Cover Page

Universiteit Leiden

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

5

A Lock-free Algorithm for Parallel MCTS

This chapter[1] addresses *RQ3* which is mentioned in Section 1.7.

- ***RQ3***: *How can we design a correct lock-free tree data structure for parallelizing MCTS?*

One of the approaches for parallelizing MCTS for shared-memory systems is Tree Parallelization. The method is called so because a search tree is shared among multiple parallel threads. Each iteration of the MCTS has four operations (SELECT, EXPAND, PLAYOUT, and BACKUP). They are executed on the shared tree simultaneously. The MCTS algorithm uses the tree for storing the states of the domain and guiding the search process. The basic premise of the tree in MCTS is relatively straightforward: (a) nodes are added to the tree in the same order as they were expanded and (b) nodes are updated in the tree in the same order as they were selected. Therefore the following holds, if two parallel threads are performing the task of adding (EXPAND) or updating (BACKUP) the same node, there are potentially *race conditions*. Thus, one of the main challenges in Tree Parallelization is the prevention of *race conditions*.

In a parallel program a race condition shows a non-deterministic behavior that is generally considered to be a programming error [Wil12]. This behavior occurs when

[1] Based on:

- S. A. Mirsoleimani, H. J. van den Herik, A. Plaat and J. Vermaseren, A Lock-free Algorithm for Parallel MCTS, in Proceedings of the 10th International Conference on Agents and Artificial Intelligence - Volume 2, 2018, pp. 589--598.

parallel threads perform operations on the same memory location without proper *synchronization* and one of the memory operations is a write. A program with a race condition may operate correctly sometimes and fail other times. Therefore, proper synchronization helps to coordinate threads to obtain the desired runtime order and avoid a race condition.

There are two lock-based methods to create synchronization in Tree Parallelization: (1) a coarse-grained lock, (2) a fine-grained lock [CWvdH08a].

Both methods are straightforward to design and to implement. However, locks are notoriously bad for parallel performance, because other threads have to wait until the lock is released. This is called *synchronization overhead*. The fine-grained lock has less synchronization overhead than the coarse-grained lock [CWvdH08a]. Yet, even fine-grained locks are often a bottleneck when many threads try to acquire the same lock. Hence, a *lock-free* tree data structure for parallelized MCTS is desirable and has the potential for maximal concurrency. A tree data structure is lock-free when more than one thread must be able to access its nodes concurrently. Here, the problem is that the development of a lock-free tree for parallelized MCTS is shown to be non-trivial. The difficulty of designing an adequate data structure stimulated the researchers in the community to come up with a spectrum of ideas [EM10, BG11]. As a case in point, Enzenberger et al. compromised over the correctness of computation. They accepted faulty results to have a lock-free search tree [EM10]. Below, we propose a new lock-free tree data structure without compromises together with a corresponding algorithm that uses the tree for parallel MCTS.

The remainder of this chapter is organized as follows. Section 5.1 describes the shared data structure challenge. Section 5.2 discusses related work. Section 5.3 gives the proposed lock-free algorithm. Section 5.4 shows implementation considerations. Section 5.5 presents the experimental setup, Section 5.6 describes experimental design, Section 5.7 provides the experimental results, and Section 5.8 provides an answer to RQ3.

5.1 Shared Data Structure Challenge

One of the difficulties for parallelizing MCTS is protecting a shared search tree without using locks to avoid synchronization overhead. The difficulty of this process caused the researchers in the MCTS community to even compromise over correctness of computation to have a lock-free search tree [EM10]. Below we discuss parallelization with a single shared tree in Subsection 5.1.1 and race conditions in Subsection 5.1.2. Subsection 5.1.3 provides the data protection methods for a shared tree.

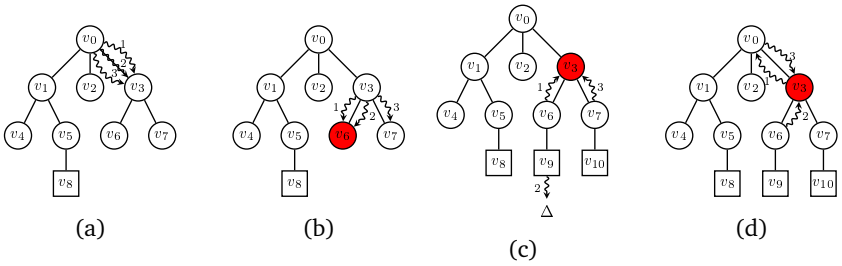


Figure 5.1: (5.1a) The initial search tree. The internal and non-terminal leaf nodes are circles. The terminal leaf nodes are squares. The curly arrows represent threads. (5.1b) Thread 1 and 2 are expanding node v_6. (5.1c) Thread 1 and 2 are updating node v_3. (5.1d) Thread 1 is selecting node v_3 while thread 2 is updating this node.

