Universiteit Leiden
The Netherlands

Structured parallel programming for Monte Carlo Tree Search

Mirsoleimani, S.A.

Citation

Mirsoleimani, S. A. (2020, June 17). *Structured parallel programming for Monte Carlo Tree Search*. *SIKS Dissertation Series*. Retrieved from https://hdl.handle.net/1887/119358

Version:	Publisher's Version
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/119358

Note: To cite this publication please use the final published version (if applicable).

Cover Page

Universiteit Leiden

Leiden University Repository

The handle http://hdl.handle.net/1887/119358 holds various files of this Leiden University dissertation.

Author: Mirsoleimani, S.A.
Title: Structured parallel programming for Monte Carlo tree search
Issue Date: 2020-06-17

7 Ensemble UCT Needs High Exploitation

The last research question is *RQ5* (see our research design in Section 1.7).

- ***RQ5***: *To what extent do the existing solutions which improve search quality, apply to our version of parallelized MCTS?*

This research question is derived from the fact that the quality of search in MCTS depends on the balance between exploitation (look in areas which appear to be promising) and exploration (look in areas that have not been well sampled yet). The most popular algorithm in the MCTS family which addresses this dilemma is UCT [KS06] (see Section 2.2). Parallelization of MCTS intends to decrease the execution time of the algorithm, but it also affects the exploitation-exploration balance. A set of solutions has been developed to control the exploitation-exploration balance when parallelizing MCTS to improve the quality of search [BPW$^+$12, KPVvdH13]. We partition the set of solutions into two parts, (1) adjusting the exploitation-exploration balance with respect to the tree size, and (2) adjusting the exploitation-exploration balance by an artificial increase in exploration, called *virtual loss*. We provide an answer for *RQ5* with respect to these two parts. This chapter [1] investigates the application of the first solution to Root Parallelization. In Chapter 8, we analyze the use of the second solution on the lock-free Tree Parallelization algorithm.

[1] Based on:

- S. A. Mirsoleimani, A. Plaat, and H. J. van den Herik, and J. Vermaseren, Ensemble UCT Needs High Exploitation, in Proceedings of the 8th International Conference on Agents and Artificial Intelligence, 2016, pp. 370--376.

Hence, we start investigating the adjustment of the exploitation-exploration balance with respect to the tree size. As with most sampling algorithms, one way to improve the quality of the result is to increase the number of samples and thus enlarge the size of the MCTS tree. However, constructing a single large search tree with t samples or playouts is a time-consuming process (see Subsection 2.3.2). A solution for this problem is to create a group of n smaller trees that each has t/n playouts and search these in parallel. This approach is used in Root Parallelization [CWvdH08a] and in Ensemble UCT [FL11] (from now on we use these two names interchangeably). In both Root Parallelization and Ensemble UCT, multiple independent UCT instances are constructed. At the end of the search process, the statistics of all trees are combined to yield the final result [BPW^{+}12]. However, there is contradictory evidence on the success of Ensemble UCT [BPW^{+}12]. On the one hand, Chaslot et al. found that, for Go, Ensemble UCT (with n trees of t/n playouts each) outperforms a plain UCT (with t playouts) [CWvdH08a]. On the other hand, Fern and Lewis were not able to reproduce this result in other domains [FL11]. They found situations where a plain UCT outperformed Ensemble UCT given the same total number of playouts. We aim to shed light on this controversy using an idea from [KPVvdH13]. Kuipers et al. argued that when the tree size in MCTS is small, more exploitation should be chosen, and with larger tree sizes, high exploration is suitable [KPVvdH13]. Therefore, the main contribution of this chapter is to show that this idea can be used in Ensemble UCT to improve its search quality by adjusting the C_p parameter depending on the ensemble size.

The remainder of the chapter is organized as follows. Section 7.1 describes Ensemble UCT. Section 7.2 discusses related work. Section 7.3 gives the experimental setup, Section 7.4 describes the experimental design, and Section 7.5 provides the experimental results for this study.

7.1 Ensemble UCT

Ensemble UCT or the Root Parallelization algorithm belongs to the category of parallel algorithms with more than one data structure (see Subsection 2.3.2). It creates an ensemble of search trees (i.e., one for each thread). The trees are independent of each other. When the search is over, they are merged, and the action of the best child of the root is selected to be performed.

Ensemble UCT is given its place in the overview article by [BPW^{+}12]. Table 7.1 shows different possible configurations for Ensemble UCT. Each configuration has its benefits. The total number of playouts is t, and the size of the ensemble (number of trees inside the ensemble) is n. It is assumed that n processors are available with n

Table 7.1: Different possible configurations for Ensemble UCT. Ensemble size is n.

