

Parallel Programming Term Project

1998001 강문수

1999031 박용백

1999043 손동환

Parallel Dynamic Tree Searching

1. Overview

Tree search algorithms play an important role in many applications in the field of artificial intelligence. For instance, theorem provers, expert systems, robot control systems and game playing programs contain tree search algorithms as their basic part.

Our goal of this term project is to implement dynamic tree searching algorithm in parallel systems such as CrayT3E or clusters. We have chosen the Omok game for adapting our dynamic tree search program. For Omok implementation, we had to write evaluation function and user interface in addition to dynamic search tree program.

We first implemented sequential codes for dynamic tree searching with minimax algorithm, which reflects the assumption that at all times each side will choose the line of play with the most favorable score. And alpha beta pruning guarantees to give the same answer as full-width minimax search without looking at all nodes of the tree.

We then parallelized the searching the branches of the tree.

2. Sequential program

2.1 Minimax

The minimax algorithm is a method of selecting the best choice of action in a situation, or game, where two opposing forces, or players, are working toward mutually exclusive goals, acting on the same set of perfect information about the outcome of the situation. It is specifically applied in searching game trees to determine the best move for the current player of a game. It uses the simple principle that at each move, the moving player will choose the best move available to them.

The game tree consists of all moves available to the current player as children of the root, and then all moves available to the next player as children of these nodes, and so forth, as far into the future of the game as desired. Each branch of the tree represents a possible move that player could make at that point in the game. Evaluating the game at a leaf of this tree yields the projected status of the game after that sequence of moves is made by the players. A deeper search of the game tree provides more information about possible advantages or traps and therefore yields a better move.

In a situation, like Omok, with two players, Black and White, Black values moves with high evaluation

scores and White values moves with low evaluation scores. A minimax search determines all possible continuations of the game to the desired level, evaluating each possible set of moves and assigning a score. The search then steps back up through the game tree and alternates between choosing the highest child score at nodes representing Black, and the lowest child score at nodes representing White. The score returned is called the "backed up" score since it is chosen by backing up through the tree. Thus, stepping through the game tree is equivalent to examining alternating moves between Black and White. Such a forward looking search is beneficial because Black might make a fantastic move at one level, only to allow White to make a move that can be fantastically devastating. By looking ahead in this way, Black can make moves that are advantageous, but that don't allow White to make moves negating such benefit.

Algorithm

The minimax algorithm is a recursive algorithm that takes as arguments the current layout of the game board and a parity number which reflects whether you want a minimax or maximin. It returns the chosen move. The algorithm calls minimax on the boards resulting from each possible move at the current level.

At each level Minimax generates the possible moves at that level, and applies them in turn to the current board, calling minimax on the resulting board. Each returned score is compared to the current best. If the returned score is more advantageous for the current player, that score replaces the best score and the move that generated it replaces the best move. Should you decide to end the search at some depth, simply give Minimax no moves to apply, and a score to return to higher levels. Whether you are maxing, or mining is taken care of by the archetype itself.

The figure below shows the minmaxing, which is based on the reasonable supposition that white always chooses the line of play that maximizes the score.

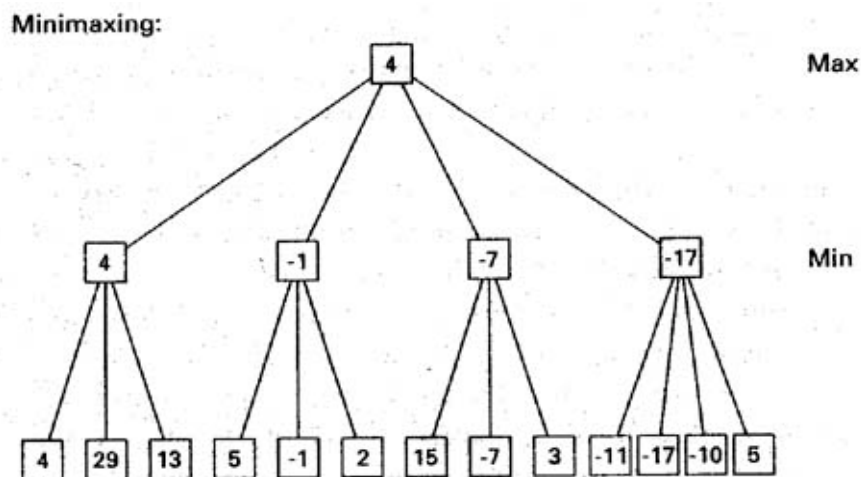


Figure 1. Minimaxing

2.2 Alpha-Beta Pruning

Since the amount of work a minimax search generates increases exponentially as a move is examined to a greater depth, to reduce the time required for the search it must be restricted so that no time is wasted searching moves that are obviously bad for the player. One key way to do this is to implement alpha-beta cutoffs in the minimax search. Alpha-beta cutoffs work on the previously discussed principle that a player, Black, will not make a seemingly good move if it allows White to make an even better move. Thus, if while examining moves, a move is found with a score worse than the best score now guaranteed, the algorithm will discontinue the search of that branch of the tree. The exact implementation of alpha-beta keeps track of the best move for each side as it moves through the tree. If a move is evaluated that is advantageous for the current player but not for the opponent, then the rest of the moves in the tree are discarded because progressing to that state would not be allowed by one of the players.