5.1.1 Parallelization with a Single Shared Tree

There are three parallelization methods for MCTS (i.e., *Root Parallelization*, *Leaf Parallelization*, and *Tree Parallelization*) that belong to two main categories: (A) parallelization with an ensemble of trees, and (B) parallelization with a single shared tree. The parallelization methods that belong to the former category (i.e., Root and Leaf Parallelization) do not need a shared search tree. But the methods that belong to the latter category use a shared search tree such as Tree Parallelization. In Tree Parallelization, parallel threads are potentially able to perform different MCTS operations on a same node of the shared tree [CWvdH08a]. These shared accesses are the source of the potential *race conditions*.

5.1.2 The Race Conditions

In parallel MCTS, parallel threads are manipulating a shared search tree concurrently. If two threads are performing the task of adding or updating the same node, there is a *race condition*.

Definition 5.1 (Race Condition) *A race condition occurs when concurrent tasks perform operations on the same memory location without proper synchronization and one of the memory operations is a write [MRR12].*

Consider the example search tree in Figure 5.1. Three parallel threads (1, 2 and 3 from v_0 to v_3) attempt to perform MCTS operations on the shared search tree. There are three race condition scenarios.

- Shared Expansion (SE): Figure 5.1b shows two threads (1 and 2) concurrently performing EXPAND(v_6). In this SE scenario, synchronization is required. Obvi-

ously, a race condition exists if two parallel threads intend to add node v_9 to v_6 simultaneously. In such an SE race, the child node should be created and added to its parent only once.

- Shared Backup (SB): Figure 5.1c shows two threads (1 and 3) concurrently performing BACKUP(v_3). In the SB scenario, synchronization is required because there are two data race conditions when parallel threads update the value of $Q(v_3)$ and $N(v_3)$ simultaneously. There are two dangers: (a) the value of either $Q(v_3)$ or $N(v_3)$ could be corrupted due to concurrently writing them, and (b) the variable $Q(v_3)$ and $N(v_3)$ could be in an inconsistent state when the writing of their values does not happen together at the same time (i.e., the state of one variable is ahead of the other one).
- Shared Backup and Selection (SBS): Figure 5.1d shows thread 2 performing BACKUP(v_3) and thread 3 performing SELECT(v_3). In the SBS scenario, synchronization is required. Otherwise, a race condition may occur between (i) thread 3 reading the value of $Q(v_3)$, and (ii) before thread 3 can read the value of $N(v_3)$, thread 2 updates the value of $Q(v_3)$ and $N(v_3)$. Thus what happens is that when thread 3 reads the value of $N(v_3)$, the variables $Q(v_3)$ and $N(v_3)$ are not in the same state anymore and therefore thread 3 reads an inconsistent set of values ($Q(v_3)$ and $N(v_3)$).

Code with race conditions may operate correctly sometimes and fail other times. We have tp protect the shared data to avoid uncertainty in the execution.

5.1.3 Protecting Shared Data Structure

There are two groups of methods to protect a shared data structure, lock-based methods and lock-free methods.

Lock-based Methods use mutexes and locks to create synchronization and protect the shared data. The first obvious design used one mutex to protect the entire search tree, but later ones used more than one mutex to protect smaller parts of the search tree and allow a greater level of concurrency in accesses to the search tree [CWvdH08a]. Locks are notoriously bad for parallel performance, because other threads have to wait until the lock is released, and locks are often a bottleneck when many threads try to acquire the same lock. If we can write a search tree data structure that is safe for concurrent accesses without locks, there is the potential for maximum concurrency.

Definition 5.2 (Lock-based) *A data structure is lock-based when it uses mutexes and locks to create synchronization to protect the shared data.*

Lock-free Methods use a lock-free data structure. Such a data structure often uses the compare/exchange operation to make progress in an algorithm, rather than protecting a part that makes progress. For example, when modifying a shared variable, an approach using locks would first acquire the lock, then modify the variable, and finally release the lock. A lock-free approach would use compare/exchange to modify the variable directly. This requires only one memory operation rather than three, but designing a lock-free data structure is hard and needs extreme care.

