Parallel Monte-Carlo Tree Search in Distributed Environments

by

Marc Christoph

Thesis presented in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science

at the University of Stellenbosch

Supervisor: Prof. R. S. Kroon

Co-supervisor: Dr. C. P. Inggs

March 2020

The financial assistance of the Centre for Artificial Intelligence Research is hereby acknowledged.

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: March 2020 . . . .

Abstract

Parallel Monte-Carlo Tree Search in Distributed Environments

M. Christoph

Computer Science Division,
Department of Mathematical Sciences,
University of Stellenbosch,
Private Bag X1, 7602 Matieland, South Africa.

Thesis: M.Sc. Computer Science

2020

Parallelising Monte-Carlo Tree Search (MCTS) holds the promise of being an effective way to improve the effectiveness of the search, given some time constraint. Thus, finding scalable parallelisation techniques has been an important area of research since MCTS was first proposed. The inherently serial nature of MCTS makes effective parallelisation difficult, since care must be taken to ensure that all threads or processes have access to accurate statistics. This is more challenging in distributed-memory environments due to the latency incurred by network communication.

Prior proposals of distributed MCTS have presented their results on different domains and hardware setups, making them difficult to compare. To try to improve this state of affairs, we use the actor-based framework Akka to implement and compare various distributed MCTS algorithms on a common domain—the board game Lines of Action (LOA). We describe our implementation and evaluate the scalability of each approach in terms of playouts per second (PPS), unique nodes searched per second (NPS), and playing strength.

We observe that distributed root parallelisation provides the best PPS scalability, but has relatively poor scalability in terms of NPS. We contrast this with distributed tree parallelisation which scales well in terms of NPS but performs poorly in terms of PPS. Distributed leaf parallelisation is shown to scale up to 128 compute nodes in terms of PPS, but its NPS scalability is limited by its single compute node that manages the tree.

We determine that distributed root parallelisation combined with tree parallelisation is the strongest of the distributed MCTS algorithms, with none of our other implementations managing a win-rate of more than 50% against the algorithm. We show that distributed root/leaf parallelisation, as well as our distributed leaf parallelisation with a multi-threaded traverser scale well in terms of playing strength. Distributed tree parallelisation via TDS, df-UCT and UCT-Treesplit is shown to have limited playing strength scalability, and we provide possible avenues for future work that may resolve this limited performance.

We hope that these findings will provide future researchers with sufficient recommendations for implementing distributed MCTS programs.

Uittreksel

Parallelle Monte-Carlo Boomsoektogte in Verspreide Omgewings

M. Christoph
Afdeling Rekenaarwetenskap,
Departement van Wiskundige Wetenskappe,
Universiteit van Stellenbosch,
Privaatsak X1, 7602 Matieland, Suid Afrika.

Tesis: M.Sc. Rekenaarwetenskap
2020

Parallelisering van die Monte-Carlo Boomsoektog algoritme (MCTS) blyk ‘n effektiewe manier te wees om die doeltreffendheid van soektogte, onderhewig aan ’n gegewe tydsbeperking, te verbeter. Dus, is die ontwikkeling van skaaleerbare parallelisasietegnieke ‘n belangrike navorsingsarea sedert MCTS voorgestel is. Die inherente sekwensiële aard van MCTS maak effektiewe parallelisering moeilik, en tegnieke wat verseker dat alle liggewigprosesse toegang tot akkurate statistieke het, moet ondersoek word. Die deel van statistieke is meer uitdagend in verspreide geheue omgewings as gevolg van die latensie wat veroorsaak deur netwerkkommunikasie.

Vorige voorstelle van verspreide MCTS algoritmes is getoets vir verskillende take en hardeware, wat dit moeilik maak om hulle resultate met mekaar te vergelyk. Dus gebruik ons die akteur-gebaseerde raamwerk Akka om verskillende verspreide MCTS-algoritmes te implementeer en op dieselfde taak—die bordspel Lines of Action (LOA)—te toets en te vergelyk. Ons beskryf die implementasie daarvan en evalueer die skaaleerbaarheid van elke benadering in terme van simulasies per sekonde (PPS), unieke nodusse per sekonde (NPS) en spelkrag.

Ons bewys dat verspreide wortelparallelisering die beste UPS-skaaleerbaarheid bied, maar relatief swak skaaleerbaarheid in terme van NPS. Ons kontrasteer dit met verspreide boomparallelisering wat goed skaaleer in terme van NPS, maar wat swak presteer in terme van PPS. Daar word getoon dat verspreide blaarparallelisering tot 128 berekenings-nodusse in terme van PPS skaaleer, maar dat die NPS-skaaleerbaarheid daarvan beperk word deur die gebruik van 'n enkele nodus wat die boom bestuur.

Ons bepaal dat verspreide wortelparallelisering gekombineer met boomparallelisering die sterkste is van die verspreide MCTS-algoritmes, en geen van ons ander implementerings het 'n wen-koers van meer as 50 % teen hierdie algoritme nie. Ons toon aan dat verspreide wortel/blaarparallelisering, sowel as ons verspreide blaarparallelisering met 'n multi-liggewigproses deurstapalgoritme, goed skaleer ten opsigte van spelkrag. Daar word getoon dat verspreide boomparallalisering swak spelkrag-skaaleerbaarheid toon, en ons bied idees vir toekomstige werk wat hierdie swak prestasie moontlik kan oplos.

Ons hoop dat hierdie bevindings sal toekomstige navorsers voldoende aanbevelings sal gee vir die implementering van verspreide MCTS-programme.

Acknowledgements

I would like to express my sincere gratitude to the following individuals and organisations for their support throughout the course of this work:

- My supervisors, Prof. R. S. Kroon and Dr. C. P. Inggs for their consistent guidance, commitment and willingness to go above and beyond the call of duty.
- The Centre for Artificial Intelligence Research for their financial support.
- The Centre for High Performance Computing for supplying the computational resources required to complete this work.
- My parents, Ralph and Nadine, for their continued support and invaluable advice.
- My girlfriend, Lydia, for providing daily love and encouragement.
- My friends that sat beside me for the late nights and early mornings on campus, and were always there for the much needed breaks in-between.

Contents

Declaration **i**

Abstract **ii**

Uittreksel **iv**

Acknowledgements **vi**

Contents **vii**

List of Figures **x**

List of Tables **xii**

1 Introduction **1**
1.1 Problem Statement . . . 2
1.2 Objectives . . . 3
1.3 Thesis Outline . . . 4

2 Background **5**
2.1 Game Trees and Combinatorial Search . . . 6
2.2 Monte-Carlo Tree Search . . . 9
2.2.1 Algorithm Overview . . . 10
2.2.2 Upper Confidence Bound For Trees (UCT) . . . 12
2.2.3 Enhancements . . . 14
2.2.3.1 First Play Urgency . . . 16
2.2.3.2 Progressive Unpruning . . . 16
2.2.3.3 Progressive Bias . . . 17
2.2.3.4 Transposition Tables . . . 18

2.2.4 Parallelisation 20
2.2.4.1 Leaf Parallelisation 22
2.2.4.2 Root Parallelisation 24
2.2.4.3 Tree Parallelisation 24
2.3 Akka Actors and Clustering 26
2.4 Lines of Action 29
2.4.1 The Quad Heuristic 30
2.5 Summary 33

3 Related Work 34
3.1 Performance Measures and Scalability 35
3.2 Distributed Leaf Parallelisation 35
3.3 Distributed Root Parallelisation 36
3.4 Distributed Tree Parallelisation 38
3.4.1 Transposition Table Driven Scheduling 38
3.4.2 Depth-First UCT 40
3.4.3 UCT-Treesplit 42
3.5 Summary 44

