Final Project Report

Zhiying Yao, Hongyi Ding

Summary

We implemented a parallel AI engine for the game of Chinese chess. We used OpenMP and ran our code on latedays. Since there exists a tradeoff between performance and speed, the maximum speedup our group was able to achieve was 2.81x using alpha-beta pruning and 1.84x using Monte Carlo Tree Search, under a relatively high winning percentage.

Introduction

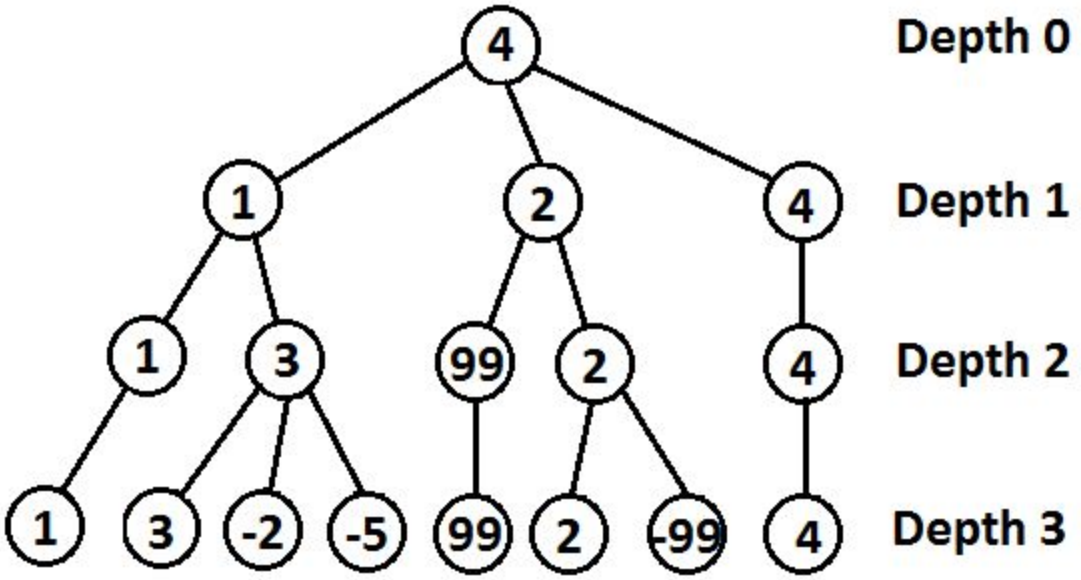
Chinese chess, or Xiangqi, is a strategy board game for two players. It is one of the most popular board games in China, and is in the same family as Western (or international) chess. The game represents a battle between two armies, with the object of capturing the enemy's general (king). Distinctive features of xiangqi include the cannon (pao), which must jump to capture; a rule prohibiting the generals from facing each other directly; areas on the board called the river and palace, which restrict the movement of some pieces (but enhance that of others); and placement of the pieces on the intersections of the board lines, rather than within the squares.

Compared to Western (or international) chess, Chinese chess is slightly more complex. According to Victor Allis (Allis 1994), Chinese chess has higher game tree complexity and a larger branching factor than Western (or international) chess.

Background

Minimax Strategy

Since Chinese chess is two-player, zero-sum game, we decided to start with the minimax strategy we learned in *15-451 Algorithm Design and Analysis*. We introduced the idea of a state, which is the game status at a certain point of the game. We chose an evaluation function, which takes a state and returns a quantitative value that captures the winning percentage based on the current state of the game. The evaluation function considers not only the pieces left on the board, but also their relative positions, since the value of each piece also depends on its position on the board. Based on this evaluation function, we are then able to apply the minimax search strategy.