Definition 5.3 (Lock-free) *A data structure is lock-free when more than one thread must be able to access it concurrently.*

5.2 Related Work

In this section, we present the related work for two categories of synchronization methods for Tree Parallelization: (1) lock-based methods and (2) lock-free methods.

5.2.1 Lock-based Methods

As already mentioned, one of the main challenges in Tree Parallelization is to prevent date race conditions using synchronization. Figure 5.2 shows the Tree Parallelization where two threads (1 and 2) simultaneously perform the EXPAND operation on a node (v_6) of the tree. There are two methods to create synchronization in this case for Tree Parallelization: (1) coarse-grained lock [CWvdH08a], (2) fine-grained lock [CWvdH08a]:

1. The coarse-grained lock method uses one lock to protect the entire search tree [CWvdH08a]. For example, in Figure 5.2a, both thread 1 and 2 want to expand node v_6, then thread 1 first acquires a lock; subsequently, it performs the EXPAND operation and finally releases the lock. During this process thread 2 also wanting to perform the EXPAND operation on node v_6 should wait for the release of the lock (see Figure 5.2b). This method is called coarse-grained because the access to the tree for performing the EXPAND operation will be given to one and only one thread, even if multiple threads want to expand different nodes inside the tree. For example, in Figure 5.2a, thread 3 also wants to perform the EXPAND operation, but on node v_7. However, the lock is already acquired by thread 1. Therefore, thread 3 should wait until the lock is released (see Figure 5.2b).

2. The fine-grained lock method uses one lock for each node of the tree to protect a smaller part of the search tree and to allow a greater level of concurrency in accesses to the search tree [CWvdH08a]. For example, in Figure 5.3a, thread 3

(a) (b)

Figure 5.2: Tree parallelization with coarse-grained lock.

(a) (b)

Figure 5.3: Tree parallelization with fine-grained lock.

also wants to perform the EXPAND operation, but on node v_7. It can acquire the lock in v_7 and should not wait (see Figure 5.2b).

Both lock-based methods use locks to protect shared data. However, these approaches suffer from synchronization overhead due to thread contentions and do not scale well [CWvdH08a]. A lock-free method can remove these problems.

5.2.2 Lock-free Methods

A lock-free implementation exists in the FUEGO package [EM10]. However, the method in [EM10] does not guarantee the computational consistency of the multithreaded program with the single-threaded program. To address the SE race condition, Enzenberger et al. assign to each thread an own memory array for creating nodes [EMAS10]. Only after the children are fully created and initialized, they are linked to the parent node. Of course, this causes memory overhead. What usually happens is the following. If several threads expand the same node, only the children created by the last thread will be used in future simulations. It can also happen that some of the children that are lost in this way already received some updates; these updates will also be lost. It means that Enzenberger et al. ignore the SB and SBS race conditions. They accept the possible faulty updates and the inconsistency of parallel computation.

In the PACHI package [BG11], the method in [EM10] is used for performing lock-

free tree updates. Again, it means that both SB and SBS race conditions are neglected. However, to allocate children of a given node, PACHI does not use a per-thread memory pool as FUEGO does, but uses instead a pre-allocated global node pool and a single atomic increment instruction updating the pointer to the next free node. This addresses the memory overhead problem in FUEGO. However, there are still two other issues with this method: (1) the number of required nodes should be known in advance, and (2) the children of a node may not be assigned in consecutive memory locations which results in poor *spatial locality* (i.e., if a particular memory location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future). The spatial locality is specifically important for the SELECT operation.

5.3 A New Lock-free Tree Data Structure and Algorithm

We show our new lock-free tree data structure in Algorithm 5.1. The type name is *Node*. The UCT algorithm that uses the proposed data structure is given in Algorithm 5.2 (for the difference, see the end of this section).