4 Design and Implementation 46
4.1 Lines of Action 46
4.1.1 Move Generation 47
4.1.1.1 Generating All Legal Moves 49
4.1.1.2 Generating A Random Legal Move 50
4.1.2 Terminal State Detection 51
4.1.3 Incorporation of Knowledge 52
4.1.3.1 Evaluation Function 52
4.1.3.2 Move Categories 53
4.2 Game Tree Representation 54
4.2.1 Tree Node 55
4.2.2 Transposition Table 56
4.2.2.1 Best Index Retrieval 57
4.2.2.2 Table Update 58
4.3 System Overview 62
4.3.1 Cluster Design 62
4.3.2 Cluster Initialisation and Match Execution 64

4.4 Distributed MCTS Agents 70
4.4.1 Distributed Leaf Parallelisation 71
4.4.2 Distributed Root Parallelisation 72
4.4.3 Distributed Tree Parallelisation 74
4.4.3.1 Vanilla TDS 74
4.4.3.2 df-UCT 76
4.4.3.3 UCT-Treesplit 77
4.5 Summary 78

5 Experiments and Results **80**
5.1 Experimental Setup 80
5.2 Parameter Tuning 81
5.2.1 Tuning with CLOP 82
5.2.2 Final Parameter Choices 84
5.2.3 Tuning N_{par} 85
5.2.3.1 Playout Rate 86
5.2.3.2 Playing Strength 89
5.2.4 Transposition Table Size 92
5.3 Scalability Analysis 93
5.3.1 Playout Rate 93
5.3.2 Tree Size 96
5.3.3 Playing Strength 98
5.4 Summary 101

6 Conclusion **103**
6.1 Future Work 105
6.2 Final Remarks 106

A Move Categories **107**

Bibliography **108**

List of Figures

2.1 A portion of the game tree for tic-tac-toe. 7
2.2 A depiction of a single MCTS iteration. 12
2.3 An illustration of leaf, root and tree parallelisation. 22
2.4 A high-level depiction of an Akka cluster consisting of three nodes. 28
2.5 The initial LOA board state. 29
2.6 A LOA board state used to clarify move legality. 29
2.7 An example of a terminal LOA board state. 29
2.8 A LOA board state containing a hole. 31
2.9 Possible quads for a LOA board. 31

3.1 A simple tree where the left-most child of every node has the best UCT value 41

4.1 A depiction of a LOA board state used to clarify our approach to move generation 47
4.2 Line decompositions for horizontal, vertical and diagonal line orientations. 48
4.3 An example line configuration with two legal moves. 49
4.4 A template for the quads used for terminal state detection. 51
4.5 A depiction of incremental quad updates via the `xor` operation. . 52
4.6 Board regions used for move classification. 54
4.7 A high-level overview of our test framework. 64
4.8 A sequence diagram depicting the initialisation process for an agent's cluster. 66
4.9 A sequence diagram depicting a single move being made in a match. 69
4.10 A sequence diagram depicting a full match being run. 70

5.1 PPS achieved by TDS at turn 1 for varying values of N_{par} and an increasing number of CNs. 87
5.2 PPS achieved by df-UCT at turn 1 for varying values of N_{par} and an increasing number of CNs. 88
5.3 PPS achieved by UCT-Treesplit at turn 1 for varying values of N_{par} and an increasing number of CNs. 88
5.4 Win-rate achieved by TDS after 50 matches versus a serial agent for varying values of N_{par} and an increasing number of CNs. . . 91
5.5 Win-rate achieved by df-UCT after 50 matches versus a serial agent for varying values of N_{par} and an increasing number of CNs. 91
5.6 Win-rate achieved by UCT-Treesplit after 50 matches versus a serial agent for varying values of N_{par} and an increasing number of CNs. 92
5.7 Playouts per second (PPS) achieved by our distributed MCTS agents with an increasing number of CNs. 94
5.8 Unique nodes expanded per second by our distributed MCTS agents with an increasing number of CNs. 97
5.9 Win-rates achieved by our distributed MCTS implementations against our root/tree agent with 1CN 101

List of Tables

5.1 Results of tuning our serial agent with CLOP. 84
5.2 Results of a round-robin tournament used to make our final choice of MCTS enhancements. 85

A.1 Transition probabilities for the move categories used by our agents 107

Chapter 1

Introduction

Humans have been playing games for entertainment and competition since the first civilised societies developed many centuries ago [43]. The complex nature of some games make them an effective domain to test computational intelligence, and the idea that computers could be used to play games dates back to 1950, when Claude Shannon proposed the first chess-playing computer program [50]. Since then, games have been an integral part of artificial intelligence (AI) research.

Game-playing engines may select moves by traversing a *game tree* (Section 2.1) using the *minimax* algorithm [45] to find optimal moves. However, the computational resources required to do this makes this approach infeasible for most games [28].

$\alpha\beta$ pruning [33] was developed in an attempt to minimise the computational resources required to perform the search by pruning less beneficial parts of the game tree. Additionally, $\alpha\beta$ pruning programs typically employ a depth-limited search and use a domain-dependent evaluation function to estimate the value of non-terminal game states. $\alpha\beta$ pruning has been the dominant approach employed by AI researchers for decades. However, it has been less successful in games with state representations that are computationally expensive to evaluate [40].

In 2006, Levente Kocsis and Csaba Szepesvári developed Monte-Carlo Tree Search (MCTS) [34, 11], a best-first search algorithm that is guided by Monte-Carlo simulations [37]. The algorithm iteratively constructs the game tree by expanding parts of the tree that show promising simulation results. The statistical significance of the information gathered by these simulations increases with an increase in the number of simulations performed [37]. Therefore, simulation results serve as an effective game state evaluation that does not require built-in domain-specific knowledge. This has enabled MCTS to dominate $\alpha\beta$ pruning in domains such as Go [39], Havannah [55] and Hex [4].

One way to improve the decision-making ability of an MCTS program is to maximise the number of Monte-Carlo simulations that the program performs. Parallelising MCTS therefore plays an important role in the development of stronger programs, especially since hardware that supports parallelism (multi-core CPUs, multi-CPU machines and large computer clusters) has become commonplace in recent years. However, effective parallelisation is not as simple as for classical $\alpha\beta$-based search techniques [17]. This is because simulation results must be shared to prevent threads from expanding and traversing parts of the game tree that other threads may have already determined to be unpromising [17, 62, 47].

1.1 Problem Statement

Ideally, a parallel MCTS implementation running on n cores will perform as well as a serial implementation that is given n times more time to run. However, this is not the case. The node statistics that guide the search change rapidly, and need to be constantly available at all compute nodes for the search to prioritise more promising parts of the tree. Therefore, a parallel MCTS implementation must either share node statistics among the available compute nodes in order to perform an effective search, or limit sharing at the risk of wasting time searching less beneficial parts of the game tree.

Root parallelisation (Section 2.2.4.2), leaf parallelisation (Section 2.2.4.1) and tree parallelisation (Section 2.2.4.3) are the three most prominent techniques that exist for parallelising MCTS [16, 17]. In leaf and tree parallelisation, a single game tree is maintained and shared among the available threads. Because of this, these two approaches lend themselves well to shared-memory (multi-core) environments. In root parallelisation, each thread maintains its own game tree, and node statistics are combined at the end of the search. This means that root parallelisation is more suitable for distributed memory environments (clusters).

