Structured parallel programming for Monte Carlo Tree Search

Mirsoleimani, S.A.

8

An Analysis of Virtual Loss in Parallel MCTS

We reiterate the last research question, *RQ5*, and continue the research work started in Chapter 7.

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

In part one of *RQ5* (see Chapter 7) we investigated to what extent the successes of MCTS depend on the balance between exploitation and exploration (see also Section 2.2). The parallelization of MCTS intends to decrease the execution time of the algorithm, but it also affects this trade-off. Therefore, solutions are developed to control the exploitation-exploration balance when parallelizing MCTS to improve the quality of search [CWvdH08a, BPW+12, KPVvdH13]. We have partitioned the set of solutions into two parts, (1) adjusting 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 provided an answer for *RQ5* (part one) in Chapter 7. This chapter[1] addresses the second part of *RQ5*.

Each iteration of the MCTS algorithm adds a new node to a tree by first selecting a path inside the tree and then using Monte Carlo simulations. This iterative process is path-dependent, which means that the outcomes of previous iterations guide the

[1] Based on:

- S. A. Mirsoleimani, A. Plaat, and H. J. van den Herik, and J. Vermaseren, An Analysis of Virtual Loss in Parallel MCTS, in Proceedings of the 9th International Conference on Agents and Artificial Intelligence, 2017, pp 648--652.

future selections. Rather recently, several studies have addressed the topic of making parallel methods for MCTS, such as Tree Parallelization and Root Parallelization [BG11, BPW+12, SHM+16, SSS+17]. Here we focus on Tree Parallelization that distributes different iterations of MCTS among parallel workers. Therefore, it has to violate the path dependency feature of sequential MCTS to make the algorithm faster.

In Tree Parallelization, the performance is decreasing when increasing the number of parallel workers. It is widely believed that part of the performance loss is due to a redundant search being done by separate parallel workers (i.e., Search Overhead). However, if the parallel algorithm is using a lock to guarantee synchronization, the contention among parallel workers also contributes to the performance loss. Therefore, a method called *virtual loss* is proposed for lock-based Tree Parallelization [CWvdH08a]. It forces parallel workers to traverse different paths inside the MCTS tree to avoid contention around a particular node. However, virtual loss then affects the balance between exploitation and exploration in the UCT algorithm by increasing the exploration level irrespective of the value of the C_p parameter.

In this chapter, we evaluate the benefit of using the virtual loss (i.e., an artificial increase in exploration against exploitation) for lock-free (instead of locked-based) Tree Parallelization. We carry out our experiments for a full range of exploitation-exploration in UCT (i.e., the C_p parameter) and a varying number of parallel workers. The result is reported concerning Search Overhead (SO) and Time Efficiency (Eff). The case studies are problems from the High Energy Physics domain.

The remainder of the chapter is organized as follows. The virtual loss method is explained in Section 8.1, the related work is presented in Section 8.2, the experimental setup is described in Section 8.3, followed by the experimental design in Section 8.4, and the experimental results in Section 8.5. Finally, the answer to the second part of *RQ5* is given in Section 8.6, with the complete answer to *RQ5* in Section 8.7.

8.1 Virtual Loss

In Tree Parallelization one MCTS tree is shared among several threads that are performing simultaneous searches [CWvdH08a]. The main challenge in this method is using data locks to prevent data corruption. A lock-free implementation of this algorithm addresses the problem as mentioned earlier with better scaling than a locked approach [EM10]. Therefore, in our implementation of Tree Parallelization, locks are removed.

Definition 8.1 (Virtual Loss) *Virtual loss is a method to make a node in the tree less favorable to be selected and therefore force parallel workers to traverse different paths inside the MCTS tree.*

Here we note that in Tree Parallelization with fine-grained locks (see Subsection 5.2.1), it is still possible that different threads traverse the tree in mostly the same way. This phenomenon causes thread contention when two different threads visit the same node concurrently, and one thread is waiting for a lock that is currently being held by another thread. Increasing the number of threads exacerbates this problem. [CWvdH08a] suggested a solution to assign a temporary *virtual loss* (a marker) to a node when a thread selects it. Without the marker, there is a higher chance for thread contention.