Algorithm 5.1 uses the new multithreading-aware memory model of the C++11 Standard [Wil12]. To avoid the race conditions, the ordering of memory accesses by the threads has to be enforced [Wil12]. In our lock-free approach, we use the synchronization properties of the *atomic* operations to enforce an ordering between the accesses. We have used the atomic variants of the built-in types (i.e., *atomic_int* and *atomic_bool*); they are lock-free on the most popular platforms. The standard atomic types have different member functions such as *load()*, *store()*, *exchange()*, *fetch_add()*, and *fetch_sub()*. The differences are subtle. The member function *load()* is a load operation, whereas the *store()* is a store operation. The *exchange()* member function is special. It replaces the stored value in the atomic variable by a new value and automatically retrieves the original value. Therefore, we use two memory models for the memory-ordering option for all operations on atomic types: (1) *sequentially consistent* ordering (*memory_order_seq_cst*) and (2) *acquire_release* ordering (*memory_order_acquire* and *memory_order_release*). The default behavior of all atomic operations provides for *sequentially consistent* ordering. This implies that the behavior of a multithreaded program is consistent with a single threaded program. In the *acquire_release* ordering model, *load()* is an *acquire* operation, *store()* is a release operation, *exchange()* or *fetch_add()* or *fetch_sub()* are either *acquire*, *release* or both (*memory_order_acq_rel*).

In Algorithm 5.1 each node v stores nine different pieces of data: (1) a the action to be taken, (2) p, the current player at node v, (3) w_n (a 64-bit atomic integer)

Algorithm 5.1: The new lock-free tree data structure.

```
1  type
2      type a : int;
3      type p : int;
4      type w_n : atomic_int_64;
5      type children : Node*[];
6      type is_parent := false : atomic_bool;
7      type n_nonexpanded_children := -1 : atomic_int;
8      type is_expandable := false : atomic_bool;
9      type is_fully_expanded := false : atomic_bool;
10     type parent : Node*;
11     Function CreateChildren(actions) : <void>
12         if is_parent.exchange(true) is false then
13             j := 0;
14             while actions is not empty do
15                 choose a' ∈ actions;
16                 add a new child v' with a' as its action and p' as its player to the list of children;
17                 j := j+1;
18             n_nonexpanded_children.store(j);
19             is_expandable.store(
20             true,memory_order_release);
21     Function AddChild() : <Node*>
22         index := -1;
23         if is_expandable.load(memory_order_acquire) is true then
24             if (index := n_nonexpanded_children.fetch_sub(1)) is 0 then
25                 is_fully_expanded.store(true);
26             if index < 0 then
27                 return current node;
28             else
29                 return children[index];
30         else
31             return current node;
32     Function IsFullyExpanded() : <bool>
33         return is_fully_expanded.load();
34     Function Get() : <int,int>
35         w_n' := w_n.load();
36         w := high 32 bits of w_n';
37         n := low 32 bits of w_n';
38         return ⟨w, n⟩;
39     Function Set(int Δ)
40         w_n' := 0;;
41         high 32 bits of w_n' := Δ;
42         low 32 bits of w_n' := 1;
43         w_n.fetch_add(w_n');
44     Function UCT(int n) : <float>
45         ⟨w', n'⟩ := Get();
46         return w'/n' + 2C_p √(2 ln(n)/n')
47 Node;
```

Algorithm 5.2: The Lock-free UCT algorithm.

```
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,State 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 := v_l;
13     return ⟨v, s⟩;
14 Function EXPAND(Node* v,State s) : <Node*,State>
15     if s is non-terminal then
16         actions := set of untried actions from state s;
17         v.CREATECHILDREN(actions);
18         v' := v.ADDCHILD();
19         if v' is not v then
20             v := v';
21             s := v.p takes action v.a from state s;
22     return ⟨v, s⟩;
23 Function PLAYOUT(Node* v,State s)
24     while s is non-terminal do
25         choose a ∈ set of untried actions from state s uniformly at random;
26         s := the current player p takes action a from state s;
27     Δ⟨v.p⟩ := reward for state s for each player p;
28     return Δ
29 Function BACKUP(Node* v,Δ) : void
30     while v is not null do
31         v.SET(Δ⟨v.p⟩);
32         v := v.parent;
```

that stores both the total simulation reward $Q(v)$ and the visit count $N(v)$, (4) the list of children, (5) the is_parent flag (an atomic boolean) that shows whether the list of children is already created, (6) $n\_nonexpanded\_children$ the number of children that are not expanded yet, (7) the $is\_expandable$ flag (an atomic Boolean) that shows whether v is ready to be expanded, (8) the $is\_fully\_expanded$ flag (an atomic Boolean) that shows whether all children of v are already expanded and (9) $parent$ that points to the parent of v. By using (a) the atomic variables, (b) the atomic operations, and (c) the associated memory models, we can solve all the three above cases of race conditions (SE, SB, and SBS).

