Parallelization of a Computer Go program

Waren Kemmerer, Abdul Dakkak
Stevenson Jian, Palash Sashittal

Abstract

Go is an ancient Asian board game with simple rules that produce extremely complex gameplay. There has been a vast amount of research on strategy and tactics involved in the game by both contemorary professional players and computer scientists. For this project we parallelize a Computer Go program called 'Pachi'. Pachi is a state of the art implementation of the Monte Carlo Tree Search algorithm for the game Go. We parallelize the Monte Carlo implementation using OpenMP. We also present the performance of our parallel code compared to Pachi run in sequential manner. A clear improvement in performance is observed.

1 Introduction

The board game of Go has proved to be an exciting challenge in the field of Artificial Intelligence. The game became well known among researchers since it turned out to be much more difficult to write a strong Go program than for example a strong Chess program. Some consider beating the top human Go players one of the grand challenges of Artificial Intelligence.
Although traditional game tree search techniques don't work well in Go, recent advances have shown that Monte Carlo Tree Search (MTCS) works extremely well for Go. The algorithm progresses from the current position by first expanding the game by picking a set of randomly chosen moves. It then simulates the game plays for both itself and the opponent. Depending on the outcome of all the explored moves, the most favourable move is chosen to play.
There are heavy constraints of time in the game Go. The game length is fixed at 60 minutes per player. Thus the computer must make decisions quickly and so speed of the algorithm is critical and exploring multiple game moves in parallel can be very advantageous.
In the next section we present the rules set and basic concepts of the game Go. Then we follow up with an overview of Monte Carlo approach of playing Go and different ways of parallelization. Finally we show in the results section that the parallelized program is much better than the serial version.

2 What is Go ?

Go is a board game involving two players that has its roots at least in the 6th century B.C. of ancient China. It is probably the oldest board game in existence and is considered one of the four essential arts of a cultured Chinese scholar in anitquity. The number of possible games is vast (10^{761} compared to only 10^{120} possibilities in chess), despite having simple rules. The is significant strategy and tactics involved and mastering them requires many years of study.

2.1 Rules

Go is palyed on a square grid of a given size. $19X19$ is the most popular, but smaller boards like $13X13$ and $9X9$ are also used. Two players alternately place black and white stones on the vacant intersections of the grid lines on the boards which are called "points". The black makes the first move.

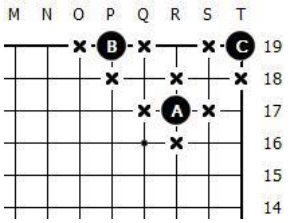

Figure 1: Example figure showing liberties for stones on the board

Once played on the board, the stones may not be moved. Stones of the same colour that are adjacent and directly connected form a group. A group shares liberties, which are vacant points adjacent to a group. When a group runs out of all liberites, then it is captured and the stones are physically removed from the board. For example, in figure 1, the liberties for stones **A**, **B** and **C** are marked with crosses. A basic principle of Go is that stones must have at least one liberty to remain on the board. An enclosed liberty is called an eye and if a group has two separate eyes then it is said to be completely alive. Such a group cannot be captured even if surrounded.

Figure 2 shows two different cases. In case **A**, we have a group of black stones which are not in immediate danger of being caught. A white move at **A** is not allowed. This is because it is a suicide move which is dicussed later. If white however were to play at **B**, it would capture all the black stones surrounding **B** since they will have no liberties left and will be removed from the board.

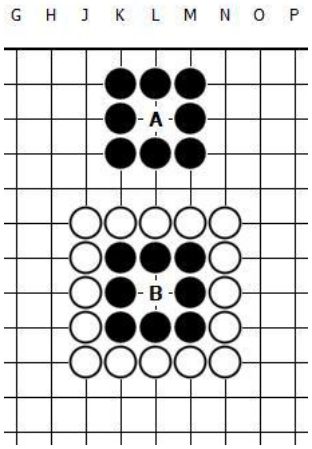

Figure 2: Example figure showing how to capture a group

Aside from the rule of liberty, the other basic rule in Go is the *ko* rule. It states that the stones on the board must never repeat position of stones in the previous move. Moves which would do so are forbidden and thus only moves elsewhere on the board are permitted.

In Figure 3, in situation **A**, when it is white's turn to play, he can capture the black stone by playing his stone in the vacant point surrounded by the black stones. If he does that the board will look like case **B** in the same figure. In this case, black can capture the white stone which will lead us back

