2 Background

The MCTS algorithm iteratively repeats four steps or operations to construct a search tree until a predefined computational budget (i.e., time or iteration constraint) is reached [CWvdH$^+$08b]. Algorithm 2.1 shows the general MCTS algorithm (see Section 1.2 and Algorithm 2.2).

Algorithm 2.1: The general MCTS algorithm.

```
Function MCTS($s_0$)
    $v_0$ := create root node with state $s_0$;
    while within search budget do
        $\langle v_l, s_l \rangle$ := SELECT($v_0, s_0$);
        $\langle v_l, s_l \rangle$ := EXPAND($v_l, s_l$);
        $\Delta$ := PLAYOUT($v_l, s_l$);
        BACKUP($v_l$,$\Delta$);
```

The purpose of MCTS is to approximate the domain-dependent theoretic value of the actions that may be selected from the current state by iteratively creating a partial search tree [BPW$^+$12]. How the search tree is built depends on how nodes in the tree are selected (i.e., tree selection policy). For instance, nodes in the tree are selected according to the estimated probability that they are better than the current best action. It is essential to reduce the estimation error of the nodes' values as quickly as possible. The current chapter provides a more detailed overview of MCTS. Section 2.1 describes the UCB selection policy. In Section 2.2, we provide the UCT formula and the UCT algorithm. Section 2.3 discusses the parallelization methods for MCTS. Section 2.4 presents the benchmarks for experimental studies. Section 2.5 explains the performance metrics. Finally, Section 2.6 briefly describes our software tool.

2.1 Upper Confidence Bound (UCB)

The tree selection policy in the MCTS algorithm is based on two fundamentally different concepts, viz. exploitation and exploration. Hence, the selection is a search process and the aim of the search is to reduce the error as soon as possible [KS06].

Definition 2.1 (Exploitation) *Exploitation looks in areas which appear to be promising [BPW$^+$12].*

Definition 2.2 (Exploration) *Exploration looks in areas that so far have not been sampled well [BPW$^+$12].*

Kocsis and Szepesvári [KS06] aimed to design a Monte Carlo search algorithm that had a small error probability if stopped prematurely and that converged to the domain-dependent theoretic optimum given sufficient time [KS06]. They proposed the use of the simplest Upper Confidence Bound (UCB) policy (i.e., UCB1) as a tree selection policy for MCTS. UCB1 is an obvious choice for node selection given its application in multi-armed bandit problems for balancing between exploitation and exploration of actions. Bandit problems are a well-known class of sequential decision problems, in which one needs to choose among *K* actions (e.g., the *K* arms of a multi-armed bandit slot machine) to maximize the cumulative reward by consistently taking the optimal action [BPW$^+$12, ACBF02].

Auer et al. [ACBF02] proposed UCB1 for *bandit problems*. The UCB1 policy selects the arm j that maximizes:

$$UCB1(j) = \overline{X}_j + \sqrt{\frac{2\ln(n)}{n_j}} \tag{2.1}$$

where $\overline{X}_j$ is the average reward from arm j; n_j is the number of times arm j was played, and n is the overall number of plays so far. The first term at the right-hand side $\overline{X}_j$ encourages the exploitation of higher-reward arms, while the second term at the right-hand side $\sqrt{\frac{2\ln(n)}{n_j}}$ promotes the exploration of less played arms.

2.2 Upper Confidence Bounds for Trees (UCT)

This section explains the most common algorithm in the MCTS family, the Upper Confidence Bounds for Trees (UCT) algorithm. The formulas are given in Subsection 2.2.1 and the algorithm in Subsection 2.2.2

2.2.1 UCT Formula

The UCT algorithm addresses the exploitation-exploration dilemma in the selection step of the MCTS algorithm using the UCB1 policy [KS06]. A child node j is selected to maximize:

$$UCT(j) = \overline{X}_j + 2C_p\sqrt{\frac{2\ln(N(v))}{N(v_j)}} \tag{2.2}$$

where $\overline{X}_j = \frac{Q(v_j)}{N(v_j)}$ is an approximation of the node j domain-dependent theoretic value. $Q(v_j)$ is the total reward of all playouts that passed through node j, $N(v_j)$ is the number of times node j has been visited, $N(v)$ is the number of times the parent of node j has been visited, and $C_p \geq 0$ is a constant. The first term at the right-hand side is for exploitation and the second term is for exploration [KS06]. The decrease or increase in the amount of exploration can be adjusted by C_p in the exploration term.