- SE: To solve the SE race condition, the EXPAND operation in Algorithm 5.2 consists of two separate sub-operations: (A) the CREATECHILDREN operation and (B) the ADDCHILD operation. The first operation has four key steps (A-1, A-2, A-3, A-4) which are given in Algorithm 5.1. (A-1): Exchanging the value of is_parent from $false$ to $true$ prevents the other threads to create the list of children (Line 12). Thus, the problem that the list of children is created by two threads at the same time is solved. (A-2): Creating the list of children (Line 14--18). (A-3): Set the value of $n\_nonexpanded\_children$ to counter j (Line 19), (A-4): Set the value of $is\_expandable$ to $true$ (Line 20). After a node successfully has become a parent, one of the non-expanded children in its list of children can be added using the ADDCHILD operation. The ADDCHILD operation in Algorithm 5.1 has three key steps (B-1, B-2, B-3). (B-1): Read the value of $is\_expandable$ (Line 24), if it is $true$, try to expand a new child (Line 25--32). Otherwise, return the current node (Line 34). (B-2): The value of $index$ is calculated (Line 25), if it is zero, then node v is fully expanded (Line 26). (B-3): $index$ shows the next child to be expanded (Line 31), if $index$ becomes negative, the current node is returned (Line 29).

- SB: To solve the SB race condition, Algorithm 5.1 uses a single 64-bit atomic integer w_n for storing both variables $Q(v)$ and $N(v)$. The value of $Q(v)$ is stored in the high 32 bits of w_n, while the value of $N(v)$ is stored in the low 32 bits. This compression technique preserves the correct state of the variables $Q(v)$ and $N(v)$ in all threads because they should always be written together using a SET operation. Therefore, we have no faulty updates and guarantee consistency of computation.

- SBS: To solve the SBS race condition, Algorithm 5.2 always reads variable w_n by a GET operation in the SELECT operation. The GET operation always reads the value of $Q(v)$ and $N(v)$ together. If a BACKUP operation wants to update the variable w_n at the same time, it happens through a SET operation which

Algorithm 5.3: The pseudo-code of the GSCPM algorithm.

```
1  Function GSCPM(State s_0,nPlayouts,nTasks)
2      v_0 := create a shared root node with state s_0;
3      grain_size := nPlayouts/nTasks;
4      t := 1;
5      for t ≤ nTasks do
6          s_t := s_0;
7          fork UCTSEARCH(v_0,s_t,grain_size) as task t;
8          t := t+1;
9      wait for all tasks to be completed;
10     return action a of best child of v_0;
```

writes the value of $Q(v)$ and $N(v)$ together. Therefore, the values of $Q(v)$ and $N(v)$ are always correct, in the same state, and consistency of computation is guaranteed.

In Algorithm 5.2, each node v is also associated with a state s. The state s is recalculated as the SELECT and EXPAND steps descend the tree. The term $\Delta\langle p(v)\rangle$ denotes the reward after simulation for each player.

5.4 Implementation Considerations

We have implemented the proposed lock-free data structure and algorithm in the *ParallelUCT* package [MPvdHV15a]. The implementation is available online as part of the package. The ParallelUCT package is an open source tool for parallelization of the UCT algorithm (see Section 2.6). It uses *task-level parallelization* to implement different parallelization methods for MCTS. We have used an algorithm called *grain-sized control parallel MCTS* (GSCPM) to implement and measure the performance of the proposed lock-free UCT algorithm. The pseudo-code for GSCPM is given in Algorithm 5.3. The GSCPM algorithm is implemented by multiple methods from different parallel programming libraries such as C++11 STL, thread pool (*TPFIFO*), TBB (*task_group*) [Rei07], and Cilk Plus (*cilk_for* and *cilk_spwan*) [Rob13] in the ParallelUCT package. More details about each of these methods can be found in [MPvdHV15a].

5.5 Experimental Setup

Section 5.5.1 discusses our case study, Section 5.5.2 explains the performance metrics, and Section 5.5.3 provides the details of hardware.

5.5.1 The Game of Hex

The performance of the lock-free algorithm is measured by using the game of Hex. The game of Hex is described in Subsection 2.4.1. Below follows complementary information needed for this chapter. Hex is a board game with a diamond-shaped board of hexagonal cells [AHH10]. In our experiments, the game is played on a board of size 11 on a side, for a total of 121 hexagons [Wei17].

In our implementation of Hex, a disjoint-set data structure is used to determine the connected stones. Using this data structure the evaluation of the board position to find the player who won the game becomes very efficient [GI91].