Transposition Table Driven Work Scheduling (TDS) (Section 3.4.1) was developed by Romein et al. [44] to efficiently distribute a single game tree across multiple compute nodes, allowing tree parallelisation to be used effectively in distributed memory environments. A major bottleneck in TDS is the need for frequent communication at the compute node responsible for managing the root of the game tree. Yoshizoe et al. [62] and Schaefers et al. [47] proposed *Depth-First UCT* (df-UCT) (Section 3.4.2) and *UCT-Treesplit* (Section 3.4.3), respectively, to mitigate this problem.

These distributed MCTS algorithms were tested on different domains using different hardware setups, which makes fair comparison challenging. In light of this, our goal is to determine the scalability of the following distributed MCTS algorithms in terms of playouts per second, tree size and playing strength. We use *Lines of Action* (LOA) (Section 2.4) as a common test domain:

- Distributed leaf parallelisation (see Section 3.2 for background and Section 4.4.1 for implementation details).
- Distributed root parallelisation combined with either leaf parallelisation or tree parallelisation (see Section 3.3 for background and Section 4.4.2 for implementation details).
- Distributed tree parallelisation with TDS (see Section 3.4.1 for background and Section 4.4.3.1 for implementation details).
- df-UCT (see Section 3.4.2 for background and Section 4.4.3.2 for implementation details).
- UCT-Treesplit (see Section 3.4.3 for background and Section 4.4.3.3 for implementation details).

1.2 Objectives

We identify the following objectives to achieve the goal described above:

- Implement the test domain (LOA).
- Implement each of the distributed MCTS algorithms discussed in Section 1.1.

- Identify the strongest set of enhancements to apply to our distributed MCTS algorithms.
- Identify the strongest set of hyperparameters for our distributed MCTS algorithms.
- Determine the scalability of each distributed MCTS algorithm in terms of simulations per second.
- Determine the scalability of each distributed MCTS algorithm in terms of unique nodes expanded per second.
- Determine the scalability of each distributed MCTS algorithm in terms of playing strength.
- Use the results of these scalability experiments to determine how playing strength is influenced by playout rate and tree size.
- Provide a comprehensive analysis and comparison of these distributed MCTS algorithms using a common test domain and hardware setup.

1.3 Thesis Outline

The remainder of this thesis is structured as follows: Chapter 2 provides the background necessary to understand the remainder of the thesis. This includes information on game trees and classical search, MCTS, including its enhancements and parallelisation, Akka and actor systems, and LOA. Chapter 3 presents the existing literature for applying parallel MCTS algorithms to distributed environments and analyses the scalability of each approach. Chapter 4 discusses the design and implementation of each of our distributed MCTS agents, as well as our test framework and LOA. In Chapter 5, we present our experimental setup and results. This includes parameter tuning and the scalability of each of our implementations in terms of playout rate, tree size and playing strength. We conclude the thesis with Chapter 6, where we reflect on our results in terms of the objectives presented in Section 1.2. Additionally, we provide possible avenues for future work that could shed light on, or improve upon, the work presented in this thesis.

Chapter 2

Background

Games have been an integral part of Artificial Intelligence (AI) research since Claude Shannon developed the first chess-playing computer program [50]. *Combinatorial games* have been particularly popular for AI researchers since they typically have a simple set of rules but are also complex enough to provide significant research challenges [11]. By definition, two-player combinatorial games have the following properties [11]:

- *Zero-sum*: Both players are in direct competition with each other, i.e. a gain in utility for one player implies an equivalent loss in utility for the opposing player.
- *Perfect information*: The full state of the game is visible to both players.
- *Deterministic*: Chance does not play any role in the game.
- *Sequential*: Players perform actions sequentially, not simultaneously.
- *Discrete*: The number of possible game states and actions are finite.

In this chapter, we provide the background required to contextualise our research and discuss the literature leading up to distributed Monte-Carlo tree search with a focus on two-player combinatorial games. We introduce the concept of a *game tree* and consider classical search techniques in Section 2.1. Section 2.2 introduces Monte-Carlo tree search and some enhancements to the core algorithm, as well as prevalent parallelisation techniques. Section 2.3 provides some background on the concurrency framework we use for our implementations (see Section 4.3 for details). We conclude the chapter with a summary of our test domain, Lines of Action, in Section 2.4.

2.1 Game Trees and Combinatorial Search

Combinatorial search algorithms attempt to find a specified combinatorial object in a defined search space. Combinatorial search problems can be tackled by reducing the problem to a tree where the root node represents the initial problem to be solved, other nodes represent (possibly interacting) sub-problems, and branches represent actions that may taken to reduce a problem to one of its sub-problems [42].

The goal of a game-playing AI, or *agent*, is to determine the best possible move for a given game state. For many games (including combinatorial games), this can be accomplished through the use of a combinatorial search algorithm where the search space consists of all possible game states in the domain. Each node in the tree represents a single game state and each edge represents a possible action, or *move*, that can be applied to the state [41]. Such a tree is referred to as a *game tree*.

Figure 2.1 depicts the game tree for tic-tac-toe. Although tic-tac-toe has a small number of possible game states, the full game tree is still too large to feasibly depict here. Therefore, we omit portions of the tree and replace most of the omitted portions by ellipses.

Game states that represent completed games are called *terminal states*, and an AI agent can use a *utility function* to assign numerical values to terminal states based on the winner of the game. In general, terminal states where the agent performing the search has won are assigned higher values than those where it draws or loses. In Figure 2.1, the utility function assigns a value of 1 to a terminal state where **X** has won, 0 for a draw, and -1 when **O** has won.

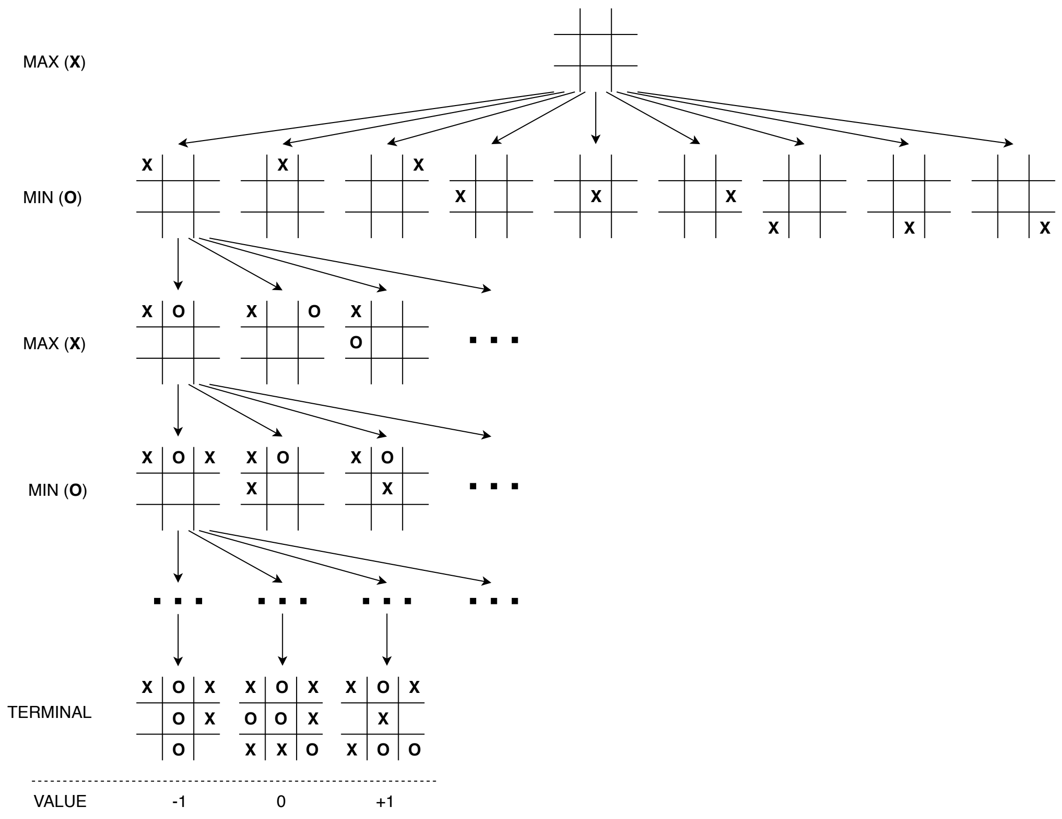

Figure 2.1: A portion of the game tree for tic-tac-toe.