Implementing the virtual loss is straightforward. A thread is selecting a path inside the tree to find a leaf node. It is reducing the UCT value of all the nodes that belong to the path, assuming that the playout from the leaf node results in a loss. Therefore, the virtual loss will inspire other threads to traverse different paths and avoid contention. A thread removes the assigned virtual loss immediately before the backup step when updating the nodes with the real playout result. It is worth mentioning that Tree Parallelization with virtual loss is more explorative compared to plain Tree Parallelization because the virtual loss encourages different threads to explore different parts of the tree regardless of the value of C_p. Regarding the virtual loss, $UCT(j)$ decreases as more threads select node j, which encourages other threads to favor other nodes. Algorithm 8.1 gives the pseudocode for the virtual loss technique.

Algorithm 8.1: The lock-free UCT algorithm with virtual loss.

```
1  Function UCTSEARCH(Node* v_0, State s_0, budget)
2      while within search budget do
3          ⟨v_l, s_l⟩ := SELECT(v_0, s_0);
4          ⟨v_l, s_l⟩ := EXPAND(v_l, s_l);
5          Δ := PLAYOUT(v_l, s_l);
6          BACKUP(v_l,Δ);
7  Function SELECT(Node* v, s) : <Node*,State>
8      while v.ISFULLYEXPANDED() do
9          ⟨w, n⟩ := v.GET();
10         v_l := argmax_{v_j ∈ children of v} v_j.UCT(n);
11         s := v.p takes action v_l.a from state s;
12         v_l.SET(+LOSS(v_l.p));
13         v := v_l;
14     return ⟨v, s⟩;
15 Function BACKUP(Node* v,Δ) : void
16     while v is not null do
17         v.SET(−LOSS(v.p));
18         v.SET(Δ⟨v.p⟩);
19         v := v.parent;
```

8.2 Related Work

[CWvdH08a] reported that Tree Parallelization with local locks and virtual loss performs as well as Root Parallelization in the game of Go. However, [SCP+14] suggested that adding a virtual loss to Tree Parallelization with local locks has almost no effect on the performance for the game of Lords of War. [Seg11] showed that MCTS could scale nearly perfectly to at least 64 threads when combined with virtual loss, but without virtual loss scaling is limited to just eight threads. [EMAS10] showed that the virtual loss technique is very effective for Go in FUEGO. However, [BG11] found that increasing exploration by multiple virtual losses *slightly* improves Tree Parallelization with lock-free updates for Go in Pachi.

8.3 Experimental Setup

We perform a sensitivity analysis of C_p on the number of iterations for different thread configurations for one expression, namely HEP(σ) which is a polynomial from the HEP domain with 15 variables [Ver13, KPVvdH13, RVPvdH14]. The plain UCT algorithm and parallel methods are implemented in the ParallelUCT package.

The results are measured on a dual socket machine with 2 Intel Xeon E5-2596v2 processors running at 2.40GHz. Each processor has 12 cores, 24 hyper-threads and 30 MB L3 cache. Each physical core has 256KB L2 cache. The pack TurboBoost frequency is 3.2 GHz. The machine has 192GB physical memory. Intel's *icc 14.0.1* compiler is used to compile the program.

8.4 Experimental Design

In our case study, we investigate Horner schemes. We consider a Horner Scheme as an optimization problem (see Subsection 2.4.2). The playing strength of Tree Parallelization for the Horner scheme is measured by the number of operations that are found for a number of playouts (see Subsection 2.5.2). Here, we define search overhead ((SO)) and time efficiency (Eff) based on the number of playouts.

$$SO = \frac{\mathit{number\ of\ playouts}_{parallel}}{\mathit{number\ of\ playouts}_{sequential}} - 1. \tag{8.1}$$

$$\mathit{Eff} = \frac{\mathit{time}_{sequential}}{\mathit{number\ of\ parallel\ workers} \cdot\ \mathit{time}_{parallel}}. \tag{8.2}$$

In our experiments, the algorithm stops when it found 4,150 operations, or the limit of 10,240 playouts is reached. The numbers are set at 4,150 and 10,240 as