Using alpha-beta cutoffs can cause immense speedups in the run time of Minimax by reducing the size of the tree that must be searched. Figure 2 shows the above minimax tree adjusted with alpha-beta pruning.

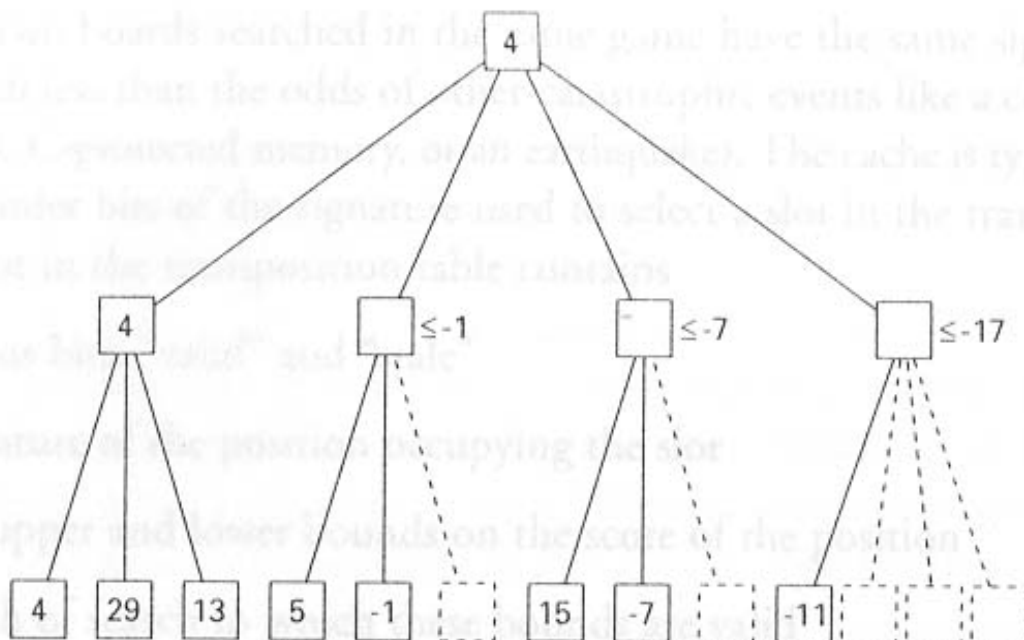


Figure 2. Alpha-beta pruning

3. Parallel Program

The parallelism comes from searching different parts of the tree at the same time. Master processor assigns the first depth nodes to slave processors and each processor searches dynamic tree separately. In our Omok program, the first nodes are the possible positions on board the number of which is 143. So the $143/n$ leaves are assigned in each node separately in case of n processors available. Each processor searches trees regarding the child nodes as root and return the result to master node.

Figure 3 shows the load balancing in case of 4 processors are used.

