Exploring Parallel MCTS on Chess Game

曾正豪 0716325
CS NYCU
Hsinchu, Taiwan

王健業 0716098
CS NYCU
Hsinchu, Taiwan

張宸愷 0710018
EECSHP NYCU
Hsinchu, Taiwan

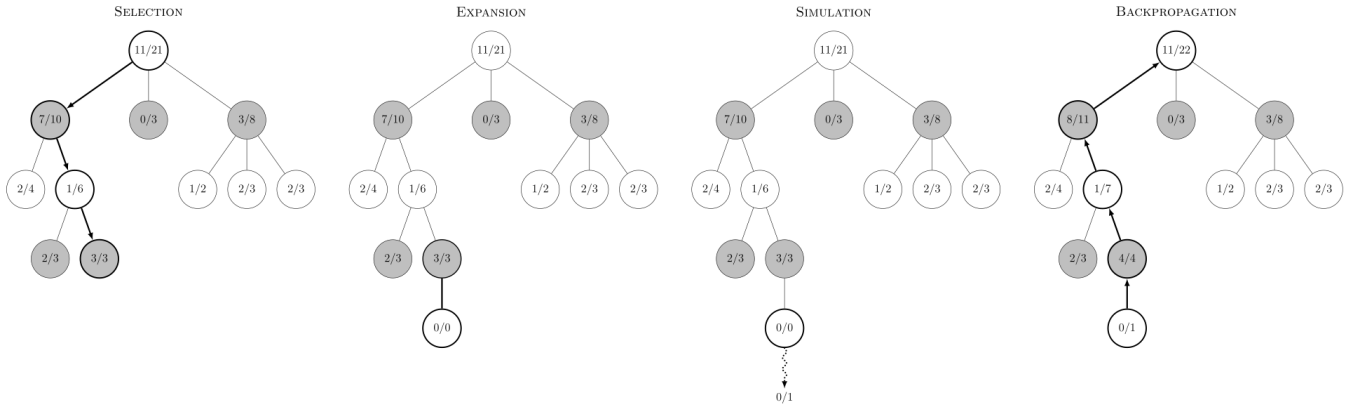

Figure 1: Illustration for a single step of MCTS

ABSTRACT

We decided to explore the parallelization of Monte Carlo Tree Search on Chess games using the techniques and knowledge we have learned in this course. We will use quantitative benchmarks to compare different parallelization approaches.

KEYWORDS

MCTS, parallel programming, Pthreads, CUDA, Chess

ACM Reference Format:
曾正豪 0716325, 王健業 0716098, and 張宸愷 0710018. 2021. Exploring Parallel MCTS on Chess Game. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

'Monte Carlo Tree Search', abbreviated in this text from now on as MCTS, is a probability sampling based tree search method for many applications. One of the most famous application of MCTS is AlphaGO [5]. It uses MCTS with 2 other neural networks to play Go. An early version of AlphaGo was tested on hardware with various numbers of CPUs and GPUs, running in asynchronous or distributed mode. It was tested with search threads from 12 to 64, number of CPUs from 48 to 1920, and number of GPUs from 1 to 280. And in 2016, it changed to use TPUs (tensor processing units) as its computing unit. In recent years, it keeps beating many go players. Overall, MCTS is an algorithm that can be highly parallelized because of the high number of simulations. Hence, we decided to use MCTS as the topic of our final project.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

2 STATEMENT OF PROBLEM

One of our teammates had taken the course AI capstone, and during that course he had been doing a final project about playing a 3D version of connect-4 using upper confidence bound MCTS, i.e. UCB-MCTS. He noticed that when he uses root parallelization, the performance of the MCTS decreases dramatically. The multithreaded version with 8 threads didn't even manage to beat the single-threaded one. The teammate's guess is that it's because of false sharing. Thus, we want to explore further on ways to improve multithreaded performance, and hopefully use this new improved MCTS to play on well-known games such as Chess, instead of some obscure game that our teammate played back then.

3 PROPOSED SOLUTION

3.1 Language selection

We will use C++ as our main programming language, together with Pthread and CUDA, since we want to compare the performance of CPU and GPU on parallel programs.

3.2 Overall design

Figure 2 is our system design. Component descriptions:

- Chess game module: Deal with the game state of Chess, once it receives a legal move, it will change its game state and return it.