A game tree like the one depicted in Figure 2.1 can in theory be traversed by the *minimax* algorithm to produce perfect play. In the minimax algorithm, the values assigned to terminal states by the utility function are backpropagated up the tree with the ultimate goal of determining the utility of every child of the root so that the optimal move can be chosen.

In minimax, it is customary to refer to the searching agent as `MAX` and its opponent as `MIN`. In the game tree shown in Figure 2.1, the engine playing crosses is `MAX`, since it is expected to make a move first.

The game tree is traversed recursively by minimax in a depth-first manner with play alternating between `MAX` and `MIN`. Once a terminal state is reached, the utility of the game state is determined and associated with the node. When minimax has assigned utility values to all the siblings of the terminal node, it assigns a utility value to the node's parent in the following way:

- If `MIN` is expected to make a move at the parent, it is assigned the minimum of its children's utility values. This is because minimax assumes that `MIN` plays optimally and will choose the best move possible from its perspective.

- Similarly, if `MAX` is expected to make a move at the parent, it is assigned the maximum of its children's utility values.

This process is applied recursively until all children of the root node have been assigned utility values. The optimal move is then the one leading to the child with the best utility value for `MAX`.

The size of the search space (the state-space complexity) varies drastically between games, with tic-tac-toe having an upper bound of $3^9 = 19683$ possible states, and Chess having a state-space complexity in the order of 10^{43} [50]. Using minimax to traverse a game tree containing all possible game states allows an agent to make the optimal move at every turn in principle. However, for a game tree with a uniform branching factor b (the number of children for internal nodes in the tree) and depth d, the number of leaf nodes evaluated by minimax is b^d. This exponential growth in the number of nodes with increased depth makes exact minimax infeasible for games with a large state-space complexity [28].

Sometimes, minimax will unnecessarily search nodes that cannot affect the final move choice. In order to address this inefficiency, researchers conceived of the idea to *prune*, or ignore, nodes by stopping the evaluation of a node once it is proven that the node is worse than a previously encountered one. The most effective and widely-used such enhancement to minimax is known as *$\alpha\beta$ pruning* and the resultant algorithm is called $\alpha\beta$ search.[1] Pruning is performed in such a way that the solutions found by $\alpha\beta$ search are equivalent to those found by standard minimax [33].

The technique introduces two new variables, α and β, that represent the best utility values encountered for `MAX` and `MIN`, respectively. These values are initialised to the worst-case utility values, i.e. $\alpha = -\infty$ and $\beta = +\infty$.

α and β are updated as the search advances, with the difference between α and β becoming progressively smaller with an increase in depth. When β

[1]It is difficult to determine who initially conceived of $\alpha\beta$ search, but the recognition for the algorithm's conception is given to John McCarthy and Alexander Brudno [12], who independently developed the ideas that would eventually become the $\alpha\beta$ search that is used today.

becomes smaller than α, the node being searched can no longer be the result of optimal play by `MAX` and `MIN`, and the subtree rooted at the node is pruned.

The average number of nodes evaluated by $\alpha\beta$ search when a node's children are arranged in a random order is given by $O((\frac{b}{log(b)})^d)$ while the best case is $O(b^{\frac{d}{2}})$. This is a significant improvement over standard minimax, but the running time of the algorithm still grows exponentially with an increase in game tree depth.

In order to apply $\alpha\beta$ search to domains with a larger state-space complexity, it is necessary to limit the search to some depth d as opposed to descending the full game tree up to the terminal states. This partial game tree is called a *search tree*.

Since a utility function cannot be applied to non-terminal states, a depth-limited $\alpha\beta$ search will make use of an *evaluation function* to assign heuristic values to the non-terminal leaves of the search tree. Like the utility function used in a full game tree search, the evaluation function assigns values to game states according to how beneficial the state is considered to be for the searching player (`MAX`). By making use of an evaluation function to traverse a search tree instead of a full game tree, depth-limited $\alpha\beta$ search implicitly defines a new proxy game to which $\alpha\beta$ search is applied.

Although this depth-limited version of $\alpha\beta$ search bypasses the exponential growth in the number of searched nodes, it has still proven to be ineffective for games with large branching factors and game states that are difficult to evaluate such as Go [28]. Therefore, a different approach is required to develop strong game-playing AIs for these games.

2.2 Monte-Carlo Tree Search

Monte-Carlo tree search (MCTS) was independently developed by Kocsis and Szepesvari [34] and Rémi Coulom [21] as a best-first, anytime search technique that does not require an evaluation function to determine node values. The algorithm performs simulations in the search space to estimate node values and iteratively grow a search tree while focusing on parts of the tree with the best estimated values [34, 16, 11].

The fact that MCTS does not require an evaluation function has allowed the algorithm to achieve success in domains where a high-quality evaluation function is difficult to construct such as Go. For example, early MCTS-based Go programs such as *Crazy Stone* and *Mogo* showed significant improvements over previous $\alpha\beta$-based programs [57, 20]. Additionally, in 2016, *AlphaGo* became the first computer program to beat a professional player on a full 19×19 board with no handicap [51, 39]. MCTS has also shown promise in non-game applications such as constraint satisfaction, scheduling problems and combinatorial optimisation [11].

In this section, we provide an overview of the MCTS algorithm in Section 2.2.1, followed by a discussion of the most pervasive selection policy for MCTS—upper confidence bound for trees (UCT)—in Section 2.2.2. We discuss selected MCTS enhancements in Section 2.2.3 and conclude the section with a discussion of various MCTS parallelisation techniques in Section 2.2.4.

2.2.1 Algorithm Overview

The vanilla MCTS algorithm builds a game tree asymmetrically as the search progresses. This process is guided by simulation results so that the most promising parts of the tree are preferentially expanded, allowing computational resources to be used efficiently by avoiding unnecessarily searching less beneficial subtrees.

Each iteration of the main MCTS loop is termed a *playout*, and consists of four distinct phases, as depicted in Figure 2.2. These are repeated until some computational budget (normally a time, memory or iteration constraint) has been expended. When this happens, the search is stopped and the best child of the root node is returned according to some *final move policy*. The four phases of MCTS are as follows:

1. **Selection**: Starting at the root of the game tree, child nodes are recursively selected until a terminal state or a node that is not yet fully expanded (a node with actions that have not been considered yet) is reached.

2. **Expansion**: If the game state represented by the node is not terminal, a new child is added to the node, thereby expanding the tree. If the game state is terminal, the tree is not expanded and the value of the terminal node is backpropagated (see step 4) without the need for simulation.

3. **Simulation**: Starting at the newly expanded node, moves are applied to the game state until a terminal state is reached, at which point the value of the state (commonly 1 for a win and 0 for a loss) is obtained.

4. **Backpropagation**: The reward value obtained in the simulation phase is used to update the statistics for every node on the path from the newly expanded node to the root of the tree. This normally entails incrementing each node's visit count and updating average node rewards based on the simulation result [34].

