CS87 Project Report: Parallel Monte Carlo Tree Search

Michael Davinroy, Shawn Pan, Jimmy Shah
Computer Science Department, Swarthmore College, Swarthmore, PA 19081

July 31, 2018

Abstract

Several recent advances in the field of artificial intelligence suggest that sequential Monte Carlo Tree Search (MCTS) offers an elegant solution for games with large search spaces such as Go, where the size of potential game states is too large to be searched exhaustively. MCTS finds optimal decisions by combining the generality of random simulation with the precision of tree search. MCTS utilizes random simulations to explore promising paths and the quality of the algorithm improves with additional simulations. Parallelization can provide performance improvements to MCTS because using more computing time allows the algorithm to visit more nodes and explore more promising paths. Different parallelization methods have been widely discussed in the AI community, and three common parallel approaches are root parallelization, leaf parallelization, and tree parallelization. Root parallelization constructs multiple game trees simultaneously, while leaf parallelization performs the simulation stage in parallel from the leaf nodes. Tree parallelization is the same as root parallelization, but each process shares the same game tree state.

We present two novel techniques for parallelizing Monte Carlo Tree Search. Our solutions use MPI to capitalize on the benefits of root parallelization, but go further in that they search more extensively promising move paths. Our first implementation, Top-K parallelization, performs two rounds of simulation. In the first round, Top-K parallelization runs MCTS for a given number of exploratory rollouts on child processes and then more fully explores the top k moves in a second round, splitting work evenly among child processes. Smart-K parallelization uses a similar narrowing of search on promising moves, but delegates work to child processes by calculating a weight based on current move values and number of visits. Our experiments show negatives results for Top-K parallelization and promising results for Smart-K parallelization.

1 Introduction

In Artificial Intelligence problems, game trees play an important role as a structure in representing states and possible outcomes. They are often used in studying games, given that they represent possible moves and strategies to win. Traditional tree search algorithms, such as minimax search [3] or brute force search, prove extremely ineffective on games with large game trees that have no efficiently computable or reliable heuristics (a heuristic being equivalent to a rule of thumb for which player is currently winning for a given game state). Therefore, algorithms such as Monte Carlo Tree Search were developed to combine the generality of random simulation with the precision of tree search.

Instead of conducting a thorough tree search, MCTS executes simulations of games until reaching an end state to estimate the values of its potential moves. Therefore, the more that MCTS explores the game tree, the better its solution. MCTS always guarantees a solution (although not necessarily the best one), and the quality of the decisions improves as the algorithm runs longer. MCTS algorithms are a promising way to tackle various AI challenges and have proven extremely effective in games with large game trees and no efficiently computable or reliable heuristics.

Because the quality of the decisions improves as the algorithm runs longer, current researchers have suggested three different methods of parallelization: Leaf parallelization, Root Parallelization, and Tree Parallelization (all explained in a later section). In this article we present two novel techniques for parallelizing the MCTS algorithm: Top-K and Smart-K. We first compare two common parallelization methods (leaf and root) against serial MCTS. We do this by using a *winning percentage* metric on game Hex (where winning percentage is the percentage of games won by each player) to understand the strength gained from parallelizing certain phases of MCTS. Then we play our novel versions, Top-K and Smart-K, against serial and root parallelized versions to evaluate their performance.

In Section 2, we present the basic structure of MCTS. In Section 3, we discuss different parallelization methods of MCTS. In Section 4, we share the findings from existing research as motivations for our work. In Section 5, we discuss our process and solution in detail. Then in Section 6, we empirically evaluate our parallelization methods, and we present our conclusions in Section 7.

2 Monte-Carlo Tree Search

The MCTS algorithm is based on randomized exploration of the search space. Through recording the results of prior explorations, the MCTS algorithm grows its game tree in memory and successively makes more promising moves.

The MCTS algorithm consists of four phases: selection, expansion, simulation, and backpropagation. These four phases are repeated until a timeout is reached or a certain number of iterations (rollouts) is reached.