2.2.2 UCT Algorithm

The UCT algorithm is given in Algorithm 2.2. Each node v stores four pieces of data: the action to be taken $a(v)$, $p(v)$ the current player at node v, the total simulation reward $Q(v)$ (a real number), and the visit count $N(v)$ (a non-negative integer). Each node v is also associated with a state s. The state s is recalculated as the SELECT and EXPAND steps descends the tree. The term $\Delta\langle p(v)\rangle$ denotes the reward after simulation for each player.

2.3 Parallelization Methods for MCTS

In this section, two categories for parallelization of MCTS are presented. Traditionally, parallelization methods for MCTS are classified based on the parallelism technique. Currently, we believe that we should classify them into two categories solely based on the way that the search tree is used. We introduce parallel methods with a shared tree in Subsection 2.3.1 and with an ensemble of search trees in Subsection 2.3.2.

2.3.1 Parallel Methods with a Shared Data Structure

The first category is for the parallel methods with a shared search tree. The tree is shared among parallel threads or processes which means data is accessible globally. The methods that belong to this category can be implemented on both shared-memory and distributed-memory systems. In both environments, a synchronization method should create constraints threads from accessing the tree simultaneously. The most well-known method in this category is Tree Parallelization.

Algorithm 2.2: The UCT algorithm.

```
Function UCTSEARCH($s_0$)
    $v_0$ := create root node with state $s_0$;
    while within search budget do
        $\langle v_l, s_l \rangle$ := SELECT($v_0, s_0$);
        $\langle v_l, s_l \rangle$ := EXPAND($v_l, s_l$);
        $\Delta$ := PLAYOUT($v_l, s_l$);
        BACKUP($v_l$,$\Delta$);
    return $a(best\ child\ of\ v_0)$
Function SELECT(Node $v$,State $s$) : <Node,State>
    while $v$ is fully expanded do
        $v_l := \underset{v_j \in children\ of\ v}{\arg\max} \frac{Q(v_j)}{N(v_j)} + 2C_p\sqrt{\frac{2\ln(N(v))}{N(v_j)}}$;
        $s_l := p(v)$ takes action $a(v_l)$ from state $s$;
        $v := v_l$;
        $s := s_l$;
    return $\langle v, s \rangle$;
Function EXPAND(Node $v$,State $s$) : <Node,State>
    if $s$ is non-terminal then
        choose $a \in$ set of untried actions from state $s$;
        add a new child $v^{'}$ with $a$ as its action to $v$;
        $s^{'} := p(v)$ takes action $a$ from state $s$;
    return $\langle v^{'}, s^{'} \rangle$;
Function PLAYOUT(Node $v$,State $s$)
    while $s$ is non-terminal do
        choose $a \in$ set of untried actions from state $s$ uniformly at random;
        $s := p(v)$ takes action $a$ from state $s$;
    $\Delta\langle p(v) \rangle$ := reward for state $s$ for each player $p$;
    return $\Delta$
Function BACKUP(Node $v$,$\Delta$) : void
    while $v$ is not null do
        $N(v) := N(v) + 1$;
        $Q(v) := Q(v) + \Delta\langle p(v) \rangle$;
        $v$ := parent of $v$;
```

Definition 2.3 (Tree Parallelization) *In Tree Parallelization, the tree is shared among parallel threads, tasks, or processes which means data is accessible globally.*

2.3.2 Parallel Methods with More than one Data Structure

The second category is for the parallel methods where several search trees or an ensemble of search trees are used. Each parallel thread has its own search tree which means the information is local to that thread. The methods that belong to this category can also be implemented on both shared-memory and distributed-memory environments. The most well-known method in this category is Root Parallelization.