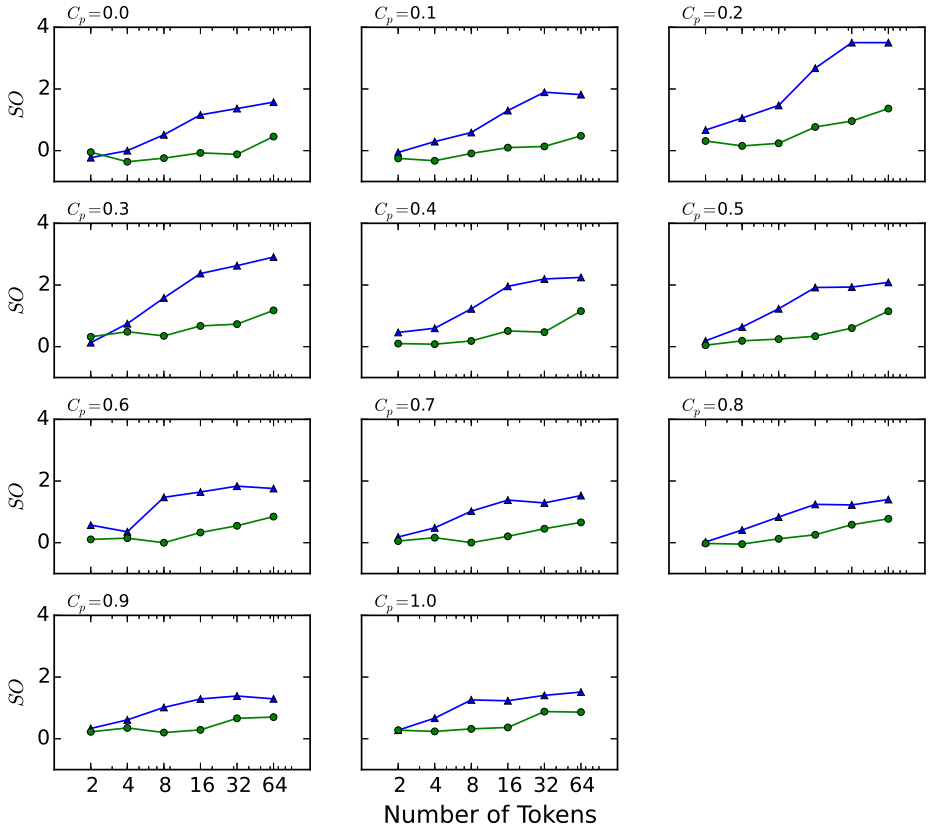

Figure 8.1: Search overhead (SO) for Horner (average of 20 instances for each data point). Tree parallelization is the green line which is indicated by circles, and Tree Parallelization with virtual loss is the blue line which is indicated by triangles. Note that the higher SO of Tree Parallelization with virtual loss means lower performance.

"relaxed" upper bound above 4,000 and 10,000 which are found by [KPVvdH13] for the HEP(σ) polynomial. Throughout the experiments, the number of tokens or tasks is multiplied by a factor of two. Each data point represents the average of 20 runs.

8.5 Experimental Results

Below we provide our experimental results. The first factor is Search Overhead (SO). We hope for the reduction of SO by using virtual loss. Figure 8.1 shows the SO of plain Tree Parallelization and Tree Parallelization with virtual loss for different values