2.1 Selection Phase

During the selection phase, the algorithm will start at the root node of the game tree, and explore children nodes according to a default tree policy. In this case, MCTS uses the Upper Confidence Bound (UCB) as a metric to make decisions. The decisions will influence the direction of growth for the tree as the game progresses. One trade-off is the decision to explore more paths or exploit a given optimal move to find further optimal moves. The UCB formula aims to balance both by guaranteeing to be within a constant factor of the best possible bound on the growth of regret, which is the expected loss due to not playing the best choice [2]. The policy for selecting child nodes is defined as:

$$UCB = v_i + C \cdot \sqrt{\frac{\ln N}{n_i}}$$

where v_i is the value estimate of the child node, C is a tune-able parameter, N is the total number of times the parent node has been visited, and n is the number of times the current child i has been visited. Additionally, all children of a given node are visited at least once, since we expand currently unexplored child nodes of a given node before further exploring the other child nodes. Adjusting C changes the amount of exploration governed by the policy [2].

The exploration-exploitation trade-off is explained as follows. The denominator increases as more nodes are visited, thus reducing the effect of exploration. However, as more children of the parent node are visited, the numerator increases, thus exploring more siblings becomes valued. The exploitation is determined by v_i, which is the current value of a node. Selection continues until a terminal node is reached or a node that is not fully expanded is added to the queue, at which point MCTS enters the expansion phase.

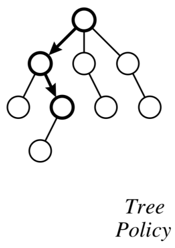

Figure 1: Selection Phase, *Browne et al*

2.2 Expansion Phase

During the expansion phase, MCTS looks at each action available to a node and selects an unexplored child node uniformly at random to become a new leaf node of the tree.

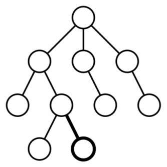

Figure 2: Expansion Phase, *Browne et al*

2.3 Simulation Phase

During the simulation phase, we produce a reward value for the leaf node described above. These simulations are often random moves played until an ending game state and the value is usually determined to be a 1 or -1 depending if the player won or lost, respectively. After the search ends, simulation is complete and the reward value for the action is backpropagated as described in the next section.

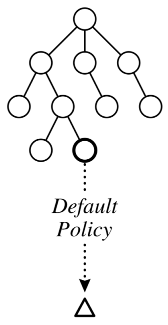

Figure 3: Simulation Phase, *Browne et al*

2.4 Backpropagation Phase

In the backpropagation step, the reward of the action from the child node is added to a running average value for all nodes along the path to that child node. After a given number of rollouts of these four phases, the optimal action from the root node (representing the current state of the game) can now be selected, since we have estimates of optimality for each possible action of from the root node. Different criteria can be used to determine the optimal action, but usually the move with the highest estimated value is chosen.

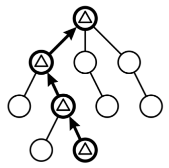

Figure 4: Backpropagation, *Browne et al.*

3 Parallel Monte Carlo Tree Search

Current research [2] [3] [4] [5] has shown that different phases of the MCTS algorithm can be parallelized, and parallelization can further improve the performance of a player making decisions based on the simulation of the game tree. Therefore, the motivation behind this work is to parallelize the MCTS algorithm to improve player performance. In addition, we will compare the results of the parallelized MCTS algorithm in the context of Hex and examine the benefits of parallelization in different phases.

3.1 Leaf Parallelization

In the simulation phase, independent simulations are played from the leaf node for all the available child processes. When all simulations terminate, the results of simulations are passed back to a single master process, and the master process backpropagates a combined result up the game tree. Generally, the most common result among the child process simulations or a combination of those results are backpropagated.

