficient parallelization of irregular applications on a cluster of workstations. Fig1 explains the main structure of PVSplit.

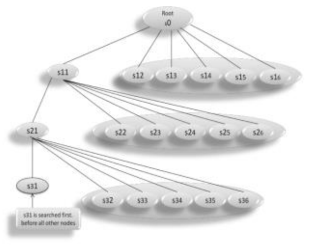

Figure 12 PVSplit structure

The most left and deepest node, s31, would search first before other nodes. After one thread finds this node, something would equivalently assign the same depth of nodes to processors. When nodes in-depth 3 finish to be searched, nodes in-depth 2 would assigned to the processors, and so on. This method could increase the scalability of performance. Even we have pruned the nodes, each processor could still equivalently have a balanced workload.

8 REFERENCES

[1] Gao, Y., & Marsland, T.A. Multithreaded Pruned Tree Search in Distributed Systems.

[2]Matthew Guidry, Charles McClendon A Distributed Chess Playing Software System Model Using Dynamic CPU Availability Prediction

[3] Selim G. Akl, David T. Barnard, and Ralph J. Doran. Design, analysis, and imple- mentation of a parallel tree search algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-4(2):192–203, March 1982.

[4] Rainer Feldmann. *Game tree search on massively parallel systems*. PhD thesis, Dept. of Math. and Computer Sci., University of Paderborn, Paderborn, Germany, 1993.

[5] R. M. Hyatt and B. W. Suter. A parallel alpha/beta tree searching algorithm. *Parallel Computing*, 10:299–308, 1989.

[6] Jean-Christophe Weill. The ABDADA distributed minimax search algorithm. In *Proceedings 1996 ACM Computer Science Conference*, pages 131–138, Philadelphia, 1996.

[7] Russell, Stuart J.; Norvig, Peter (2003), Artificial Intelligence: A Modern Approach (2nd ed.)

[8] E. Felton and S. Otto, "A highly parallel chess program," in Proc. Int. Conf. Fifth Generation Computer Systems, Tokyo, Japan, 1988.

[9] Feldman, R., Mynsliwietz, P., and Manien, B. game tree on a massively parallel system in Advances in Computer Chess 7 (1994) University of Limburg pg. 203-215

[10] “Parallel Chess Searching and Bitboards” http://www2.imm.dtu.dk/pubdb/views/edoc _download.php/3267/ps/imm3267.ps