These first three phases of MCTS may be grouped into the following two policies:

1. A *tree policy* that defines how the agent descends the tree. This includes the choice of children during selection and which nodes are added to the tree during expansion. The main consideration in choosing a tree policy is balancing the *exploitation* of parts of the tree that are believed to be beneficial and the *exploration* of less-visited parts of the tree in the hope of finding more promising sub trees.

2. A *default policy* that defines how moves are applied to the game state during simulation. A possible default policy is to choose moves uniformly at random [34], but more sophisticated policies are often used instead [11].

Note that, although backpropagation does not use either of these policies, the manner in which node statistics are updated during backpropagation can differ when some enhancements, such as *Rapid Action Value Estimation* (RAVE), are used [26].

Popular final move policies include the following:

- *Max child*: Select the child with the highest reward.
- *Robust child*: Select the child with the most visits. This is the most commonly used strategy [18, 11].
- *Max-robust child*: Select the node with the most visits and the highest reward. If none exist, continue searching until a sufficient visit count is reached.

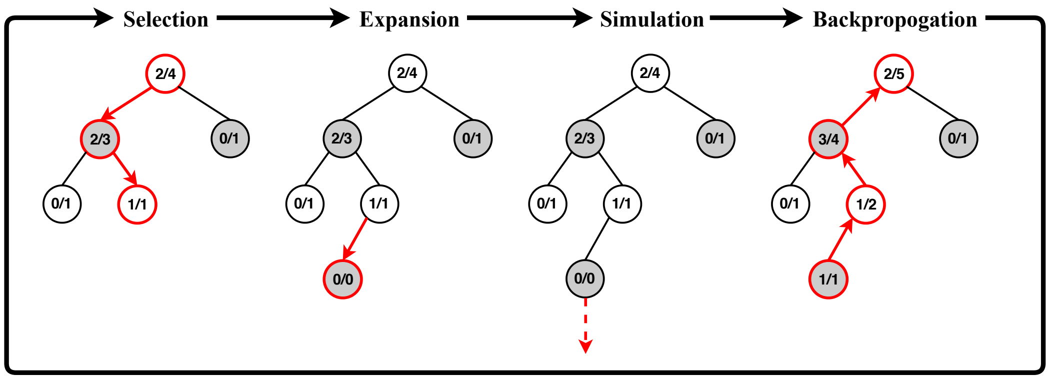

Figure 2.2: A single iteration of the MCTS algorithm. Node fill colour indicates the player to move at the node. Values inside nodes represent average rewards (total reward divided by number of playouts through the node) from the perspective of the player to move. Bold arrows indicate the path taken during selection, the newly added action/node during expansion, and the path taken during backpropagation. The dotted arrow indicates an out-of-tree simulation.

- *Secure child*: Select the child which maximises some lower confidence bound.

2.2.2 Upper Confidence Bound For Trees (UCT)

In order to develop a tree policy that addresses the exploration-exploitation tradeoff discussed in Section 2.2.1, Kocsis and Szepesvari [34] formulated the selection phase of MCTS as a *multi-armed bandit problem* (MAB)—a class of problems where one must repeatedly choose amongst K actions with the goal of maximising one's cumulative average reward over time. The choice of action is difficult to determine as the underlying reward distribution for each action is unknown, and can only be estimated based on past observations.

In the context of an MAB, *regret* after n turns refers to the expected loss (relative to the best option in hindsight) incurred by failing to select the optimal action, and the policies for action selection typically aim to minimise this regret. Lai and Robbins [35] showed that there is no action selection policy with a regret that grows slower than $O(\ln(n))$. Therefore, a policy with logarithmic regret growth is deemed to have solved the exploration-exploitation dilemma.

Among these policies is UCB1—a policy that leverages the probabilistic *upper confidence bound* (UCB) on the value of an action [5]. After n previous selections, UCB1 dictates to choose the action i that maximises:

$$\mathrm{UCB1}(i) = \overline{X}_i + \sqrt{\frac{2\ln n}{n_i}} \tag{2.1}$$

where $\overline{X}_i$ is the average reward for action i and n_i is the number of times action i was chosen previously. A higher $\overline{X}_i$ value encourages the exploitation of actions with greater accumulated reward, while the second term in the formula encourages exploration of less-selected actions.

In UCT, each selection step is modelled as a MAB where the set of available actions corresponds to the set of child nodes that may be selected and the value of a child node is approximated by previous simulations. In light of this, the average reward $\overline{X}_i$ of a child node i is given by:

$$\overline{X}_i = \frac{v_i}{n_i} \tag{2.2}$$

where n_i is the number of times a playout has passed through child i and v_i is the total reward obtained in the simulation phase of those playouts. This leads to a selection policy that dictates to select the child i that maximises:

$$\mathrm{UCT}(i) = \frac{v_i}{n_i} + 2C_{\mathrm{uct}}\sqrt{\frac{2\ln n}{n_i}} \tag{2.3}$$

where n is the number of simulations that have passed through the parent of i, i.e. $\sum_i n_i = n - 1$ and C_{uct} is a constant that determines the degree to which UCT favours less explored tree nodes.

There is a balance between the first (exploitation) and second (exploration) terms of Formula 2.3. When child i is selected, the denominator of the exploration term increases for that child, and so the contribution of the exploration term to the value of the child decreases. On the other hand, the contribution of the exploration term to the value of i's siblings increases with an increase in the numerator. The value of C_{uct} can be adjusted to lower or increase the degree of exploration. While it has been proven that $C_{\mathrm{uct}} = \frac{1}{\sqrt{2}}$ allows node values to converge to their game-theoretic values for $\overline{X}_i \in [0, 1]$ [34], the value of C_{uct} is commonly optimised empirically.

If $n_i = 0$ for some child i, The UCT value of the child is understood to be ∞, so that each child of a node is visited at least once before any of its children are expanded.[2]

After selection, the UCT algorithm progresses as described in Section 2.2.1. The child to be expanded can be chosen at random from the unexpanded children, or domain knowledge can be used to select the most promising child (see Section 2.2.3 for examples).

Following this, the default policy is used to perform a simulation at the newly expanded node. The reward obtained at the end of the simulation is then added to v_i for every node i on the path from the root, and the number of visits is incremented for each node. Modular pseudocode for UCT is provided in Algorithm 1.

2.2.3 Enhancements

Several enhancements have been proposed to improve the effectiveness of MCTS. These enhancements can be broadly grouped into the following two categories [11]:

- *Domain-dependent* enhancements require some form of domain-specific knowledge such as game mechanics and game state representation.
- *Domain independent* enhancements are applicable to any domain and do not require any prior knowledge in order to function.

In this section, we discuss three enhancements to the selection phase in UCT: first play urgency (Section 2.2.3.1), progressive unpruning (Section 2.2.3.2), and progressive bias (Section 2.2.3.3). We conclude this section with a discussion of *transposition tables*—an efficient technique for storing the tree—in Section 2.2.3.4.

[2]In practise, this is not always the case. Some MCTS enhancements assign finite UCT values to unvisited nodes, such as *first play urgency* (FPU) (Section 2.2.3.1).