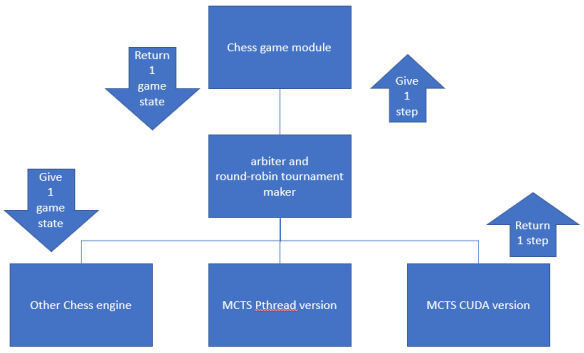

Figure 2: Our system diagram

- Arbiter and round-robin tournament maker: It will coordinate the game player and the game module. It checks if one move is legal and whether the game ends.
- 3 MCTS versions: To calculate the best move next. It will receive a game state and return its best move. The difference between these 3 versions is their parallel approach.
- Additional: Maybe we will compete with other Chess engines to see the performance pitted against other state of the art, if we have enough time.

The performance of each method will depend on the games won and the total amount of expanded nodes.

3.3 parallelization methods

There are four ways to parallelize traditional UCB-MCTS mentioned in [1], and illustrated in figure 3. Leaf parallelization, root parallelization, tree parallelization with global mutex, and tree parallelization with local mutexes.

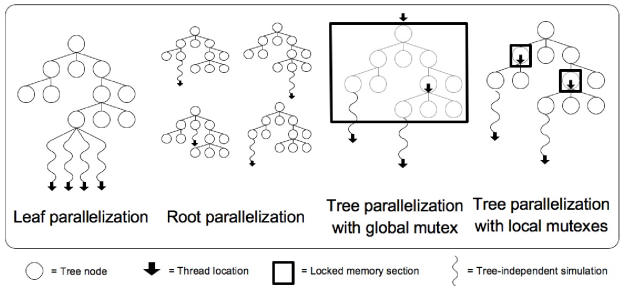

Figure 3: Ways of parallelizing MCTS

We will use leaf parallelization, root parallelization, and tree parallelization with global mutex for our project. We will explain them in the following sections.

3.4 root parallelization

In root parallelization, we use Pthread as our method of achieving parallelization. In each thread an independent MCTS tree is constructed. When the opponent plays a move, the master thread will update the shared memory of all the worker threads. When done updating, the worker threads waiting on the condition variable will receive a broadcast from the master thread and unfreeze to do independent MCTS grow algorithms. After time is up, the master thread will wait on all the worker threads, and collect the results from the threads to give the best move. The entire control flow is illustrated in 4.

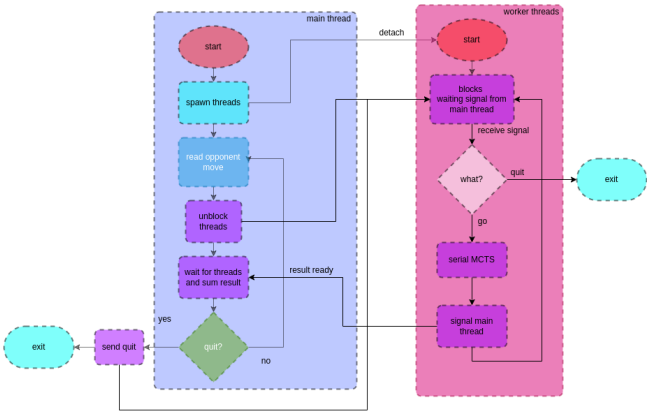

Figure 4: The flow of root parallelization

3.5 tree parallelization

In tree parallelization, we use global mutex instead of local mutex for mutex location in reaching tree parallelization. Global mutex method locks the whole tree in such a way that only one thread can access the search tree at a time for 3 phases which are Selection, Expansion and Backpropagation. In the meantime, several other processes can still make simulation. The Pseudocode for tree parallelization are shown in below.