5.5.2 Performance Metrics

In our experiments, the performance is reported by (A) playout speedup (or speedup) and (B) playing strength (or percentage of win). We defined both metrics in Section 2.5. Here we operationalize the definitions. The scalability is the trend that we observe for these metrics when the number of resources (threads) are increasing.

5.5.3 Hardware

Our experiments were performed on a dual socket Intel machine with 2 Intel Xeon E5-2596v2 CPUs running at 2.4 GHz. Each CPU has 12 cores, 24 hyperthreads, and 30 MB L3 cache. Each physical core has 256KB L2 cache. The peak TurboBoost frequency is 3.2 GHz. The machine has 192GB physical memory. We compiled the code using the Intel C++ compiler with a -$O3$ flag.

5.6 Experimental Design

The goal of this experiment is to measure the performance and scalability of a lock-free algorithm for parallel MCTS on both multi-core and many-core shared-memory machines. We do so using the ParallelUCT packages. The package implements, highly optimized, Hex playing program, in order to generate realistic real-world search spaces.

To generate statistically significant results in a reasonable amount of time, 2^{20} playouts are executed to choose a move. The board size is 11×11. The UCT constant C_p is either 0, 0.1, or 1 in all of our experiments. To calculate the playout speedup the average of time over ten games is measured for making the first move of the game when the board is empty. The empty board is used because it has the biggest playout

time; it is the most time-consuming position (since the whole board should be filled randomly). The results are within less than 3% standard deviation.

5.7 Experimental Results

In Subsection 5.7.1 we discuss two topics. (A) the scalability is studied and the achieved playout speedup is reported, and (B) the effect of differences in values of C_p parameters on the speedup of the parallel algorithm is measured. The performance of the proposed lock-free algorithm for Tree Parallelization when playing against Root Parallelization is reported in Subsection 5.7.2.

5.7.1 Scalability and C_p parameters

As mentioned before, we are interested in strong scalability. Therefore, the search budget is fixed to $2^{20} = 1,048,576$ playouts as the number of tasks is increasing. Figure 5.4 shows the scalability of the algorithm for different parallel programming libraries on a CPU when the first move on the empty board is made. Each data point is the average of 21 games. Figure 5.4a illustrates the scalability when a coarse-grained lock is used (the graph is taken from [MPvdHV15a]) and Figure 5.4b demonstrates the scalability when the proposed lock-free method is used.

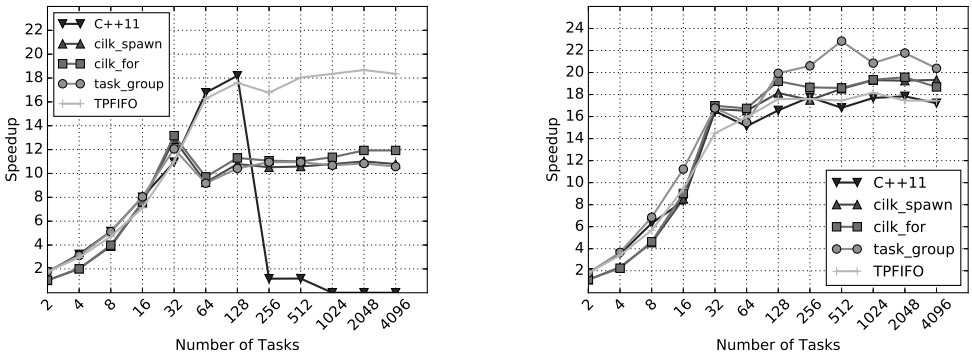
A: Playout Speedup

There are three main improvements when the lock-free tree is used (see A1 to A3).

- A1: the maximum speedup increases from 18 to 23.
- A2: the scalability of all methods is improved (it shows the notoriously bad effect of locks on the scalability for Cilk Plus, TBB, and C++11).
- A3: 32 tasks are sufficient to reach near 17 times speedup, while for the lock-based method at least 64 tasks are required.