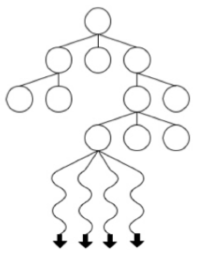

Figure 5: Leaf Parallelization, *Browne et al.*

3.2 Root Parallelization

Root parallelization consists of creating multiple MCTS trees in parallel, with an independent tree per process. Like leaf parallelization, the processes do not share information with each other. This parallelization method requires minimal communication overhead among processes and therefore the parallelization is almost embarrassingly task parallel. When each process explores their respective game tree for a given number of rollouts, they each return some information about their independent simulation. These results are then combined in some way, usually by adding the values of children nodes discovered by the child processes, or by taking a vote over the best found move of each child process. Some articles, such as [3] suggest that the master process could merge all individual game tree from other processes and makes the best move according to the merged tree. In our implementation, we decided not to complete the merge step because it might be too costly due to the large volume of data that would need to be passed around. We found the voting protocol to be most effective in our root parallel implementation.

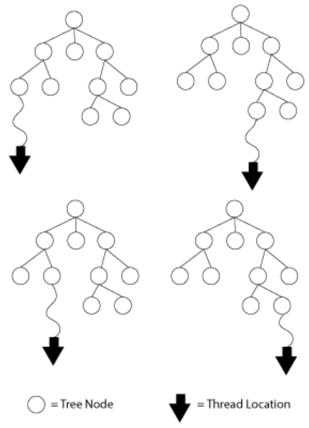

Figure 6: Root Parallelization

3.3 Tree Parallelization

A Tree parallelization is a variant of root parallelization where each process shares the same game tree state. Because each process is accessing and potentially modifying the same game state tree, there needs to be a lot of precaution in preventing race conditions and data corruption. Therefore, in a typical tree parallelization setup, both global and local mutexes are often used.

The experiments done by Chaslot et al. [3] have shown great promises of tree parallelization. However, we decided not to implement tree parallelization in our research because creating an abstraction of shared memory would be too costly in our setup.

3.4 Top-K Parallelization

The first novel technique that we present here is Top-K parallelization. It is a variant of root parallelization that runs for two distinct rounds and it is depicted in Figure 8. In the first round, Top-K runs root parallel MCTS for some n number of exploratory rollouts on its child processes.

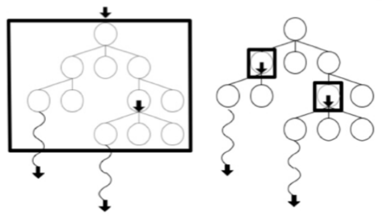

Figure 7: Tree Parallelization, *Browne et al.*

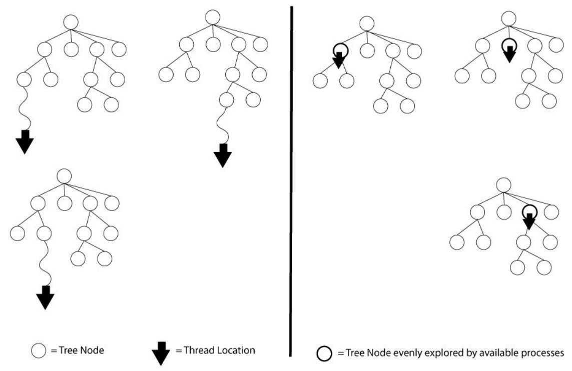

Figure 8: Top-K Parallelization

In the second round, it explores exclusively the top k moves from the previous round. The top k moves are strictly decided upon by their respective predicted values from the first round. Once we have obtained these top k child nodes, we evenly divide our processes among these nodes for further exploration. Then, we run MCTS with these child nodes as the root node of these simulations on the child processes. Once we have completed m number of rollouts per child process, the child processes return the predicted value of the child node they explored to the parent process. The parent node then combines the values of these nodes and then selects the child node with the highest value as the final move. Both m and n are tune-able parameters.

We hypothesize that Top-K parallelization outperforms our serial implementation given the same number of rollouts in each process. In addition, since it is a derivation of root parallelization, we hypothesize that Top-K parallelization performs comparably to root parallelization given same number of rollouts and processes.