Number of playouts			playout speedup		playing strength
UCT	Ensemble UCT		n cores	1 core	
	Each tree	Total			
t	t	$n \cdot t$	1	$\frac{1}{n}$	Yes, known
t	$\frac{t}{n}$	t	n	1	?

equal to the ensemble size.

The third line of Table 7.1 shows the situation where Ensemble UCT has $n \cdot t$ playouts in total, while plain UCT has only t playouts. In this case, there would be no speedup in a parallel execution of the ensemble approach on n cores, but the larger search effort would presumably result in a better search result. We call this use of parallelism *playing strength* (see Subsection 2.5.2). The fourth line of Table 7.1 shows a different possible configuration for Ensemble UCT. In this case, the total number of playouts for both UCT and Ensemble UCT is equal to t. Thus, each core searches a smaller tree of size t/n. The search will be n times faster (the ideal case). We call this use of parallelism *Playout speedup* (see Subsection 2.5.1). It is important to note that in this configuration both approaches take the same amount of time on a single core. However, there is still the question whether we can reach any *playing strength*. This question will be answered in Section 7.5 as the first part of RQ5.

7.2 Related Work

From the introduction of this chapter we know that [CWvdH08a] provided evidence that, for Go, Root Parallelization with n instances of t/n iterations each outperforms plain UCT with t iterations, i.e., Root Parallelization (being a form of Ensemble UCT) outperforms plain UCT given the same total number of iterations. However, in other domains, [FL11] did not find this result. [SKW10] also analyzed the performance of root parallelization in detail. They found that a majority voting scheme gives better performance than the conventional approach of playing the move with the greatest total number of visits across all trees. They suggested that the findings in [CWvdH08a] are explained by the fact that Root Parallelization performs a shallower search, making it easier for UCT to escape from local optima than the deeper search performed by plain UCT (see also Section 8.2, in relation with part two).

In Root Parallelization each process does not build a search tree larger than the sequential UCT. Moreover, each process has a local tree, which contains characteristics that differ from tree to tree. Rather recently, [TD15] proposed a new idea by distinguishing between tactical behavior and strategic behavior. They transferred the RAVE (Rapid Action Value Estimate) ideas as developed by [GS07], from the selection phase to the simulation phase. This implies that influencing the tree policy is changed into also influencing the Monte-Carlo policy.

Fern and Lewis thoroughly investigated an Ensemble UCT approach in which multiple instances of UCT were run independently. Their root statistics were combined to yield the final result [FL11]. So, our task is to explain the differences in their work and that by [CWvdH08a].

7.3 Experimental Setup

Section 7.3.1 discusses our case study and Section 7.3.2 provides the details of hardware.

7.3.1 The Game of Hex

The game of Hex is described in Subsection 2.4.1. Below follows complementary information needed for this chapter. The 11×11 Hex board is represented by a disjoint-set. This data structure has three operations *MakeSet*, *Find* and *Union*. In the best case, the amortized time per operation is $O\left(\alpha\left(n\right)\right)$, where $\alpha\left(n\right)$ denotes the inverse Ackermann function. The value of $\alpha\left(n\right)$ is less than 5 for all remotely practical values of n [GI91].

In Ensemble UCT, each tree performs a completely independent UCT search with a different random seed. To determine the next move to play, the number of wins and visits of the root's children of all trees are collected. For each child the total sum of wins and the total sum of visits are computed. The child with the largest number of wins/visits is selected.

The plain UCT algorithm and Ensemble UCT are implemented in the ParalellUCT package. In order to make our experiments as realistic as possible, we use the ParallelUCT program for the game of Hex [MPVvdH14, MPvdHV15a]. This program is highly optimized, and reaches a speed of more than 40,000 playouts per second per core on a 2,4 GHz Intel Xeon processor (see Section 2.6).

7.3.2 Hardware

The results were measured on a dual socket machine with 2 Intel Xeon E5-2596v2 processors running at 2.40GHz. Each processor has 12 cores, 24 hyperthreads and 30 MB L3 cache. Each physical core has 256KB L2 cache. The pack TurboBoost frequency

Table 7.2: The performance of Ensemble UCT vs. plain UCT based on win rate.

Approach	Win (%)	Performance vs. plain UCT	Playing Strength
Ensemble UCT	< 50	Worse than	No
	$= 50$	As good as	No
	> 50	Better than	Yes

is 3.2 GHz. The machine has 192GB physical memory. Intel's *icc 14.0.1* compiler is used to compile the program.