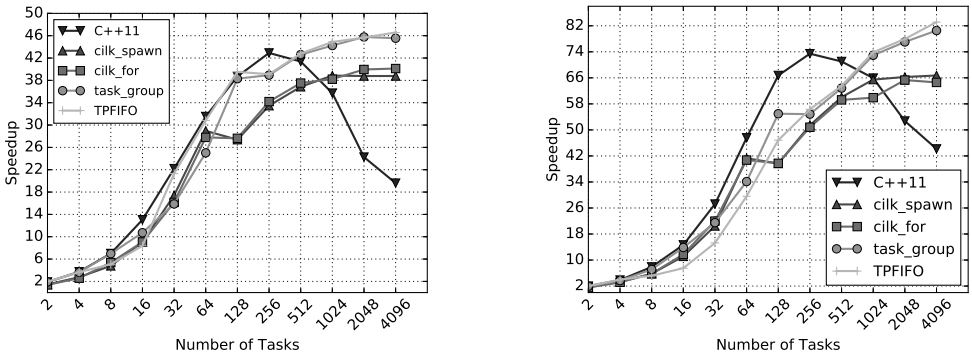
Figure 5.5 shows the scalability on the Xeon Phi. There are three main improvements when the lock-free tree is used (see A4 to A6).

- A4: the maximum speedup increases from 47 to 83.
- A5: the scalability of all methods is improved (it shows the notoriously bad effect of locks on the scalability for Cilk Plus, TBB, and C++11).
- A6: 64 tasks are sufficient to reach near 46 times speedup, while for the lock-based method at least 2048 tasks are required.



(a) (b)

Figure 5.4: The scalability of Tree Parallelization for different parallel programming libraries when $C_p = 1$. (5.4a) Coarse-grained lock. (5.4b) Lock-free.



(a) (b)

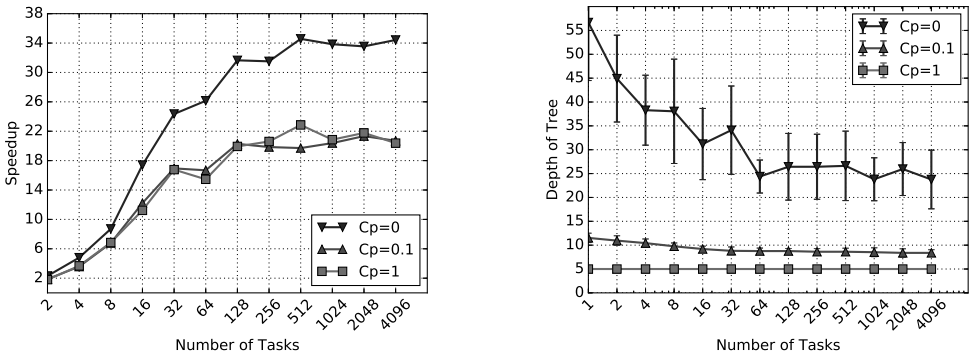
Figure 5.5: The scalability of Tree Parallelization for different parallel programming libraries when $C_p = 1$ on the Xeon Phi. (5.5a) Coarse-grained lock. (5.5b) Lock-free.

B: The Effect of C_p on Playout Speedup

Table 5.1 shows the execution time of the sequential UCT algorithm for three different C_p values. It is observed that the execution time is decreasing as the value of C_p is increasing. There is an obvious explanation for this behavior. When the algorithm uses high exploitation (i.e., low value for C_p), it constructs a search tree that is deeper and more asymmetric. In Figure 5.6b, the depth of the tree is 56 when the number of tasks is 1 and $C_p = 0$. When the shape of the tree is more asymmetric, each iteration of

Table 5.1: Sequential execution time in seconds.

C_p	Time (s)	Depth of Tree (Avg.)
0	59.97 ± 10.93	56.66 ± 12.16
0.1	26.66 ± 0.81	11.52 ± 0.98
1	20.7 ± 0.3	5



(a) (b)

Figure 5.6: (5.6a) The scalability of the algorithm for different C_p values. (5.6b) Changes in the depth of tree when the number of tasks are increasing.

the algorithm must traverse a deeper path of nodes inside the tree using the SELECT operation until it can perform a PLAYOUT operation. The SELECT operation consists of a *while loop* which for a tree with a depth of 56 has to perform 56 iterations in the worst case (see Algorithm 5.2). The BACKUP operation also consists of a while loop which for a deeper tree has more iterations. These two operations are also memory intensive ones (i.e., accessing the nodes of the tree which reside in memory). The results are that the execution time of the sequential algorithm becomes higher for high exploitation. Increasing the value of C_p means more exploration and thus a more symmetric tree with a lower depth. In Figure 5.6b, the depth of the tree is 5 when the number of tasks is 1 and $C_p = 1$. In this case, the while loop in the SELECT operation has to perform only 5 iterations in the worst case.