Algorithm 1 UCT

```
 1: function UCT($s_0$)                                ▷ Return the best move for the state $s_0$
 2:     create root node $v_0$ with state $s_0$
 3:     while computational budget is not reached do
 4:         $v \leftarrow$ TREEPOLICY($v_0$)
 5:         $s \leftarrow$ the game state associated with $v$
 6:         $\Delta \leftarrow$ DEFAULTPOLICY($s$)
 7:         BACKUP($v$, $\Delta$)
 8:     end while
 9:     bestChild $\leftarrow$ BESTCHILD($v_0$, 0)
10:     return the move leading to bestChild
11: end function
12:
13: function TREEPOLICY($v$)                           ▷ Apply the selection process to node $v$
14:     while $v$ is not terminal do
15:         if $v$ is not fully expanded then
16:             return EXPAND($v$)
17:         else
18:             $v \leftarrow$ BESTCHILD($v$, $C$)
19:         end if
20:     end while
21:     return $v$
22: end function
23:
24: function EXPAND($v$)                               ▷ Add an unexplored child to $v$
25:     $a \leftarrow$ an unexpanded action for $v$
26:     $s \leftarrow$ result of applying $a$ to state associated with $v$
27:     add a child $c$ to $v$ with associated state $s$
28:     return $c$
29: end function
30:
31: function BESTCHILD($v$, $C$)                       ▷ Return child of $v$ with the best UCT value
32:     return the child $i$ of $v$ that maximises $\frac{Q_i}{n_i} + 2C\sqrt{\frac{2\ln n_v}{n_i}}$
33: end function
34:
35: function DEFAULTPOLICY($s$)                        ▷ Simulate from $s$ and return a reward
36:     while $s$ is not terminal do
37:         $a \leftarrow$ a random legal move for $s$
38:         apply $a$ to $s$
39:     end while
40:     $\Delta \leftarrow$ the reward for $s$
41:     return $\Delta$
42: end function
```

```
43: function BACKUP($v$, $\Delta$)          ▷ Backpropagate the reward $\Delta$, starting at $v$
44:     $i \leftarrow v$
45:     $v \leftarrow$ the parent of $v$
46:     while $v$ is not null do
47:         $n_i \leftarrow n_i + 1$
48:         $Q_i \leftarrow Q_i + \Delta(v, p)$          ▷ $\Delta(v, p)$ contains the reward for the
                                                   player $p$ to move at node $v$
49:         $i \leftarrow v$
50:         $v \leftarrow$ the parent of $v$
51:     end while
52: end function
```

2.2.3.1 First Play Urgency

In standard UCT, unvisited nodes are implicitly assigned a UCT value of ∞. This forces the algorithm to expand every child of a node n before growing the subtree rooted at n any deeper. In a domain with a large branching factor, an agent employing this approach may fail to reach a significant depth in the tree within a reasonable time constraint.

First play urgency (FPU) is a domain-independent selection enhancement proposed by Gelly et al. [27] to tackle this issue. FPU assigns a fixed value to unexplored nodes while using the UCT formula for visited nodes. This means that early exploitation will be encouraged with a low FPU value since the UCT values of nodes with good statistics will be greater than the assigned FPU values. On the other hand, higher FPU values will allow early exploration. Implementations that assign a high enough FPU value to unvisited nodes will expand every child of a tree node before applying the UCT formula, just like standard UCT.

2.2.3.2 Progressive Unpruning

Similarly to FPU, *progressive unpruning* [18] promotes early exploitation in order to explore deeper parts of the tree. Progressive unpruning achieves this by artificially restricting the number of expandable child nodes early in the search—effectively limiting the branching factor and forcing deeper exploration.

When a node is first encountered, the progressive unpruning strategy will tentatively *prune* all of its children, except for some constant number

U_0 of them, so that only the best U_0 children are initially considered in the search. For all subsequent playouts, a function $U(n)$ determines the number of children of a node to be considered, where n is the node's total visit count. When $U(n)$ is incremented, an additional move is made available (or *unpruned*). The order in which moves are unpruned is usually determined by some heuristic evaluation function [18], which makes progressive unpruning a domain-dependent enhancement. A possible definition for $U(n)$ is as follows [30]:

$$U(n) = U_0 + \left\lfloor \frac{\log(n)}{\log(\mu)} \right\rfloor . \tag{2.4}$$

Here, μ is a hyperparameter that determines the rate at which children are unpruned, with the rate increasing as μ decreases.

2.2.3.3 Progressive Bias

The reliability of a node's statistics generally increases as more playouts pass through the node [37]. This means that a node with a low visit count is more likely to have an inaccurate estimated value, and it may be beneficial to take into account an initial value of such a node through the use of domain-specific heuristic knowledge.

Chaslot et al. [18] proposed *progressive bias* as a means to incorporate heuristic knowledge into the standard UCT formula. Progressive bias estimates a node's value with the domain knowledge having a strong influence when the node has been visited a small number of times and the node's UCT statistics are less reliable. As the node's visit count increases, the domain knowledge contribution is decreased in favour of the node's increasingly accurate UCT statistics. This dynamic influence is achieved by adding a decaying bias term to the standard UCT formula (see Formula 2.3) so that, as n_i grows, the contribution of the newly added bias term becomes less significant. The modified formula is as follows:

$$\text{UCT}_{\text{pb}}(i) = \text{UCT}(i) + C_{\text{pb}} \left(\frac{H_i}{n_i + 1} \right) . \tag{2.5}$$

The right-hand term in Equation 2.5 is the *bias term* that is weighted with the constant C_{pb}. H_i is the heuristic value of child i that is usually determined by a (naïve) static evaluation function, i.e. progressive bias is a domain-dependent enhancement.

2.2.3.4 Transposition Tables

In many domains, different sequences of moves can lead to the same game state. When the same game state is reached by more than one path in the tree, we call the state a *transposition*. If an MCTS agent can not detect transpositions, it may store multiple tree nodes that represent the same game state. Since the agent accumulates statistics for these nodes independently, failure to detect transpositions can lead to a search overhead incurred by searching nodes that already have well established value estimates.

A commonly used technique to tackle this issue is to store tree nodes in a *transposition table* [52]. A transposition table is a large hash table that maps game states to table entries containing the information associated with the state. When an MCTS agent using a transposition table encounters a game state, it performs a table lookup to determine whether the state already has associated statistics stored in a corresponding table entry. If so, the agent uses the stored statistics to perform selection or updates them if the playout is in the backpropagation phase. If not, the agent inserts a new entry into the transposition table for the encountered game state and continues with the search.

In practice, transposition tables are implemented as a one-dimensional array with each array index holding a single transposition table entry. The index into the array for a game state is usually determined by computing a hash of the game state modulo the length of the array. For a game state s with hash $h(s)$, the index into a transposition table with length $|T|$ is computed as follows:

$$I(s) = h(s) \bmod |T|. \tag{2.6}$$

The most common hashing scheme for board game agents is *Zobrist hashing* [63], which maps game states to large integers (usually 64-bit or more). We outline the approach using tic-tac-toe as an example domain.

Hashing begins by populating a table of random bitstrings for each possible combination of piece and position on the board. For example, in tic-tac-toe, this consists of 2 piece types × 9 squares on the board. Pseudocode for the initialisation procedure in tic-tac-toe is provided in Algorithm 2.

Algorithm 2 Zobrist random table initialisation for tic-tac-toe.

```
X ← 0
O ← 1
randomTable ← a 2-d array of size 9 × 2

function INITZOBRIST                    ▷ Populate a table of random bitstrings
    for i in 0...8 do                   ▷ Loop over the squares on the board
        for j in 0...1 do               ▷ Loop over the pieces
            randomTable[i][j] ← getRandomBitstring()
        end for
    end for
end function
```

Once this table is populated, a game state can be broken up into piece/square components, which map to the random bitstrings generated during initialisation. The Zobrist hash of a game state can then by computed by `xor`ing appropriate bitstrings together. Pseudocode to compute the Zobrist hash for a tic-tac-toe game state represented as a 1-dimensional array of length 9 is provided in Algorithm 3.