7.4 Experimental Design

As Hex is a 2-player game, the playing strength of Ensemble UCT is measured by playing versus a plain UCT with the same number of playouts. We *expect* to see an improvement for the Ensemble UCT playing strength against plain UCT by choosing 0.1 as the value of C_p (high exploitation) when the number of playouts is small. We start our experiments by setting the value of C_p to 1.0 for plain UCT (high exploration). Note that for the purpose of this research, it is not essential to find the optimal value of C_p, but to show the difference in effect on the performance when C_p is varying.

The board size for Hex is 11×11. In our experiments, the maximum ensemble size is $2^8 = 256$. Thus, for 2^{17} playouts, when the ensemble size is 1, there are 2^{17} playouts per tree and when the ensemble size is $2^6 = 64$ the number of playouts per tree is 2^{11}. Throughout the experiments, the ensemble size is multiplied by a factor of two.

7.5 Experimental Results

Our experimental results show the percentage of wins for Ensemble UCT with a particular ensemble size and a particular C_p value. In Figure 7.1 results are shown, with C_p =0 (only exploitation) and ensemble size equals 8. Each data point represents the average of 200 games with a corresponding 99% confidence interval. Table 7.2 summarizes how the performance of Ensemble UCT versus plain UCT is evaluated. The concept of *high exploitation for small UCT tree* is significant if Ensemble UCT reaches a win rate of more than 50%. (Section 7.5 will show that this is indeed the case.)

Below we provide our experimental results. We distinguish them into (A) hidden exploration in Ensemble UCT and (B) exploitation-exploration trade-off for Ensemble UCT.

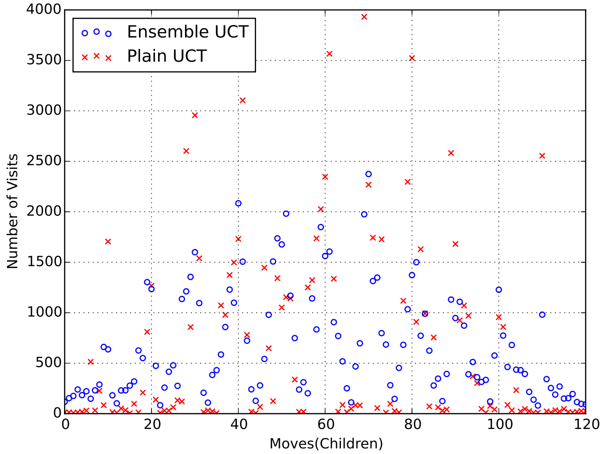
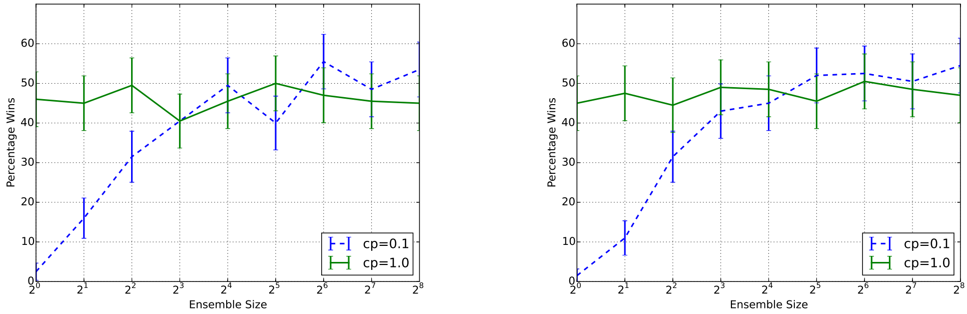


Figure 7.1: The number of visits for root's children in Ensemble UCT and plain UCT. Each child represents an available move on the empty Hex board with size 11×11. Both Ensemble UCT and plain UCT have 80,000 playouts and $C_p = 0$. In Ensemble UCT, the size of the ensemble is 8.

A: Hidden Exploration in Ensemble UCT

It is important to understand that Ensemble UCT has a hidden exploration factor by nature. Two reasons are: (1) each tree in Ensemble UCT is independent, and (2) an ensemble of trees contains more exploration than a single UCT search with the same number of playouts would have. The hidden exploration is because each tree in Ensemble UCT searches in different areas of the search space.