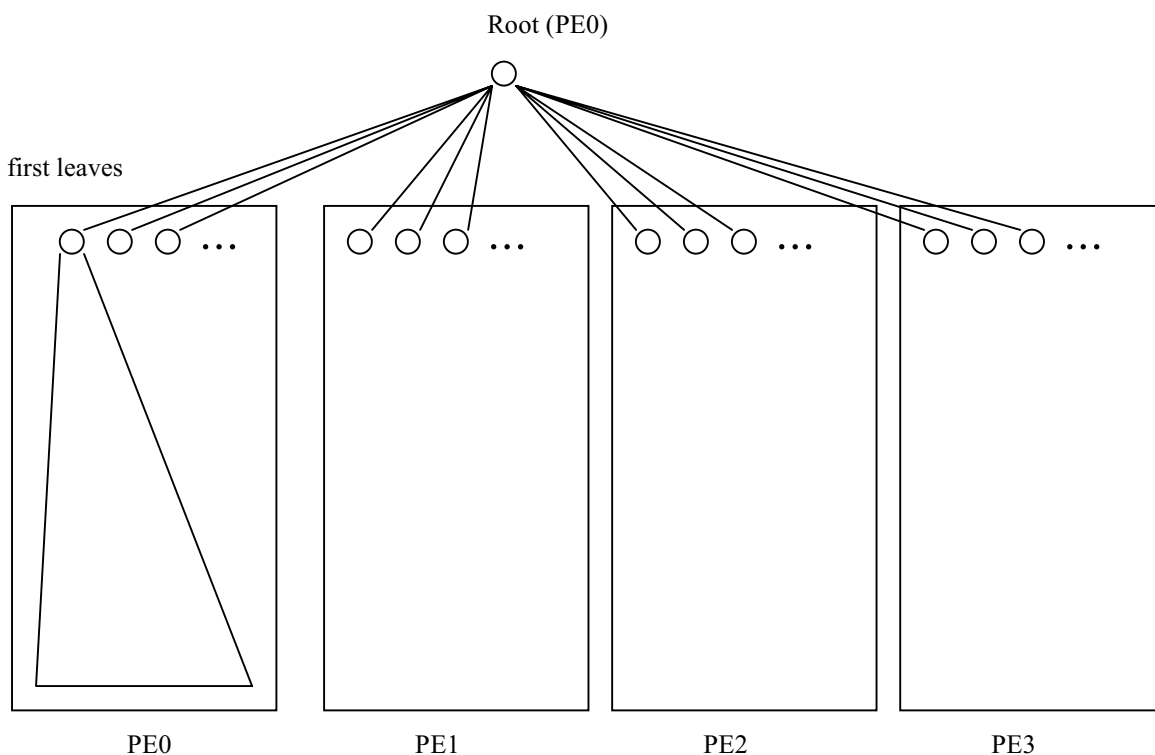
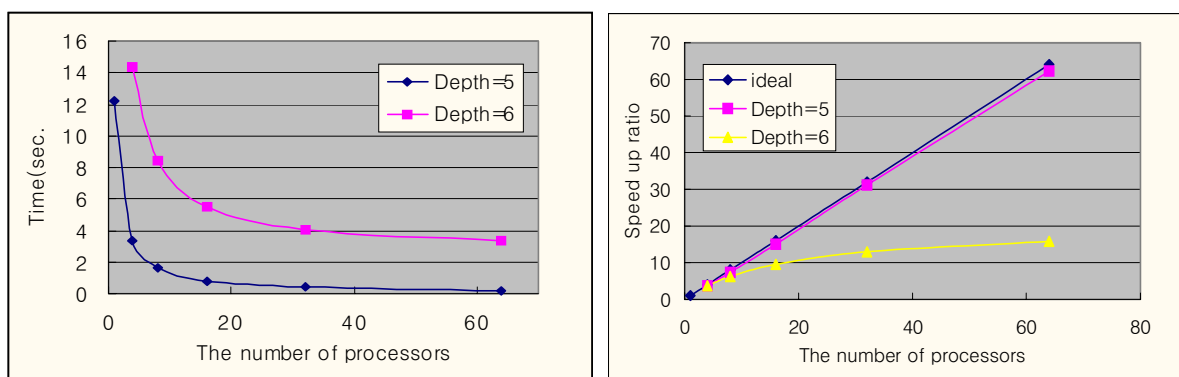


Figure 3. Leaf distribution

4. Results

We have executed the program on CrayT3E on various node both sequential and parallel codes. We set the depth of tree 5th and 6th and checked time as shown below.

Processor	Depth	
	5	6
1 (sequential code)	12.211790 sec	not executed for memory allocation fail
4	3.344301sec	14.367762sec
8	1.658309sec	8.435735sec
16	0.811530sec	5.499355sec
32	0.392454sec	4.049213sec
64	0.196337sec	3.322507sec



As seen in our results, time required for searching in depth 5 is almost linear as processors increase, which shows dynamic tree searching is paralleled well. But in depth 6, the execution times of 32 processors and 64 processors are almost same. This non-linearity in depth 6 shows that our parallel has problems in scalability of number of processors. Our algorithm divides the child nodes in the first level of dynamic game tree. As the number of child nodes in first level is almost static, if the small number of processors are provided, the effect of parallelization is high, but with large number, the effect is small. For example, if the number of child nodes is 143 and 4 processors are provided, 37 or 38 child nodes are assigned to each processor. In case of 16 processors, 8 or 9 child nodes are assigned. But in case of 32 processors, 4 or 5 child nodes, and in case of 64 processors, 2 or 3. We can see easily that the difference between 4 and 16 processors is much larger than that of 32 and 64. For this reason, the effect of parallelization is smaller as processors are more provided.