Algorithm 3 Zobrist hash computation for tic-tac-toe after random table initialisation.

```
function GETZOBRISTHASH(gameState[])
    hash ← 0                            ▷ Initialise the Zobrist hash
    for i in 0...8 do
        piece ← gameState[i]
        hash ← hash xor randomTable[i][piece]
    end for
    return hash
end function
```

The advantage of Zobrist hashing is that the hash for a game state does not have to be fully re-computed every time a move is applied to it. Instead, the hash can be incrementally updated (e.g. as one descends the search tree during the selection phase of MCTS) by `xor`ing out the bitstrings for pieces that are removed from a square and `xor`ing in the bitstrings for pieces that are placed on a square.

When implementing a transposition table using Zobrist hashing, care must be taken to detect and handle hashing errors appropriately. The following two types of errors have been defined in the literature [63]:

- **Type 1 errors**: If the number of available hash values is smaller than the number of possible game states, there will be cases where two or more game states yield the same hash. When this happens during a search, the information in the transposition table entry will be related to a completely different game state, and this will introduce search errors. Although one can detect type 1 errors in some cases by checking that the moves stored in the table entry are legal from the game state being searched, there is no way to detect these errors with 100% certainty [9].
- **Type 2 errors**: If the number of entries in the transposition table is smaller than the number of possible game states, there will be cases where multiple game states map to the same transposition table entry. When this occurs, the transposition table implementation must determine whether to replace the existing entry with the new one or keep the existing entry and find a new entry for the game state. The technique used by the implementation to resolve type 2 errors is called a *replacement scheme* [9]. Although there is always a loss of information incurred when replacing nodes, many replacement schemes have been proposed that use metrics such as node age and depth to minimise the impact of this loss [9].

2.2.4 Parallelisation

It has been shown that MCTS performs better when more playouts are performed and a larger tree is produced [17, 37]. Therefore, when implementing an MCTS agent, it is important to maximise its playout rate and the size of the tree that it builds so that stronger moves can be found within some fixed time constraint.

An effective technique for improving the peformance of computationally expensive algorithms such as MCTS is *parallelisation*. Parallelising MCTS allows it to take advantage of multiple CPU cores on a symmetric multi-processing (SMP) machine or a cluster of networked machines to increase the rate at which playouts are performed. Parallelisation can be divided into two distinct categories:

- **Multi-core (shared-memory) parallelisation** spreads work across one or more CPUs, each with one or more cores on an SMP machine, with memory being shared by all participating cores.
- **Cluster (distributed-memory) parallelisation** distributes work across a network of inter-connected machines (a compute cluster). Each machine, or *compute node* (CN) consists of one or more CPUs, each with one or more cores that share the CNs memory.

For the purposes of this section, we use the term *compute entity* (CE) to denote a single unit of computation that may either be a CPU core in an SMP environment or a compute node in a cluster.

The effectiveness of a parallelisation technique is determined by its *scalability*. If a program running on N CEs is better than a version running on $N-1$ CEs according to some metric, the program is said to *scale* to N CEs in terms of that metric. An ideal parallel MCTS implementation would scale linearly with an increase in the number of CEs. However, parallelising any search algorithm introduces overhead that can restrict scalability [36, 48, 10], and the inherently sequential nature of MCTS makes parallelisation difficult [17]. The three notable types of overhead in parallel MCTS are as follows [54]:

1. *Search overhead* is incurred when some CEs waste time searching nodes that have been deemed unfavourable by other CEs.
2. *Synchronisation overhead* is incurred when some CEs must wait for other CEs to finish their computations before moving forward with their own.
3. *Communication overhead* is incurred when network latency causes delays in the exchange of information between CEs.

Minimising the performance impact of these overheads is of utmost importance when parallelising MCTS.

The most established MCTS parallelisation techniques are *root parallelisation*, *leaf parallelisation* and *tree parallelisation* [17]. Leaf and root parallelisation were first proposed by Cazenave and Jouandeau [14] under the names *at-the-leaves parallelisation* and *single-run parallelisation*, respectively, but we do not use this terminology. Although far more than three parallel MCTS algorithms have been proposed, they are mostly variations on these three techniques [11]. Figure 2.3 pictorially contrasts these approaches, which are discussed in detail in Sections 2.2.4.1-2.2.4.3.

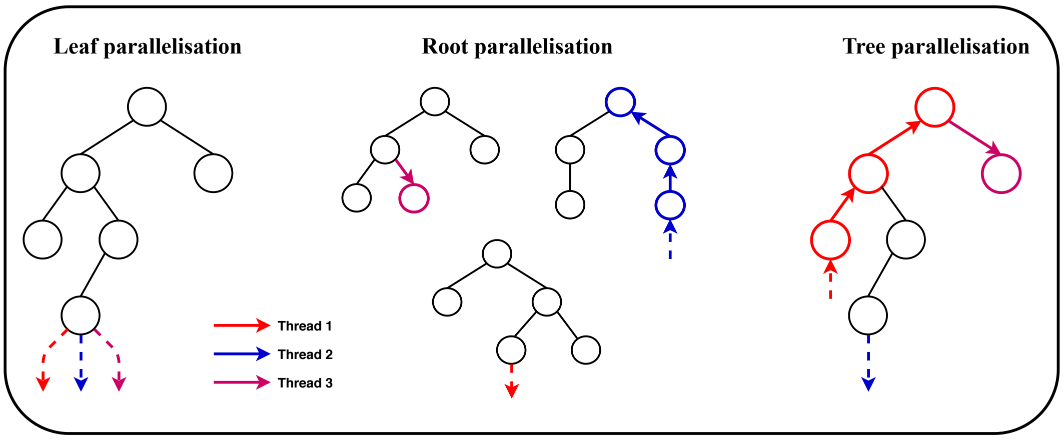

Figure 2.3: An outline of the operation of leaf, root and tree parallelisation. Dotted arrows represent the simulation phase of MCTS while solid arrows represent selection if they point towards leaf nodes and backpropagation if they point towards the root.

2.2.4.1 Leaf Parallelisation

While the in-tree phases of MCTS (selection, expansion and backpropagation—see Section 2.2) are dependent on previous simulation results, the simulation phase has no such dependency. Due to the independent nature of MCTS simulations and the fact that the simulation phase is generally the most time-consuming part of MCTS [14], leaf parallelisation aims to increase the playout rate by performing simulations in parallel while building and traversing the tree sequentially.

In leaf parallelisation, the tree is maintained by one CE (the *traverser*) and simulations are performed in parallel by the remaining CEs. The various leaf parallelisation algorithms that have been proposed can be divided into the following two distinct approaches [49]:

- *Single leaf multiple playouts* (SLMP) is the earliest approach to leaf parallelisation [14, 17]. In SLMP, the traverser performs selection and expansion as usual. When a node is added to the tree, the traverser signals to every simulator that a simulation must be performed from the newly expanded node. The traverser then waits for all simulations to finish and proceeds with backpropagation.

  SLMP suffers from the following two types of overhead:

  1. Synchronisation overhead incurred at the traverser while it waits for simulations to finish and at the simulators while they wait for the traverser to perform backpropagation and selection.
  2. If multiple simulations at a node have already resulted in a loss, it is likely that the majority of the remaining simulations will also lead to a loss. Since the traverser always waits for all simulations to finish before performing backpropagation, this could lead to computational resources being wasted on performing simulations at unpromising nodes, thereby incurring a search overhead.