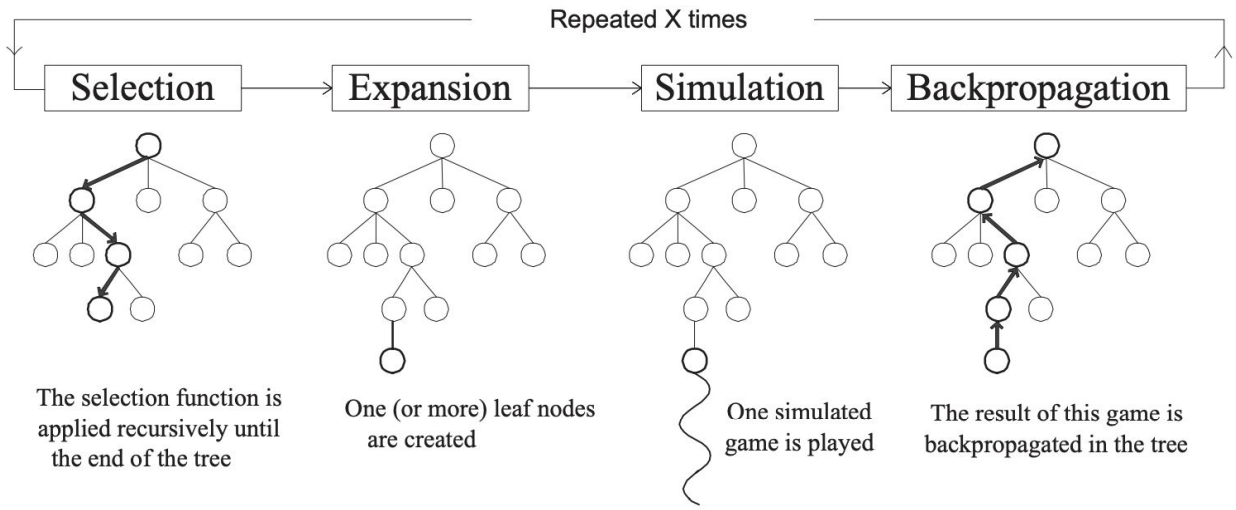
Alpha-Beta Pruning

However, according to Victor Allis, Chinese chess has a branching factor of 38, which is relatively large and affects the runtime of our program. The branching factor is the average number of children of each node, so at each depth, there are 38 possible moves on average a player can choose from, and the number of nodes to be explored increases exponentially as the search depth increases. Therefore, the number of nodes to be explored for our analysis is roughly the branching factor raised to the power of the search depth. In order to improve the performance of our program, we would like the search depth to some non-trivial number, which dramatically increases the number of nodes we are going to explore. Thus, before we map possible moves to machine concepts like threads, we would like to use a better algorithm to reduce the computation algorithm by a huge factor. Alpha–beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It stops completely evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision. In the game of Chinese chess, for example, moves that capture valuable pieces like the general or the chariots may be examined before moves that do not, or moves that have scored highly in earlier passes through the game-tree analysis may be evaluated before others.

As for the implementation, we used similar data structures as data structures used in Project 3 *Wire Route*. Specifically, we used a 2d array of integers to represent the board, and each piece was given an integral value according to its type. We maintained a list of all possible

moves and used the evaluation function to assign a quantitative value to each game state. Possible parallelism could occur when we were evaluate each game state. I.e., each thread could be responsible for some paths of the game tree. However, since we decided to use the alpha-beta pruning to reduce the computation time by a huge factor, it was important to keep the alpha value up-to-date. Therefore, there exists a tradeoff between performance and speed. If we would like better performance, then each thread is supposed to see a very accurate value of alpha, and it requires us to run the code sequentially. If we would like faster completion of the program, then each thread is running independently, and the result returned by the alpha-beta pruning algorithm is likely to be inaccurate, and the performance of our program drops.

Monte-Carlo Tree Search



In the MCTS algorithm, at each iteration, we select the next untried move for simulation and create a child. Once we have tried all the possible moves, we select the child node with highest UCB value, and continue selection from that child node. After we select the next child for simulation, we randomly choose a path of depth at most 10, and do a back propagation of the

simulation result. The data structure used to construct monte-carlo tree include a parent node, a list of possible moves, a list of children nodes, the move and the current player for that move. Possible parallelism could occur in the simulation step, so we can expand all the children and do the simulation at once. However, in this approach, we cannot utilize the information from other threads. For example, if over a half of the finished games are all losses, it will be highly probable that most games will lead to a loss, so the work down by the other half of slower threads might not be necessary.