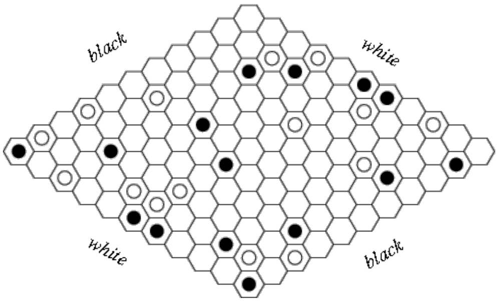

Figure 2.1: A sample board for the game of Hex

Definition 2.4 (Root Parallelization) *In Root Parallelization, each parallel thread, task, or process has its own search tree which means the information is local to that thread.*

2.4 Case Studies

In this section, we present two case studies for MCTS. In Subsection 2.4.1 we present the game of Hex, a strategy board game for two players. In Subsection 2.4.2 we describe the method for approximating the roots of a polynomial called Horner scheme.

2.4.1 Case 1: The Game of Hex

Hex is a board game with a diamond-shaped board of hexagonal cells [AHH10, HT19]. The game is usually played on a board of size 11 on a side, for a total of 121 hexagons, as illustrated in Figure 2.1 [Wei17]. Each player is represented by a color (Black or White). Players take turns placing a stone of their color on a cell on the board. The goal for each player is to create a connected chain of stones between the opposing sides of the board marked by their colors. The first player to complete this path wins the game. The game cannot end in a draw since no path can be completely blocked except by a complete path of the opposite color. Since the first player to move in Hex has a distinct advantage, the swap rule is generally implemented for fairness. This rule allows the second player to choose whether to switch positions with the first player after the first player has made a move.

Evaluation Function

In our implementation of Hex, a disjoint-set data structure is used to determine the connected stones. A disjoint-set data structure maintains a collection of disjoint (non-overlapping) subsets of a set of elements $S = \{S_1, S_2, \ldots, S_k\}$. A union-find algorithm performs two operations on such a data structure: First, the *Find* operation determines in which subset a particular element is located. This can be used for determining whether two elements are in the same subset. Second, the *Union* operation joins two subsets into a single subset. Each set is identified by a representative, which usually is a member in the set. Using this data structure and algorithm, the evaluation of the board position to find the player who won the game becomes very efficient [GI91].

2.4.2 Case 2: Horner Schemes

Horner's rule is an algorithm for polynomial computation that reduces the number of multiplications and results in a computationally efficient form [OS12]. For a polynomial in one variable

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0, \tag{2.3}$$

the rule simply factors out powers of x. Thus, the polynomial can be written in the form

$$p(x) = ((a_n x + a_{n-1})x + \ldots)x + a_0. \tag{2.4}$$

This representation reduces the number of multiplications to n and has n additions. Therefore, the total evaluation cost of the polynomial is $2n$. Here it is assumed that the cost of addition and multiplication are equal.

Horner's rule can be generalized for multivariate polynomials. Here, Eq. 2.4 applies to a polynomial for each variable, treating the other variables as constants. The order of choosing variables may be different, each order of the variables is called a *Horner scheme*.

The number of operations can be reduced even more by performing common subexpression elimination (CSE) after transforming a polynomial with Horner's rule [ALSU07]. CSE creates new symbols for each subexpression that appears twice or more and replaces them inside the polynomial. Then, the subexpression has to be computed only once.

2.5 Performance Metrics

In our experiments, the performance is reported by two metrics: (A) playout speedup (Subsection 2.5.1) and (B) playing strength (Subsection 2.5.2). Below, we define both metrics.

2.5.1 Playout Speedup

The most important metric related to performance and parallelism is *speedup*. In the literature, this form of speedup is called *playout speedup* [CWvdH08a]. We use playout speedup to show the effect on a program's performance in terms of speed of execution after any resource enhancement (e.g., increasing the number of threads or cores). The speedup can be defined for two different types of quantities: (A1) latency and (A2) throughput.