Algorithm 1 pseudocode for tree parallelization （Global Mutex)

```
Function MonteCarloTreeSearch(state0)
create root node node0 with state state0
create N threads to run below code asynchronously:
while time ≤ maximumallowedtime do
    Lock()
    chosenNode = TreePolicy(node0)
    UnLock()
    result = SimulationPolicy(chosenNode)
    Lock()
    BackPropagation(result , chosenNode)
    UnLock()
end while
return best scoring node
```

3.6 leaf parallelization

In leaf parallelization, we use CUDA to accelerate the computation. Since CUDA is SIMT architecture, we need to reduce the number of branch as many as possible. Hence, we use evaluation function (figure 6) to evaluate the game state of leaf nodes rather than play

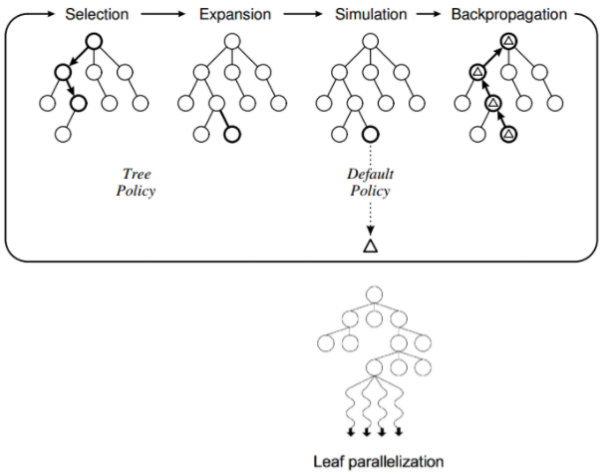

Figure 5: The flow of leaf parallelization

the game until finish. In this section, we use the evauation function from chess programming wiki.

function CUDASimulationKernel(*state_list*)

resultArray[threadIdx] = evaluation-function(*state_list* [threadIdx])

```
f(p) = 200(K-K')
       + 9(Q-Q')
       + 5(R-R')
       + 3(B-B' + N-N')
       + 1(P-P')
       - 0.5(D-D' + S-S' + I-I')
       + 0.1(M-M') + ...

KQRBNP = number of kings, queens, rooks, bishops, knights and pawns
D,S,I = doubled, blocked and isolated pawns
M = Mobility (the number of legal moves)
```

Figure 6: The evauation function we used in leaf parallelization

4 EXPERIMENTAL METHODOLOGY

We used 3 different ways in the experiment. The first one is that we will test how long it will run in the four stages of MCTS between these 3 parallelization methods. The second one is that we will test how many nodes it will expand on the Monte Carlo Tree between these 3 parallelization methods. The final experiment is that we will let each engine to compete with each other and we will also add the world record chess engine, stockfish, in the competition. Our experiment platform is consist of an AMD R5-3600 CPU with DDR4 8GB*2@3600Mhz memory. In the leaf parallelization, we use an RTX3080 GPU. In the experiment of the first 2 parts, our standard is that we will let our engines the calculate the first move at the starting position for exactly 5 seconds.

5 EXPERIMENTAL RESULTS

Figure 7 shows the time consumption of each stage in different parallelization methods. Form the figure, you can see that the simulation stage take the most time on all the parallelization methods. Even we use evauation function to cooperate the CUDA architecture, it still takes the most time among these stages. We found that the number of expanded nodes of leaf parallelization is close to root parallelization runs in 8 threads, but it only use 1 CPU thread to run each stage. Hence, the percentage of simulation of leaf parallelization will be less.

Benchmarks-compare the time of each state

	selection	expansion	simulation	backpropogation
single thread	2.9665ms	10.5583ms	4980.28ms	0.5641ms
root parallelization	3.1230ms	10.5413ms	4981.12ms	0.6213ms
leaf parallelization	25.144ms	189.566ms	4762.59ms	3.2255ms
tree parallelization	4.3435ms	20.6726ms	4964.43ms	4.6199ms

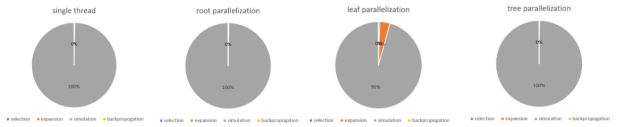

Figure 7: The result of experiment 1

Figure 8 shows the number of expanded nodes in different parallelization methods runs in different number of threads. As you can see, the root parallelization has scalibility, its speedup is close to its running threads. Since the root parallelization has little dependencies, only in the last step we need to vote the best move needs to do the reduction. As for tree parallelization, its speedup is not close to its number of threads. This is because we use a single mutex for the whole tree. It may improve if we use mutex for each nodes. In the part of leaf parallelization, we found that the speed up of GPU acceleration is very powerful. Even we use a single CPU thread, the number of expanded nodes is close to the other 2 methods runs in 4 threads and its simulation leaf nodes is 10 times higher than other parallelization methods.