3.5 Smart-K Parallelization

The second novel technique that we present here is Smart-K. It is another variant of root parallelization that runs for n rounds, each for m rollouts, and its phases are depicted in Figure 9. Like Top-K, Smart-K further explores its children nodes after a preliminary parallelization from the current root node. However, unlike Top-K, Smart-K does not evenly distribute its processes in further searching. Smart-K consists of three distinct phases. In the first phase, Smart-K simply does a single round of a root parallel MCTS simulation for m rollouts on each of its child processes. The child processes then pass back the values and visits of the explored child nodes for use in future rounds of simulation. In the second phase, Smart-K runs for $n-2$ rounds and the master process does the following: normalize the values of child nodes and distribute processes based on these normalized values; run MCTS for m rollouts from the child node assigned to each child process; have the child processes send back the values and visits of their explored child node; have the parent add the values and the visits for each child node to their current values; and then repeat the same steps for the remaining $n-3$ rounds. In the third and final phase, Smart-K does one more root parallelization like the first phase, but instead of combining the values and visits with the already known values, each child process decides what the best move is based on highest value and sends this move and the number of visits to that move to the parent process. The final step is then for the parent process to select the most voted for move to play, breaking ties based on a higher number of visits.

Our hypothesis is that our Smart-K Parallelization outperforms our serial implementation given the same number of rollouts in each process. Moreover, because Smart-K parallelization contains synchronization that allows the parent

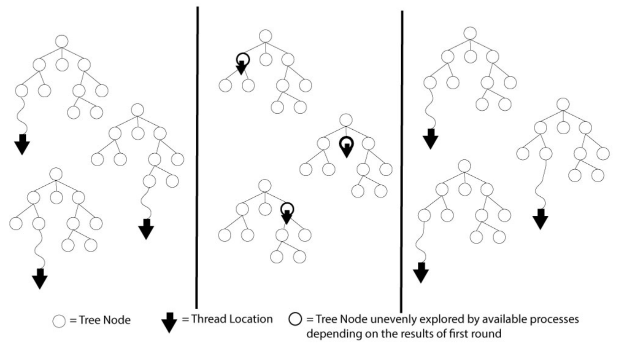
= Tree Node
= Thread Location
= Tree Node unevenly explored by available processes depending on the results of first round

Figure 9: Smart-K Parallelization

process to iteratively update its whole game tree, which then gets passed to child processes, we expect that Smart-K parallelization outperforms root parallelization given same number of rollouts and processes.

4 Related Work

Monte Carlo Tree Search has had a profound impact on AI approaches due to various traits, such as being a statistical online algorithm (an algorithm that uses probability to do simulations of future events and that develops plans for actions while it is currently running instead of developing an entire plan before starting to play) and the lacking the need for domain specific knowledge (strategic knowledge about the game or task it is performing). Parallelizing MCTS seems a natural way to further explore the benefits of MCTS in decision applications and there has been research done to compare the effects of different parallelization methods, such as leaf parallelization and root parallelization. Since a MCTS algorithm consists of four main phases: Selection, Expansion, Simulation and Backpropagation, parallelism has the potential to be applied to different or even multiple sections of the algorithm. There has been existing research done to compare the benefits of parallelism in each phase. Chaslot et al. demonstrate that root parallelization performs better compared to leaf parallelization under CPUs.[3]

The teams' findings also showed that Root Parallelization outperforms the sequential MCTS algorithm, with the other methods trailing behind, as shown in the figure adopted below.