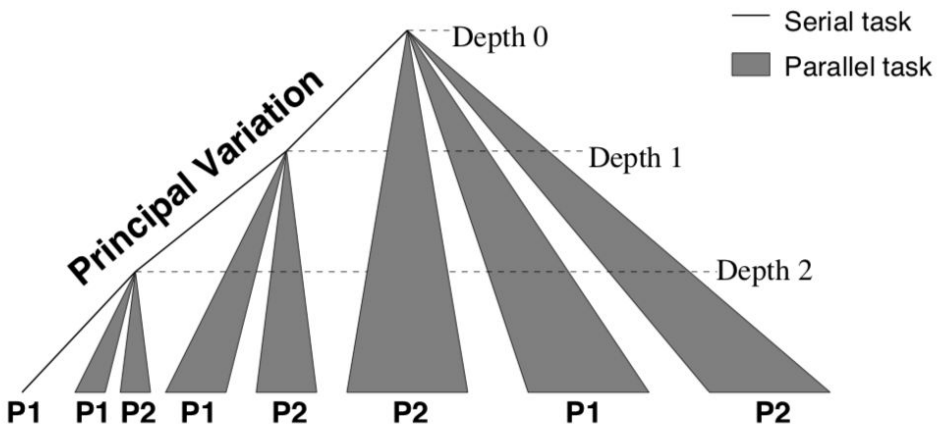
Approach

In order to exploit parallelism, it is important to assign processors to search multiple subtrees/paths of the game tree simultaneously. But the question is how to partition the tree for the reasons mentioned in the last section. We started with a naive parallel version of the alpha-beta pruning. However, both the performance and the speed of our program were poor, since each thread only saw a very stale value of alpha.

We then used an algorithm that is called principle variation search. The basic idea of the algorithm is that, it first recursively searches the left-most subtree at each node to find an alpha bound for the other subtrees, which can then be searched in parallel. Compared to our first attempt, there still existed two problems. The first problem was that the performance of the code still heavily depends on alpha. However, as long as the value of alpha used in each iteration is less than or equal to the value that would occur in the sequential version, the iterations can execute independently and the result will be correct. But still, as mentioned in the previous section, keeping the value of alpha as up-to-date as possible is critical for performance. Notice

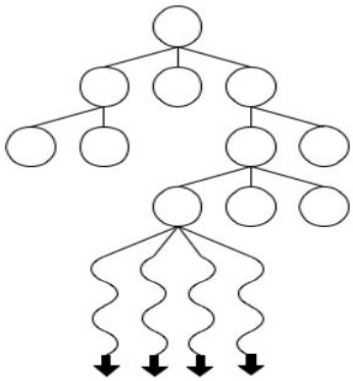
that based on the nature of the algorithm, if the best move is always on the left most subtree, then the parallel version of the principle variation search visits the same number of nodes as the sequential version of the alpha-beta pruning algorithm. In general cases, a weaker alpha cound is given at each iteration, and as a result, the parallel algorithm usually visits more nodes than the sequential version. Another shortcoming of principle variation search is that all of the parallel searches must complete before the current principle variation search invocation may return, which results in inefficient processor usage. However, since we are using OpenMP, we are able use it to dynamically schedule the tasks and resolve this issue.

How the data structures and operations map to machine concepts like cores and threads can be seen from the figure below. We changed the original serial algorithm to enable better mapping to a parallel machine.



We also implemented the parallel Monte Carlo Tree Search (MCTS) for the AI. The parallelism is over the simulation step. After we selected a children, we do all the simulations of next possible moves at the same time, a detailed view is shown in the next graph. However, since

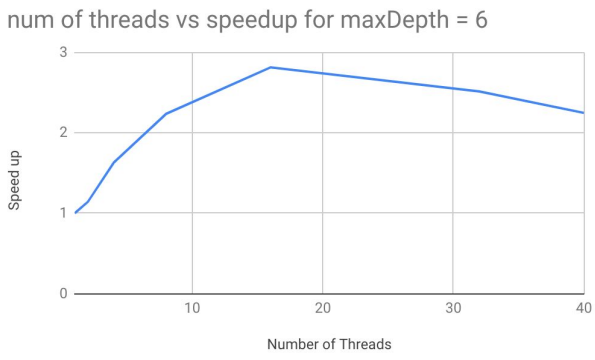
selection and back propagation requires non-trivial runtime, and it was done with single thread in our implementation, the speedup was not ideal.

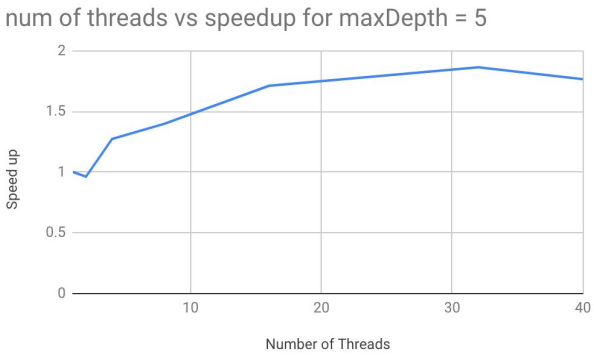


Results

We used speedup of generating the first move to measure the performance of our program. The size of the input is a standard chess board, and there are approximately available 44 next moves at each stage for each player. Whenever a user input a move sequence, the AI will generate a move that creates best result according to our algorithm.