A1: Playout speedup in latency

We measure the speedup in time, which is a latency measure. Speedup compares the time for solving the identical computational problem on one worker versus that on P workers

$$PlayoutSpeedup_{latency} = \frac{T_1}{T_P}. \tag{2.5}$$

where T_1 is the time of the program with one worker and T_P is the time of the program with P workers. In our results we report the scalability of our parallelization as *strong scalability* which means that the problem size remains fixed as P varies. The problem size is the number of playouts (i.e., the search budget) and P is the number of threads or tasks.

Definition 2.5 (Strong Scalability) *Strong scalability means that the problem size remains fixed as the number of resources varies.*

A2: Playout speedup in throughput

We measure the speedup in Playouts per Second (PPS), which is a throughput measure. First, we execute the program with one thread, which yields a PPS of n. Next, we execute the program with P threads, which yields a PPS of m. Using the speedup formula gives

$$PlayoutSpeedup_{throughput} = \frac{Q_{\text{P}}}{Q_1} = \frac{m\text{ PPS}}{n\text{ PPS}} \tag{2.6}$$

2.5.2 Playing Strength

The second most important metric related to the performance of parallel MCTS is *playing strength*. We use playing strength to show the effect on a program's performance in terms of quality of search after any resource enhancement (e.g., increasing the number of threads or cores). Playing strength can be defined for two different types of problems: (B1) two-player game and (B2) optimization problem.

B1: Playing Strength in a two player game

We measure the strength of player a in Percentage of Wins (PW) per tournament versus player b for the two-player game, such as Hex or Go, which is a win-rate

$$PlayingStrength(a)_{PW} = \frac{W_a}{W_a + W_b} * 100. \tag{2.7}$$

where W_a is the number of wins for player a and W_b is the number of wins for player b. If there is a draw, it will be counted as a win for both players.

B2: Playing strength in an optimization problem

We measure the strength of an MCTS player a in the *number of operations* in the optimized expression, which is a solution for the Horner scheme optimization problem. A lower value is desirable when we increase the numbers of threads or tasks.

2.6 Our ParallelUCT Package

To be able to investigate the research questions of this thesis a new software framework for parallel MCTS has been developed. The tool has been designed from scratch and is implemented in C++. Our tool is named *ParallelUCT*. The tool is open source, and its source codes are accessible [1].

The ParallelUCT framework has many features that enable us to answer the research questions mentioned in Section 1.7. Below we describe the three most important elements of this package, viz. in Subsection 2.6.1 we present multiple benchmark problems that are provided by the ParallelUCT, in Subsection 2.6.2 we describe multiple parallelization methods in ParallelUCT, and in Subsection 2.6.3 we provide the list of parallel programming models that are used in ParallelUCT.

2.6.1 Framework of multiple benchmark problems

In our research we focus on two benchmark problems. They are our case studies (Hex and Horner schemes). Both are implemented in the ParallelUCT framework. This software framework is extensible. It means that new problems such as other games or optimization problems can be added to it quickly. A developer should solely implement the original problem and provide it to the framework. The only requirement is to follow the standard of implementation which is provided by the software framework. The standard is available in the documentation of the ParallelUCT package [MPvdHV15a].

[1] `http://github.com/mirsoleimani/paralleluct`

2.6.2 Framework of multiple parallelization methods

We focus on two parallelization methods. They are methods with a shared data structure and with more than one data structure which are implemented in the framework. Examples of such methods are Tree Parallelization and Root Parallelization. A user can run the ParallelUCT executable program using one of these methods via command line options. The complete list of command line options is accessible via the help option of the program (see `http://github.com/mirsoleimani/paralleluct`).

2.6.3 Framework of multiple programming models

Finally, we focus on programming models. The parallelization methods are implemented in the framework using modern threading libraries such as Cilk [LP98], TBB [Rei07], and C++11. A user can run the ParallelUCT executable program with one of these threading libraries also via command line options. The complete list of command line options is accessible via the help option of the ParallelUCT executable program (see `http://github.com/mirsoleimani/paralleluct`).