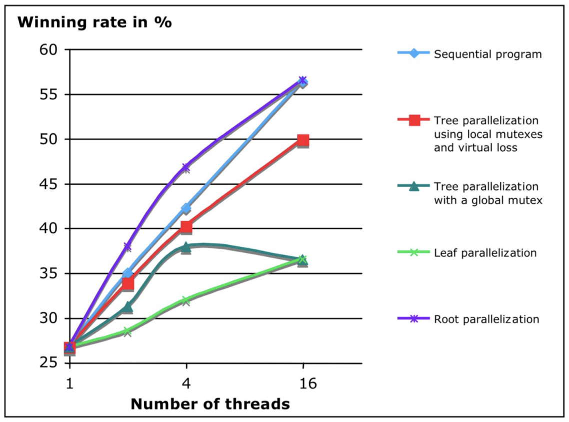

Figure 10: Parallelization Methods Performance, *Chaslot et al.*

As Chaslot et al. concluded, leaf parallelization was the weakest parallelization method, while root parallelization was the best [3]. Tree parallelization can be effective if the mutexes are implemented locally. Other researchers have also expanded parallel MCTS to scale across multiple CPU's. Rocki and Suda [5] explored the benefits of root parallelization in MCTS algorithms. In their implementation, a master process broadcasts the input data (state/node) to the other processes and is in charge of collecting the data as well. Moreover, it is implemented through MPI to take advantage of cluster systems such as the TSUBAME supercomputer. Through their experiments, the researchers find an almost linear speedup on the simulation time as the number of cores increases. Therefore, the experiments demonstrate that the root parallelization is very scalable in terms of multi-CPU performance. In addition, the paper concludes that, due to the low communication overhead from MPI, dedicating more CPU cores instead of utilizing sequential MCTS search time yields significant parallel performance increase. Overall, this paper offers the root parallelization as the promising and efficient way to parallelize the MCTS algorithm.

In addition, there are some preliminary studies [1] that suggest communication among processors during the parallelization process is beneficial to achieve optimal performance. Their study demonstrates that synchronization of best performing nodes with small depth (up to 3) consistently beats the root parallelization. Therefore, their experiment results shed light on the benefits of synchronization in the parallelization process.

5 Our Solution

In our project, we planned to answer the question of "Does parallelizing Monte Carlo Tree Search lead to significant gains in player performance as compared to a sequential implementation, and if so, will the parallelization implementations we create perform better than commonly studied parallelization techniques?" To do this, we implemented several different parallel versions of Monte Carlo Tree Search using the the python multiprocessing library and mpi4python and then played them against each other to obtain winning percentages. We expected that parallelizing Monte Carlo Tree Search in any fashion would give some performance improvement to players performing this algorithm over our sequential approach.

We started this research with a sequential implementation of MCTS written in python that plays the game of Hex. The first parallel implementation that we developed is a leaf parallel version of MCTS that has an abstraction of shared memory to synchronize data. However, after attempting to test this implementation's performance against our sequential version, we found that the leaf parallel version was excruciatingly slow to make a move even when only executing 1000 rollouts (the same number as the sequential version). Through further experimentation and changing implementations, we found that overhead in the simulation phase is incredibly costly because simulation must be carried out until the end of the game for every rollout. For leaf parallelization in particular, each rollout entails spawning new processes and then collecting their results, leading to significant overhead. We also believe that the overhead of the abstraction of shard memory is too great of a cost in this bottleneck phase of MCTS. Therefore, we abandoned further testing on a multiprocessing, leaf parallel implementation.

After noticing this bottleneck problem of our leaf parallel version, we decided to focus on root parallelization using mpi4python. This is where we saw our first significant performance enhancement. By simulating multiple games across multiple processes, we were able to see the players consistently play better against the serial version, even for as little as 16 child processes. At 128 child processes, our root MCTS player always wins against the serial version. Moreover, we decided to increase the UCB constant in the root parallelization because a small UCB constant could potentially cloud the added benefit of more processing power and explorations due to parallelization.

In addition, inspired by previous research and various experimentation, we implemented a voting mechanism to decide the best move by utilizing results obtained from the child processes. We speculate that the serial version usually finds a good move, but random simulations may sometimes lead to unpromising paths. Therefore, by voting over what the child processes estimate is a good move, the player negates the error associated with some process or a few processes happening to simulate unrealistic games. Finally, we observed that the root parallel version takes almost the exact same time to make moves as the serial version because only two messages need to be exchanged between any given child process and the parent process, namely at the beginning of the simulation and at the end of the simulation for each move made.