In Figure 7.1 the difference in exploitation-exploration behavior of the Ensemble UCT and plain UCT is shown in the number of visits that one of the root's children counts when using one of the algorithmic approaches with $C_p = 0$. Both Ensemble UCT [BPW+12] and plain UCT [BPW+12] have 80,000 of playouts. In each experiment, a search tree for selecting the first move on an empty board is constructed. Each of the children corresponds to a possible move of an empty Hex board (i.e., 121 moves). Ensemble UCT is more explorative compared to plain UCT if it generates more data points with more distance from the x-axis than plain UCT. In Ensemble UCT the number of playouts is distributed among 8 separate smaller trees. Each of the trees has 10,000 playouts and for each child the number of visits is collected. When the value of C_p is 0, which means the exploration part of the UCT formula is turned off, all possible moves in the Ensemble UCT receive at least a few visits. While for plain UCT with 80,000 playouts and $C_p = 0$ there are many of the moves with



(a) The total number of playouts is $2^{17} = 131072$

(b) The total number of playouts is $2^{18} = 262144$

Figure 7.2: The percentage of wins for ensemble UCT is reported. The value of C_p for plain UCT is always 1.0 when playing against Ensemble UCT. To the left few large UCT trees, to the right many small UCT trees.

no visits. The data points when using plain UCT are closer to the x-axis compared to Ensemble UCT. However, for Ensemble UCT the peak is 2400, while it is 4000 visits for plain UCT. It means that plain UCT is more exploitative.

B: Exploitation-Exploration trade-off for Ensemble UCT

Below we discuss two experiments: (B1) an experiment with 2^{17} playouts and (B2) an experiment with 2^{18} playouts. In Figures 7.2a and 7.2b, from the left side to the right side of the graph, the ensemble size (the number of search trees per ensemble) increases by a factor of two, and the number of playouts per tree (tree size) decreases by the same factor. Thus, at the right-hand side of the graph, we have the largest ensemble with the smallest trees. The total number of playouts always remains the same throughout an experiment for both Ensemble UCT and plain UCT. The value of C_p for plain UCT is always 1.0, which means high exploration.

B1: Experiment with 2^{17} playouts

Figure 7.2a shows the *relations* between the value of C_p and the ensemble size, when both plain UCT and Ensemble UCT have the same number of total playouts. Moreover, Figure 7.2a shows the *performance* of Ensemble UCT for different values of C_p. It shows that when $C_p = 1.0$ (highly explorative) Ensemble UCT performs as good as (or mostly worse than) plain UCT. When Ensemble UCT uses $C_p = 0.1$ (highly exploitative) then for small ensemble sizes (large sub-trees) the performance of Ensemble UCT sharply drops down. By increasing the ensemble size (smaller sub-trees), the performance of Ensemble UCT keeps improving until it becomes as good as or even better than plain UCT.

B2: Experiment with 2^{18} playouts

A second experiment is conducted using 2^{18} playouts to investigate the effect of enlarging the number of playouts on the performance of Ensemble UCT. Figure 7.2b shows that when for this large number of playouts the value of $C_p = 1.0$ is high (i.e., highly explorative) the performance of Ensemble UCT cannot be better than plain UCT, while for a small value of $C_p = 0.1$ (i.e., highly exploitative) the performance of Ensemble UCT is almost always better than plain UCT when the ensemble size is 2^5 or larger. Hence, our conclusion is that there exists a marginal playing strength. The potential playout speedup could be up to the ensemble size if a sufficient number of processing cores is available.

7.6 Answer to the First Part of RQ5

This chapter aims at answering the first part of *RQ5* (i.e., adjusting the exploitation-exploration balance with respect to the tree size) of the following question.

- ***RQ5***: *To what extent do the existing solutions which improve search quality, apply to our version of parallelized MCTS?*

The chapter described an empirical study on Ensemble UCT with different sets of configurations for the ensemble size, the tree size, and the exploitation-exploration trade-off. Previous studies on Ensemble UCT/Root Parallelization provided inconclusive evidence on the effectiveness of Ensemble UCT (see the beginning of the chapter).

Our results suggest that the reason for uncertainty (concerning the controversy in the previous studies) lies in the exploitation-exploration trade-off in relation to the size of the sub-trees. With this knowledge, it is explainable that [CWvdH08a] found an improvement in their Root Parallelization for Go (which has big search trees for small ensemble sizes where exploration can open new perspectives). For [FL11], it is also explainable that they did not arrive at the same success in other domains (which have small search trees for large ensemble sizes). Our experiments for Ensemble UCT now confirm earlier ideas as provided by [KPVvdH13] on this topic. In summary, our results provide clear evidence that the performance of Ensemble UCT is improved by selecting higher exploitation for smaller search trees given a fixed time-bound or fixed number of simulations.

Our work is particularly motivated, in part, by the observation in [CWvdH08a] of super-linear speedup in Root Parallelization. Finding super-linear speedup in two-agent games occurs infrequently. Most studies in parallel game-tree search report a battle against search overhead, communication overhead (e.g., [Rom01]), synchronization overhead, and deployment overhead (see, Chapter 1). For super-linear speedup to occur, the parallel search must search fewer nodes than the sequential search.