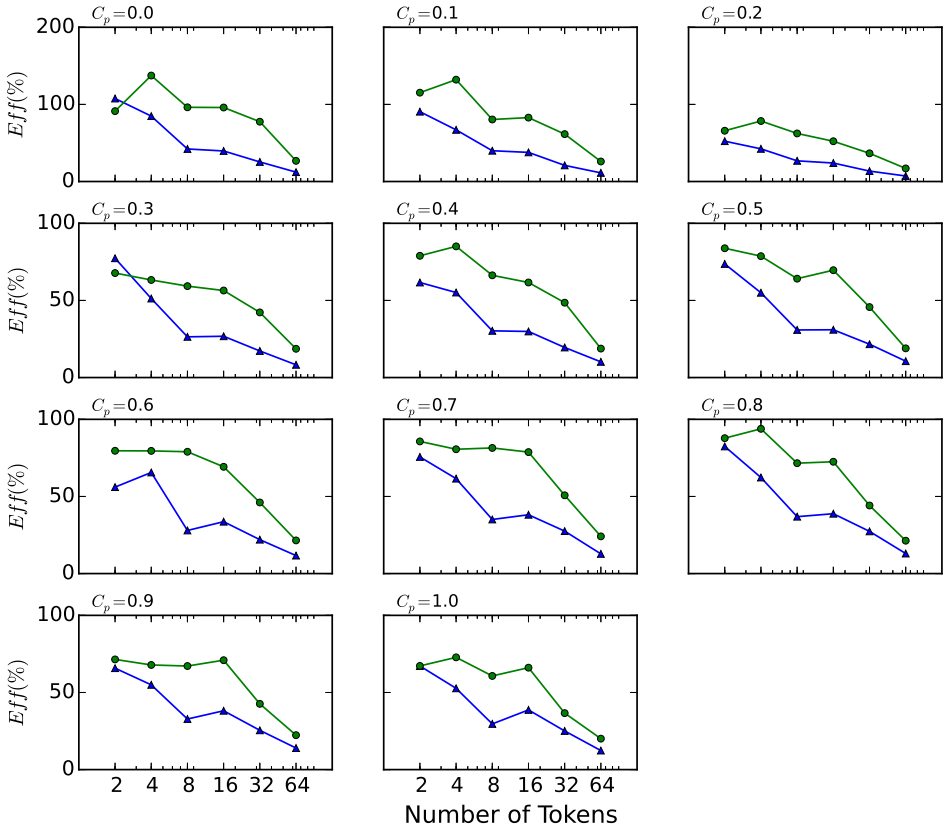

Figure 8.2: Efficiency (Eff) for Horner (average of 20 instances for each data point). Tree parallelization is the green line which is indicated by circles and Tree Parallelization with virtual loss is the blue line which is indicated by triangles. Note that Tree Parallelization with virtual loss has a lower efficiency meaning lower performance.

of C_p. With four tokens (a parallel thread can run each token/task) we see that both methods have similar SO for all values for C_p. However, plain Tree Parallelization has smaller SO than Tree Parallelization with the virtual loss on all points, which is opposite to our expectation. The second factor is Time Efficiency (Eff). We hope for the increase of Eff by using virtual loss. Figure 8.2 shows the Eff of each method. We see that plain Tree Parallelization outperforms Tree Parallelization with the virtual loss in almost all tokens for all values of C_p, which is opposite to our expectation. The only exception is when the number of tokens is 4 and C_p is 0 and 0.3.

Interestingly, adding virtual loss degrades the performance of lock-free Tree Par-

allelization in the selected problems. This outcome may be due to several factors. We mention two of them. (1) Virtual loss enables parallel threads to search different parts of the shared tree, thus reducing the synchronization overhead caused by using the locks [SKW10]. However, when the algorithm is lock-free, there is no such overhead. (2) Virtual loss disturbs the exploitation-exploration balance of the UCT algorithm.

8.6 Answer to the Second Part of RQ5

In this chapter, we addressed part two of *RQ5*.

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

We investigated the virtual loss method (i.e., an artificial increase in exploration) for task-level parallelization of the lock-free Tree Parallelization algorithm (see Section 8.1). Our preliminary results using an application from the High Energy Physic domain shows that when a virtual loss is used for lock-free Tree Parallelization, there is almost no improvement in performance. That is our provisional conclusion of part two of *RQ5*. We showed that (1) the virtual loss method suffered from a high search overhead and that (2) it suffered from a low time efficiency (see Section 8.5).

Originally virtual loss was designed to improve the performance of lock-based Tree Parallelization for the game of Go and not for lock-free Tree Parallelization. If this trend continues, then the new setting (without virtual loss) is (according to our findings) to be preferred. Therefore, we recommend not to use virtual loss along with the task-level parallelization of the lock-free Tree Parallelization algorithm to achieve higher performance.

8.7 A Complete answer to RQ5

In Chapter 7 and Chapter 8, we answered *RQ5* in two parts. Here we provide a complete answer for *RQ5*.

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

Chapter 7 provided an answer for part one of *RQ5*. We investigated to what extent a solution (i.e., adjusting the exploitation-exploration balance with respect to the tree size) is for improving the quality of search in Ensemble UCT/Root Parallelization. Previous studies on Ensemble UCT provided inconclusive evidence on the effectiveness

of Ensemble UCT. 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 (or ensemble size). Our results provide clear evidence that the performance of Ensemble UCT will be improved by selecting higher exploitation for smaller search trees given a fixed number of playouts or a fixed search budget.

Chapter 8 presented an answer for part two of *RQ5*. We analyzed a solution (i.e., an artificial increase in exploration called virtual loss) for the lock-free Tree Parallelization algorithm (see Section 8.1). Our preliminary results using an application from the HEP domain showed that when a virtual loss is used for lock-free Tree Parallelization, there is almost no improvement in performance. Moreover, we showed that the virtual loss method suffered from (1) a high search overhead and (2) a low time efficiency (see Section 8.5). As stated in Section 8.6, we recommend not to use virtual loss along with the task-level parallelization of the lock-free Tree Parallelization algorithm to achieve higher performance.