After we implemented the root parallel version using mpi4python, we implemented a leaf parallel version in a similar manner. Like the multiprocessing leaf parallel version, this MPI leaf parallel implementation suffered from a large amount of overhead in the simulation phase. In the case of MPI leaf parallelization, the overhead originated from the large number of message exchanges among all the child processes in every rollout. Since the large overhead prevented a MPI leaf parallel player from making a move in reasonable time as well, we abandoned leaf parallelization for the rest of the project. This is not to suggest that leaf parallelization is not helpful, but that in our setup, efficiently performing it was not feasible. We would like to explore leaf parallelization further in the future if we port to a setup that better supports efficient shared memory.

The final, and most significant, portion of our solution to our problem comes in the form of the two approaches to improving root parallelization: Top-K and Smart-K parallelization. As mentioned above, these two implementations attempt to improve upon root parallelization by utilizing knowledge from previous rounds in future rounds more effectively than root parallelization. Namely, through first completing a round of exploratory rollouts, they further explore promising moves to a greater extent that root parallelization does. We discuss the effectiveness of these approaches in the following results section.

6 Results

In this section we compare different parallelization methods with each other and against serial version.

6.1 Experimental Set-up

The goal of the experiments is to measure the quality and scalability of the parallelization process. The primary measure that we investigate is the *winning percentage* in the game of Hex. Through measuring the *winning percentage* of an implementation against another, we will be able to understand the relative strength of different parallelization methods. Because we are investigating the merits of implementation and each implementation may have different initialization/communication overhead, we decided to set the number of rollouts as a variable rather than setting a timeout for simulations. For instance, in obtaining the optimal move, our Smart-K implementation may take a few more seconds to make a decision than Root Parallelization due to its additional MPI communication among processes. However, if these parallelization methods are utilized in practice, these differences are insignificant. Therefore, through allowing the parallelized MCTS to fully explore the game tree for a set number of rollouts, we can then accurately judge their relative game playing performance.

All our experiments are run on machines inside Swarthmore Computer Science labs. The number of cores of those machines ranges from 4 to 12. In addition, we utilize the mpi4python package in python to parallelize the MCTS simulations across available machines in the network. All experiments are run 50 times to obtain a winning percentage for players being played against each other.

In the case of leaf parallelization and root parallelization, we set the number of rollouts to be 1000 so that parallelized and serial versions complete the same number of rollouts per processor. The leaf and root parallelization experiments are to test our hypothesis that parallelization will generally lead to better player performance than a serial MCTS player. Our later experiments are to test whether Top-K and Smart-K parallelizations show significant performance gains over our root parallel implementation.

6.2 Leaf Parallelization

In the first round of experiments, we tested the performance of leaf parallelization and the results are given in Table 1. We observe that running 16 and 32 simulations leads to a winning percentage of 25% and 40%, respectively, against the serial MCTS. That indicates that serial MCTS beats leaf parallelized version most of the time despite the increased simulation steps. Therefore, we conclude that the benefits brought by parallelizing the simulation phase are not significant.

Number of processes	Winning percentage
16	25%
32	40%

Table 1: Results of Leaf Parallelization against Serial

6.3 Root Parallelization

In the second part of our experiments, we tested the performance of root parallelization against our serial version. The results are given in Table 2.

Table 2 indicates that root parallelization is an effective way to parallelize MCTS. When run on 16 processes, root parallelization beats the serial version 92% of the time while leaf parallelization beats only 25% of the time. Additionally, when we run on 32 processes, root parallelization beats serial 98% of the time, while leaf parallelization only wins 40% of the time. This suggests that constructing an entire game tree from a root node is more effective than running multiple simultaneous simulations in the simulation phase. This increase in performance could be due to the UCB constants guiding each child process to explore different parts of the game tree and, therefore, have the parent process return the globally optimal best move. Moreover, when we ran our root MCTS against serial MCTS with root utilizing 64 and 128 processes, root MCTS won 100% of the time. Finally, as we multiplied the number of processes from 16 to 128, we hardly noticed any increase in the total runtime of the simulations. Therefore, we conclude that root parallelization is a simple and extremely effective way to parallelize MCTS.