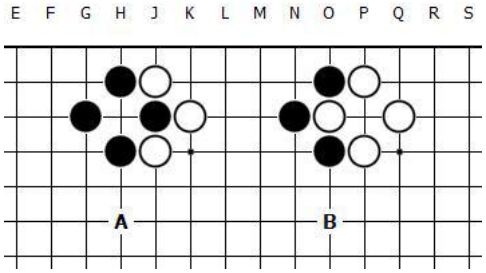

Figure 3: Example figure showing *ko*

to situation **A**. Thus, if the players want, they can stay stuck in this loop forever. This is where the ko rule comes in. This rule doesn't allow any player to bring the board back to the previous board situation. This means that while white can capture black's stone to go to situation **B**, black cannot capture white's stone. He must first play a move somewhere else on the board, so that on his next turn the whole board will have changed and then he can finally capture the white stone.
An extension to *ko* rule is the *superko* rule. This rule states that a play is illegal if it would have the effect (after all steps of the play have been completed) of creating a position that has occured previously in the game. However, it doesn't bar a player from passing. This ensures that the game eventually moves to an end by preventing indefinite repition of the same positions.
The objective of the game is to have surrounded a larger total area of the board with one's stone than the opponent by the end of the game. The number of points occupied or completely surrounded by stones of a player (and komi which are some points given to white player as a compensation for playing second) is the score and the player with higher score wins the game. The game ends when either players play a pass move. Games are also won by resignation.
Although there are minor differences between rule sets used in different countries, these differences do not change the tactics of the game. For example, some rule sets allow suicide moves while some do not. A suicide move is when a player places a stone such that it has no liberties on the board. This kind of play will only be useful in a limited set of circumstances.

3 Monte Carlo in Go

Monte Carlo methods invoke the law of large numbers by performing many randomized simulations and selecting the most likely answer gained from the largest percentage of the performed simulations. We use randomized simulations to evaluate Go positions and build a game vector called a *playout*.
Each random move is evaluated by a number of *playouts*. In each playout, the games are played-out to the very end by selecting moves at random. The move that gets the maximum number of wins at the end of the playouts is chosen.

3.1 Exploration vs Exploitation

A simple Monte Carlo algorithm gives more importance to exploration than exploitation. Exploration is trying out larger section of the sample space and exploitation is focussing on simulations of the best candidate moves. Thus, while exploration involves playing out more moves at a particular state on

the board, exploitation involves trying out different playouts for the same move to make sure that the win/loss ratio assigned to that move is correct.
This flaw is nullified by running a lot of Monte Carlo engines in parallel. This ensures that each moves gets sufficient number of playouts and will be discussed in greater detail in next section.

4 Parallel Monte Carlo

In this section we describe the parallel design we have implemented in the code. Each thread performs its own randomized simulation of the game. Number of playouts is determined based on time constraints or until a good move is found. The moves are identified by the position at which the stone is placed. The wins and losses encountered as result of the random game for any move are stored in a variable that is visible to all the threads.
A reduction is performed on the losses encountered, good moves explored and superko positions encountered across all the threads. The playouts that end in superko positions are simply ignored unless superko is the only possible outcome. If there are no good moves explored across all the threads, the computer just passes for that turn.
Since each thread runs its own random simulation which would be most likely be different than the rest of the threads, we are able to exploit the favourable moves to a greater degree as compared to a serial Monte Carlo code. This increases the expectation of the win/loss ratio assigned to the moves.

5 Monte Carlo Tree Search

MCTS is a heuristic search algorithm which is used most notably in game palying. Unlike regular Monte Carlo, it concentrates on analysing the most promising moves, basing the expansion of the search tree on random sampling of the search space. MTCS weighs the nodes of the game tree based on the outcome of playouts so that the better nodes are more likely to be chosen in future playouts.

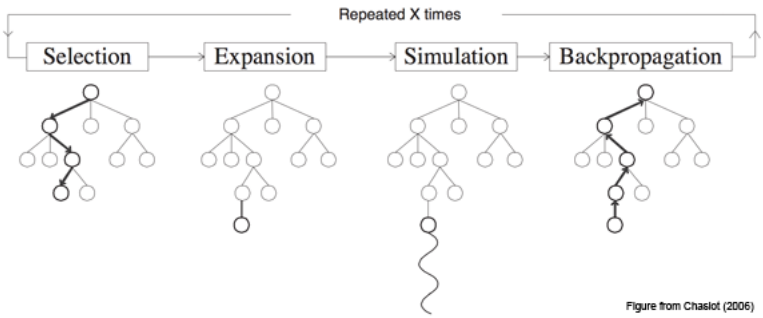

Figure 4: Schematic describing the MCTS algorithm

Each round of MCTS consists of four steps :

- Selection : starting from the Root, select successive child nodes down to a leaf node.
- Expansion : unless the leaf node engs the game, create more child nodes and select one of them
- Simulation : play a random playout at the selected node

- Backpropogation : update the information in the parent node based on the result of the playout

Figure 4 shows the four steps involved in MCTS. Such rounds are repeated as long as the time alloted for the move runs up. The node with the highest weight is chosen for the move.

6 Parallelization of Monte Carlo

In this section we look at three different approaches to parallelization of MTCS depending on which step of the Monte Carlo Tree Search is parallelized.

6.1 Leaf Parallelization

In leaf parallelization, only one thread traverses the tree and adds one or more nodes to the tree. Starting from the leaf node, independent simulated games are run parallely on each available thread. When all the games finish, the result of all these games is propogated backwards by a single thread.

6.2 Root Parallelization

Root Parallelization works as follow. It consists of multiple MCTS trees growing in parallel with one thread per tree. Similar to leaf parallelization, the threads do not communicate with each other while the games are simulated. When the available time is spent or the playouts are over, all teh root children of seperate MCTS trees are merged and results from the games played are added. The best move is selected based on this grand total.

6.3 Tree Parallelization

Finally in tree parallelization, a common tree is shared from which several simultaneous games are played. Each thread can modify the information contained in the tree and therefore this method of parallelization requires communication between threads. Hence this method is much more difficult to implement compared to other methods of parallelization.

References