In the alpha-beta pruning algorithm, the speedup graph of generating the first step is shown below. Our baseline was single-threaded CPU code.



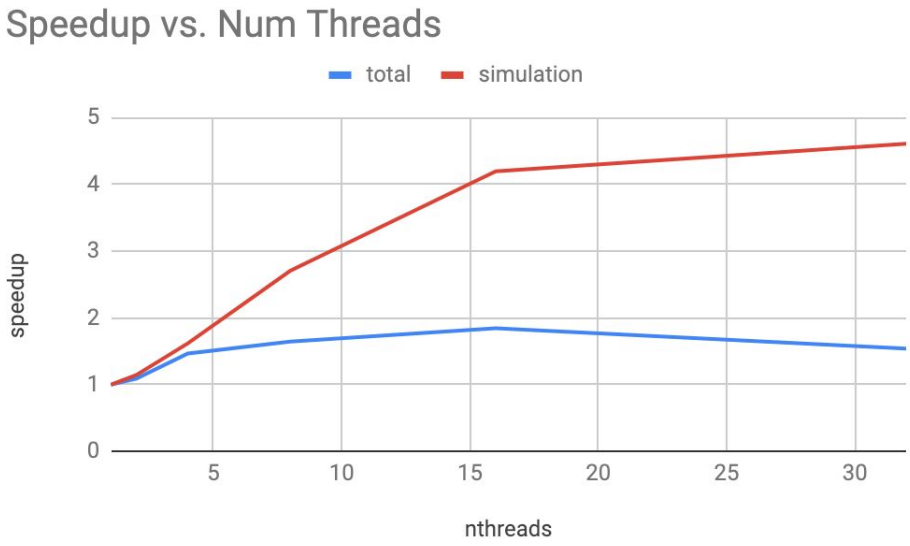


As we can see, the speedup drops after number of threads exceed 16. Since parallelism at level=0 is the most time-consuming one, we tested the time for each task, and also tested the total time for the parallel work at depth equals 0 and number of threads equals 16. We found out that in this setting, the most time consuming task takes 4.40005 seconds, and the whole parallel block takes 4.48843 seconds. Therefore, by the Amdahl's, even if we increase number of threads, there will be not much decrease in runtime when number of threads is greater than 16. Also, increase in threads led to increase in time of assigning tasks. We also found out at in the parallel region for current level equals to max depth, it is better to use sequential code, since at max depth, the computation is trivial, and assignment time will exceeds the benefits of having a parallel code.

Our algorithm was not able to achieve ideal speedup due to the nature of the algorithm. A weaker alpha bound is given at each iteration, and as a result, the parallel algorithm usually visits more nodes than the sequential one as we mentioned in the previous section.

We tested the runtime with maxdepth = 5 and maxdepth = 6, and the runtime for generating the first move for maxdepth = 5 is significantly smaller than maxdepth=6. This is because one more search depth indicates exponential growth of nodes. The runtime comparison for this two depth is attached below.

In the MCTS algorithm, the parallelism was across simulation. We break the parallelism in the simulation step, and we measured speedup for the simulation time and the total time for generating the first step. Our baseline was single-threaded CPU code. The speedup graph of both section is attached below.



As we can see from the graph the simulation speedup increase as number of threads increase, as expected. However, total speedup decreases after number of threads exceed 16. This is because when number of threads increases, the portion of time for selection and expansion increases a lot. As a matter of fact, when number of threads equals 16, the total runtime of generating the move is 12.991 seconds, while the runtime for selection and expansion is 9.14901 seconds. Again, by Amdahl's law, the speedup has a limit. For simulation time, we are not able to achieve ideal speedup because for each thread, after it finishes its simulation, it has to wait for entering critical section to write its result. Also, dynamic scheduling also requires time.

References

https://pdfs.semanticscholar.org/8e1b/da134a45bd362c61827e99fd1e2cb624079d.pdf

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.159.4373&rep=rep1&type=pdf

List of Work by Each Student, and Distribution of Total Credit

50% - 50%

50% - 50%