Number of processes	Winning percentage
16	92%
32	98%
64	100%
128	100%

Table 2: Results of Root Parallelization against Serial

6.4 Top-K Parallelization

In the third series of experiments, we tested the performance of Top-K parallelization against serial implementation and the results are in Table 3.

When run on 128 processes, Top-K lost to serial version in all fifty experiment runs. This rejects our hypothesis that Top-K would outperform serial implementation. Because additional processes ideally allow more game simulations and improve the performance of parallel implementation, we conclude that our Top-K implementation doesn't fully take advantage of parallelization, we and decided not to run experiments with fewer processes on Top-K against serial version. In addition, given the results from Table 2 on root parallelization against serial, we decided not to run any experiment playing Top-K against root parallelization because the experiments against serial MCTS already allowed us to fully gauge the effectiveness of Top-K.

Number of processes	Winning percentage
128	0%

Table 3: Results of Top-K Parallelization against Serial

Number of processes	Winning percentage
64	100%
128	100%

Table 4: Results of Smart-K Parallelization against Serial

6.5 Smart-K Parallelization

In the last series of experiments, we tested the performance of Smart-K parallelization against serial and root parallelization, and the results are in Table 4 and Table 5, respectively.

From Table 4, we see that Smart-K outperforms the serial version in all of our experiments with both 64 and 128 processes. This confirms our hypothesis that our Smart-K implementation outperforms serial version without incurring significant runtime slowdowns. Therefore, Smart-K parallelization improves the performance of MCTS significantly over that of our serial version. These results are similar to the results seen for root parallelization in Table 2.

Next, we decided to compare the performance of Smart-K parallelization against root parallelization. In our experimental runs with 128 processes, Smart-K parallelization reaches a 52% winning percentage and, therefore, performs comparably to root parallelization. This rejects our initial hypothesis that Smart-K parallelization would outperform serial implementation due to its additional feedback across different iterations. At the same time, it is too early to conclude that Smart-K is an ineffective way to parallelize MCTS. We believe that additional tuning of our Smart-K implementation will allow game trees in the child processes to better account for promising nodes and thus allow the parent process to make better overall choices when selecting a move to make. Since there is no synchronization of collected data in root parallelization, we speculate that Smart-K has the potential to outperform root parallelization with future work.

Number of processes	Winning percentage
128	52%

Table 5: Results of Smart-K Parallelization against Root Parallelization

7 Conclusions and Future Directions

In this report, we first presented leaf parallelization and root parallelization and compared their performance against our serial implementation. We showed that parallelization can indeed improve the performance of MCTS algorithm. Then we introduced two new parallelization methods: Top-K and Smart-K. Our Top-K implementation consists of two rounds of parallelization where it performs root parallelization in the first round and evenly divides available processors to explore top k moves in the second round. Our motivation behind Top-K is to more deeply explore promising paths after the initial root parallelization and then select the best move. However, from our experiments, we see that Top-K parallelization routinely loses to our serial implementation. It is too early to conclude that Top-K is ineffective and we speculate that the number of rollouts in our experiment isn't big enough to reap the benefits of Top-K implementation. We also believe that Top-K may be narrowing its search prematurely and therefore may only be finding a locally optimal move instead of the globally optimal move. In addition, it is possible that Top-K may perform better in games with larger game states, such as Go, so that its concentrated depth exploration can gain more benefits. We would like to test this in the future.

Our Smart-K implementation performs equally well as root parallelization and shows promise in its performance and scalability. In addition to performing a preliminary root parallelization, it allows extensive explorations of promising next moves from the root node and records the values of those children moves. Such synchronization obtains additional game state information that doesn't exist in the root parallelization. Current research [1] suggests that synchronization of child nodes is beneficial in improving the performance of parallel MCTS and we believe additional tuning of the parameters of Smart-K will allow it to outperform root parallelization. Currently, only the values of child nodes of the root node are passed back to be merged in one tree. Potential future work is to allow children with promising values of depth up to 3 to be passed back so that the merged tree has information on more children nodes. In addition, rather than completing the merge process after each process completes its turn of rollouts, intermittent synchronization and merging within iterations of rollouts may prove beneficial because each child process then is notified of promising paths and avoids visited and unpromising paths.