We have measured the scalability of the proposed lock-free algorithm for different C_p values (see Figure 5.6a). The sequential time for each C_p in Table 5.1 is used as the baseline. The maximum speedup for $C_p = 0$ is around 34. It is much higher than 23 times, the speedup when $C_p = 1$. There is a possible explanation for the

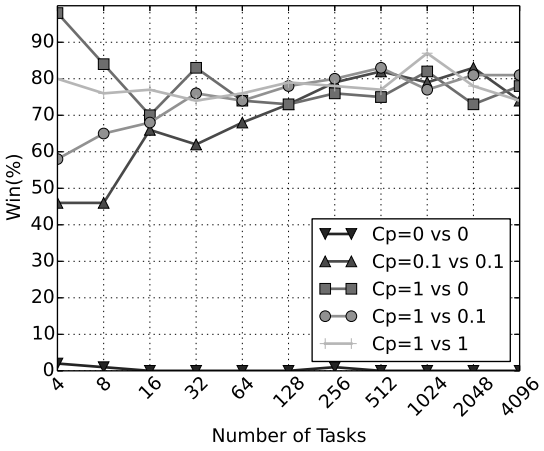


Figure 5.7: The playing results for lock-free Tree Parallelization versus Root Parallelization. The first value for C_p is used for Tree Parallelization and the second value is used for Root Parallelization.

higher speedup. The parallel algorithm may be more efficient than the equivalent serial algorithm, since the parallel algorithm may be able to avoid work that in every serialization would be forced to be performed [MRR12]. For example, Figure 5.6b shows the changes in the depth of the constructed tree with regards to the number of tasks for three different values for C_p. Increasing the number of tasks reduces the depth of the tree from 56, when the serial execution is exploitative (i.e., $C_p = 0$), to around 25. It means that, in parallel execution (a) threads explore different branches of the tree and (b) the tree is more symmetric compared to the serial execution. Hence, the number of iterations in both SELECT and BACKUP operations reduces in parallel execution and therefore causes a higher speedup. When the serial execution has high exploration (i.e., $C_p = 1$), increasing the number of tasks does not change the depth of the tree.

5.7.2 GSCPM vs. Root Parallelization

Figure 5.7 presents the result of playing Hex between the proposed lock-free Tree Parallelization against Root Parallelization. Root parallelization is also a parallelization method that does not use locks because it uses an ensemble of independent search trees. Therefore, it is interesting to see the performance of the proposed lock-free algorithm versus Root Parallelization. Figure 5.7 reports the percentage of wins for lock-free Tree Parallelization for five different combinations of C_p. Both methods use

the same number of tasks. For each data point, 100 games are played.

When $C_p = 0$ for both algorithms, Tree Parallelization cannot win against Root Parallelization. It shows that the high speedup for $C_p = 0$ (see Figure 5.6a) is not useful. However, when the value of C_p is selected to be more exploratory, the lock-free Tree Parallelization is superior to Root Parallelization, specifically for a higher number of tasks.

5.8 Answer to RQ3

In this chapter we presented the lock-free tree data structure for parallelization of MCTS. As such, this chapter proposes solutions for the following question.

- ***RQ3***: *How can we design a correct lock-free tree data structure for parallelizing MCTS?*

To answer *RQ3* we have found our way step by step. We did so in three steps.

First, we remark that the existing Tree Parallelization algorithm for MCTS uses a shared search tree to run the iterations in parallel (see Subsection 5.1.1). Here we observe that the shared search tree has potential race conditions (see Subsection 5.1.2).

Our second step is to overcome this obstacle (see Section 5.3). In this section, we have shown that having a correct lock-free data structure is possible. To achieve this goal we have used methods from modern memory models and atomic operations (see Section 5.3). Using these methods allows removing of synchronization overhead. Hence, we have implemented the new lock-free algorithm that has no race conditions (see Section 5.4).

The third step was to evaluate the lock-free algorithm. Therefore we performed an extensive experiment in a small area (Hex on a 11×11 board), see Sections 5.5 and 5.6.

To conclude, the experiment showed that the lock-free algorithm had a better performance and scalability when compared to other synchronization methods (see Section 5.7). The performance of task-level parallelization to implement the lock-free GSCPM algorithm on a multi-core machine with 24 cores was very good. It reached a speedup of 23 and showed very good scalability for up to 4096 tasks. The performance on a many-core co-processor was also very good; a speedup of 83 on the 61 cores of the Xeon Phi was reached. In summary, the Xeon Phi showed very good scalability for up to 4096 tasks.