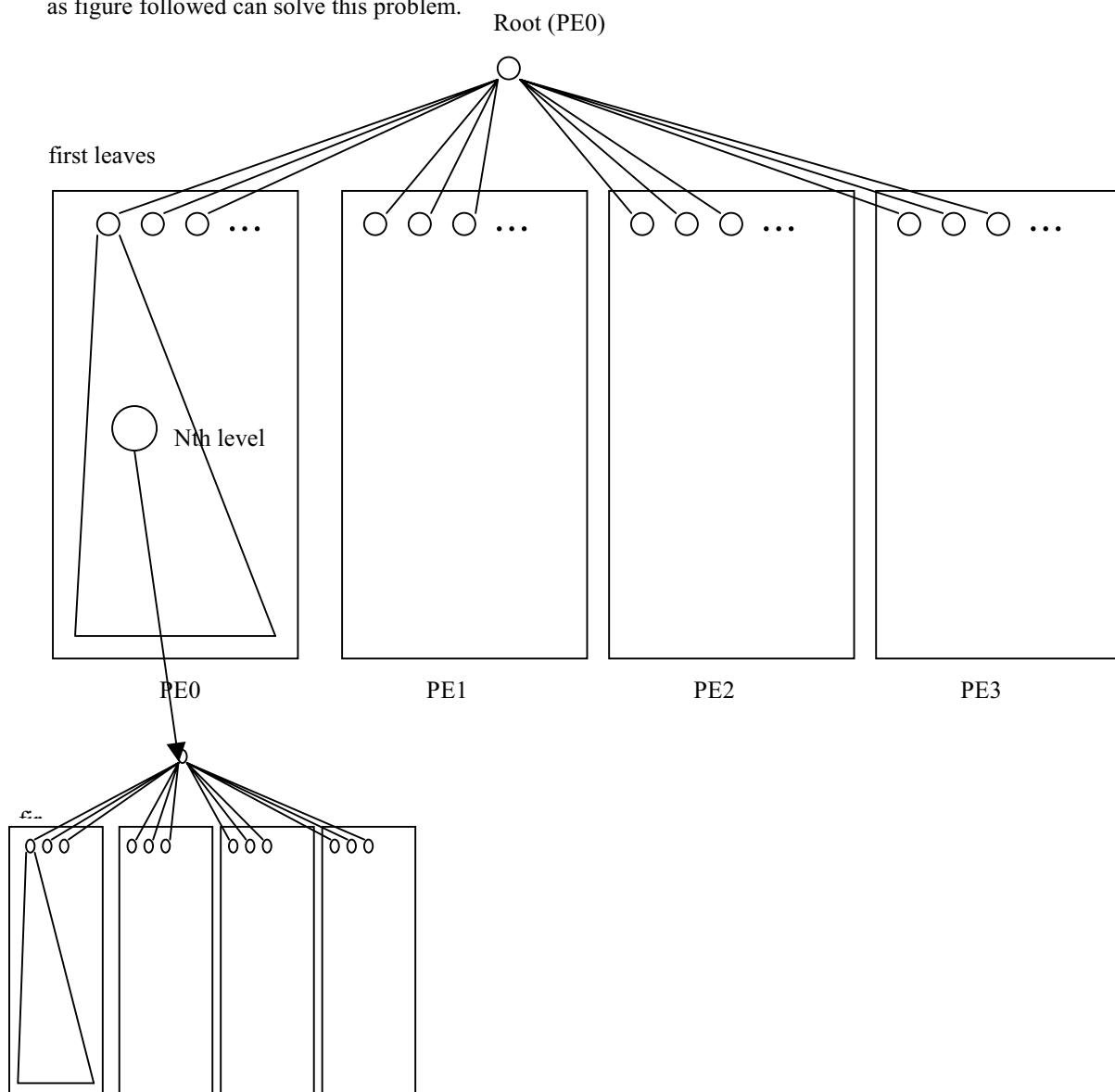
Even though the saturation point in speed up should have existed for the same reason above in case of depth 5, but it didn't for the size of dynamic game tree is much smaller than that of depth 6. The ratio in depth 5 almost shows ideal case, but real execution time of 32 and 64 processors are not much different.

5. Conclusion

Though we first intended to implement all the program for the Omok game, we just implemented the sequential and parallel code for dynamic tree search and evaluation function omitting the code for user interface for we have paid too may time on implementing sequential code as well as parallel code. But it is thought that that part is not so important in our project.

We tried to run our program in cluster but failed because we couldn't compile our program on cluster. It is thought that there were some problems in libraries for compiler in cluster system.

Our program has a problem in scalability as we parallelized only the child nodes in the first level of dynamic game tree. Extending the parallelization recursively to nodes below the first level dynamically as figure followed can solve this problem.



6. References

[Sabot] Gary W. Sabot, High Performance Computing, Addison Wesley Pub., 1995.

Rainer Feldmann, "Game Tree Search on Massively Parallel Systems" Department of mathematics and computer science, University of Paderborn, 1993

Minimax and Alpha-Beta Template

<http://www.cs.caltech.edu/~petrovic/games/archex/othello.dir/node2.html>