Overall, we see that parallelization shows great promise in improving the performance of the MCTS algorithm. In particular, root parallelization serves as a simple and effective method to parallelize MCTS, and we believe that Smart-K has the potential to outperform root parallelization with future improvements.

8 Meta-discussion

To begin, we set up the libraries that we would need to develop our project, namely the python multiprocessing library and the mpi4python library. Ultimately, this wasn't too difficult after we implemented some small demos to understand how they work. Then, we implemented the game of Go based on the existing code. This was a non-trivial task, but it familiarized us with the code base we were using. While neither of these tasks were particularly difficult, they were somewhat time consuming. Next, we implemented leaf parallelized MCTS with the multiprocessing library, which was only difficult until we figured out how to use a built in class to give the abstraction of shared memory. The first difficult step of our project was to set up mpi4python correctly and then to develop our root parallel version. The documentation for this library is not particularly extensive, and getting the child processes running, scheduling the correct tasks, and sending the correct messages were distinct challenges. However, once we got this running, tweaking it to implement leaf parallelization with mpi4python and using mpi to implement our new parallel versions became much easier. The last and most difficult part of the project was progressively implementing and improving our two ideas of improved parallel versions. We spent large amounts of time thinking about how to combine the data in useful ways and implementing slight changes to see what made our players perform the best. Additionally, running experiments was not difficult but extremely time consuming as the games can take a long time to play, so testing our implementations as we went along to get preliminary results was incredibly useful and helped steer us in the correct direction.

Our implementation varied from our original proposal in a few key ways. First, we did not end up using GPU computation nor Tree parallelization. These changes in plan came about midway through the project when we realized that these approaches would not likely be feasibly for the setup we had. For the GPU version, we realized that the simulations that we would need to do with pyCUDA were highly game specific and would be difficult to accomplish in our given time frame. We believe that this approach might actually lead to significant simulation performance gains, and we would like to explore this approach in the future, either by focusing on a specific game to simulate or by using a GPU to apply other machine learning techniques to our MCTS implementations (e.g. by using a neural net to help us evaluate board states). Additionally, we did not pursue tree parallelization. We realized that tree parallelization requires shared memory between processes, which unfortunately, due to python's Global Interpreter Lock and lack of threading, is effectively unfeasible in a computationally efficient manner. As stated before, an abstraction of shared memory in this computationally intensive algorithm would also likely cause significant overhead and slow down the program immensely. We would like to port our existing solutions to C++ or C in the future and explore using some of our ideas from this project to improve current research on tree parallelization. While we left out some of our initial ideas from the final project, we were able to explore, through guidance from Professor Tia Newhall and Professor Bryce Wiedenbeck, some new ideas to improve the player performance of root parallel MCTS. While these approaches did not experimentally lead to any significant performance gain over root parallelization, we believe that they may still hold promise for future work. Moreover, implementing new approaches and incrementally tweaking them to improve the player performance led us to personally gaining a lot of knowledge and intuition about the nature of applying parallelization to Artificial Intelligence problems.

References

[1] Amine Bourki, Guillaume Chaslot, Matthieu Coulm, Vincent Danjean, Hassen Doghmen, Jean-Baptiste Hoock, Thomas Hérault, Arpad Rimmel, Fabien Teytaud, Olivier Teytaud, et al. Scalability and parallelization of monte-carlo tree search. In *International Conference on Computers and Games*, pages 48–58. Springer, 2010.

[2] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[3] Guillaume M. J. B. Chaslot, Mark H. M. Winands, and H. Jaap van den Herik. Parallel monte-carlo tree search. In H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H. M. Winands, editors, *Computers and Games*, pages 60–71, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[4] Kamil Rocki and Reiji Suda. Massively parallel monte carlo tree search. In *Proceedings of the 9th International Meeting High Performance Computing for Computational Science*, 2010.

[5] Kamil Rocki and Reiji Suda. Parallel monte carlo tree search on gpu. In *SCAI*, pages 80–89, 2011.