Benchmarks-compare the number of expanded nodes

	1 thread	2 threads	4 threads	8 threads
single thread	4488	X	X	X
root parallelization	4432	8853	17950	35985
leaf parallelization	17088 expanded nodes / 387770 simulated leaf nodes			
tree parallelization	4394	9052	17787	29083

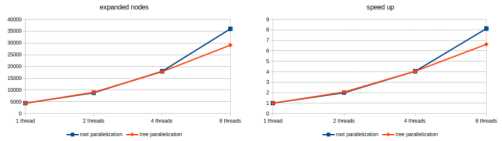

Figure 8: The result of experiment 2

Figure 9 shows the results of the competition of these engines. From the figure, we can see that our engine is nothing when competing stockfish. Can't win any battle. As for the leaf parallelizaiton,

it simulate the most leaf nodes, but its win rate is low. We think it is because we use a evaluation function that is too simple. Its ability of evaluating a chess board may not be so good. Overall, in these parallelizaiton methods, the root parallelization has the best performance, its win ration is 80% among the other methods. The second is tree parallelization. The worst is leaf parallelization.

Benchmarks-compete with each version for 100 games

(white win/black win)

white (First move) \ black	stockfish	MCTS-single thread	MCTS-root parallelization	MCTS-tree parallelization	MCTS-leaf parallelization
stockfish		100/0	100/0	100/0	100/0
MCTS-single thread	0/100		19/81	38/62	65/35
MCTS-root parallelization	0/100	84.5/15.5		82/18	78/22
MCTS-tree parallelization	0/100	56/44	11/89		75.5/24.5
MCTS-leaf parallelization	0/100	31/69	35/65	22/78	

Figure 9: The result of experiment 3

6 RELATED WORK

[4] is a introductory view on UCB-MCTS with additional enhancements e.g. rapid action value estimate (RAVE), all moves as first (AMAF), etc. A pretty detailed work [3] regarding current developments on different MCTS, including some exotic methods that uses virtual loss [2] (i.e. do simulation before the previous backprop result is known). Different methods for enhancing the capability of MCTS on board games were explored in [6]. [1][6] also outlined some of the parallelization techniques for MCTS, root parallelization, leaf parallelization, and tree parallelization, etc. [6] also outlined a way for a modularized MCTS algorithm that can scale up as the number of computer nodes grow. [1] is most similar to our work. Their root parallelization results are nearly identical to ours. However, their tree parallelization with global mutex is worse than ours.

7 CONCLUSION

There are 3 kinds of parallel MCTS, root, tree, and leaf. In our experiments, we showed the effect of parallelization methods. And we found that MCTS can be highly parallelized. Overall, in our implementation, we show the high scalability in our aproaches.

8 APPENDICES

REFERENCES

[1] Guillaume M.J-B. Chaslot, Mark H.M. Winands, and H. Jaap van den Herik. 2008. Parallel Monte-Carlo Tree Search.

[2] Anji Liu, Jianshu Chen, Mingze Yu, Yu Zhai, Xuewen Zhou, and Ji Liu. 2020. Watch the Unobserved: A Simple Approach to Parallelizing Monte Carlo Tree Search. In *International Conference on Learning Representations*. https://openreview.net/forum?id=BJlQtJSKDB

[3] Anji Liu, Yitao Liang, Ji Liu, Guy Van den Broeck, and Jianshu Chen. 2020. On Effective Parallelization of Monte Carlo Tree Search. arXiv:2006.08785 [cs.LG]

[4] Magnuson and Max. 2015. Monte Carlo Tree Search and Its Applications. *University of Minnesota, Morris Undergraduate Journal* 2, 2 (2015).

[5] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529 (01 2016), 484–489. https://doi.org/10.1038/nature16961

[6] Martin Weigel. 2017. Monte Carlo methods for massively parallel computers. arXiv:1709.04394 [physics.comp-ph]

A ONLINE RESOURCES

(1) Monte Carlo tree search (https://en.wikipedia.org/wiki/Monte_Carlo_tree_search)