- *Multiple leaves multiple playouts* (MLMP) was proposed to mitigate the overheads incurred by SLMP [15, 31]. When a node is expanded in MLMP, the traverser asynchronously sends a simulation request to a single simulator and asynchronously starts another MCTS iteration, without the need to wait for the simulation result. This mitigates the synchronisation overhead since CEs are never required to wait for other CEs to finish their computations, and it mitigates the search overhead since only one simulation is performed per newly expanded node.

While both approaches to leaf parallelisation successfully increase the rate at which an MCTS agent performs playouts, it fails to build a larger tree than a serial agent since the in-tree phases are performed sequentially by a single traverser [62].[3]

[3]This only holds for the case where the traverser is single-threaded. See Section 4.4.1 for a discussion of our multi-threaded traverser implementation.

2.2.4.2 Root Parallelisation

In root parallelisation, multiple searches are performed simultaneously, i.e. the search is parallelised at the root. Initial implementations of root parallelisation had each CE perform an independent search on its own tree until the computational budget is almost expended. At this point, the best move can be chosen by combining the value estimates obtained from the various trees by simply adding them together or using a majority voting scheme [14, 17].

The problem with this approach is that CEs can not take advantage of value estimates obtained by other CEs. This leads to a search overhead incurred by CEs wasting time searching parts of the tree that other CEs may already have determined to hold little value.

In order to address this, Cazenave and Jouandeau [14] proposed periodically sharing statistics between CEs for children of the root node. In this approach, each CE has access to a more stable set of statistics for the children of its root, and the CE is therefore deterred from searching subtrees of the root that other CEs have determined to be unfavourable. This approach to root parallelisation was termed *slow-root parallelisation* by Bourki et al. [8].

While this technique for reducing the search overhead incurred by root parallelisation showed promising results, it was refined by Gelly et. al [25], who proposed periodic sharing of statistics for nodes deeper than the children of the root. This approach, termed *slow-tree parallelisation* by Bourki et al. [8] involves sharing statistics for nodes up to some depth d that have been involved in some percentage p of the total playouts with some frequency f. Bourki et al. [8] reported results where this strategy was effective for $d = 3$, $p = 5\%$ and $f = 3\text{Hz}$.

Similarly to leaf parallelisation, the focus of root parallelisation is to increase the rate at which an MCTS agent performs playouts. However, since the tree constructed by each CE is independent, root parallelisation fails to build a larger tree than a serial implementation [62, 54].

2.2.4.3 Tree Parallelisation

Where root parallelisation and SLMP leaf parallelisation only aim to increase the playout rate, tree parallelisation aims to perform more playouts and build a larger tree by having each CE perform an independent search in parallel on a single shared tree [17]. This means that multiple CEs may attempt to write to the same memory locations simultaneously, and care must be taken to prevent data corruption. In the seminal tree parallelisation paper, Chaslot et al. [17] proposed two solutions to this problem:

- **Global mutexes** lock the whole tree in such a way that only one thread can access the tree at a time. This means that only one CE can be in the selection, expansion or backpropagation phase at a time, while multiple simulations can occur simultaneously from different nodes. Similarly to MLMP leaf parallelisation, the scalability of this approach is limited by a synchronisation overhead incurred by CEs waiting for access to the tree. Unfortunately, most MCTS implementations often spend between 25 and 50 percent of the total search time in these phases [17].

- **Local mutexes** lock a node whenever a CE accesses that node. This means that several CEs can access different nodes simultaneously. Although this technique still incurs a sychronisation overhead when CEs wait for a node to be released, the overhead is less dramatic than in the global mutex approach. However, CEs now frequently have to lock and unlock parts of the tree, which may lead to an additional overhead. Therefore, the use of fast-access mutexes such as *spinlocks* [3] are recommended to take full advantage of available resources [17].

Enzenberger and Müller [23] proposed a lock-free variation of tree parallelisation with better scalability than either local or global mutexes. They found that simultaneous updates to node statistics happen infrequently, and the data corruption caused by this can be safely ignored. While this technique relies on hardware with a specific memory model [23], most modern architectures satisfy these requirements.

While the data corruption caused by simultaneous node statistic updates is negligible, the addition of new nodes to the tree during expansion still requires some protection against corruption. When a CE performs expansion, it initialises the new node fully in a memory array dedicated to the CE. Only once the node is fully initialised is it linked to the parent node. This prevents CEs from attempting to access partially initialised nodes, but can cause a small memory overhead since multiple CEs might each create a new node for a given game state.

An inherent issue with tree parallelisation is that CEs are likely to descend the tree in a very similar fashion. If K CEs select a node n during tree descent and $K - M$ of them find that the node is unfavourable, the remaining M CEs will waste computational resources on playouts that include the unfavourable node n. Additionally, if local mutexes are used, the synchronisation overhead incurred by CEs frequently attempting to access the same nodes is exacerbated.

A heuristic solution to this problem, proposed by Chaslot et al. [17], is to increment the visit count of every node encountered during selection without updating their rewards. This is called a *virtual loss*, and it deters other CEs from selecting the same nodes by artificially decreasing their value estimates when they are selected. The virtual loss is effectively reverted during backpropagation when node rewards are updated (and visit counts are not incremented).

2.3 Akka Actors and Clustering

Our system is implemented in Java and we make use of the Akka toolkit [7] for all message passing and concurrency. Akka leverages the *actor model of computation* [29] to facilitate the development of concurrent, distributed software by treating *actors* as the primitives of concurrency.

Actors are computational entities that communicate through asynchronous message-passing. Upon receiving a message, an actor can:

- modify its local state;
- create more actors;
- send more messages; and/or
- designate a behaviour to be used for future messages

Since Akka actors can only process one message at a time, tasks are performed in parallel by delegating work to more than one actor. Akka actors do not typically share any mutable data, and one actor may only affect another actor's state by sending a message to that actor's `ActorRef`—an object that uniquely represents an actor and may be passed around freely without exposing the actor's internal state to the outside world. In fact, messages that are sent by an actor implicitly contain that actor's `ActorRef`. This model prevents the need for traditional concurrency constructs such as locks and semaphores, thereby simplifying the process of developing concurrent applications. However, immutable state is not explicitly enforced in Akka, and the developers have discussed the use of shared data as an optimisation in Akka applications [2].

Akka actors exist within a hierarchical structure called an *actor system*, and actors may only communicate with other actors in the same actor system, regardless of their relative positions within the hierarchy. When some actor A creates a new actor B, B is added to the actor system hierarchy as a child of A, and A becomes the *supervisor* of B.

A supervisor is responsible for handling any exceptions that are thrown by its children. If a supervisor delegates work to one of its children and that child throws an exception, the child actor suspends itself and all of its descendants and indicates to its supervisor that it has encountered a failure. Depending on the failure, the supervisor may respond in one of the following ways:

- ignore the exception and resume the child;
- restart the child and clear its internal state, including the messages in it's message queue;
- kill the child completely; or
- suspend itself and escalate the exception to its own supervisor.

This implicit supervision hierarchy allows Akka to gracefully handle exceptions, prevent orphaned actors, and cleanly shut down subtrees of the actor hierarchy.

Akka provides distributed computing functionality through the *clustering* module. An Akka cluster consists of a set of actor systems, or *nodes*, possibly running on independent Java Virtual Machines (JVMs) that may span multiple CNs. Nodes are identified by a `hostname:port:uid` tuple, where the UID is a unique identifier for the actor system at the given `hostname:port` pair. The actors in these actor systems may communicate with one another as if they were all in the same actor system on the condition that the relevant actor references are available to them and that they share the same UID. This property is known as *location transparency*. A depiction of a three-node Akka cluster is provided in Figure 2.4.

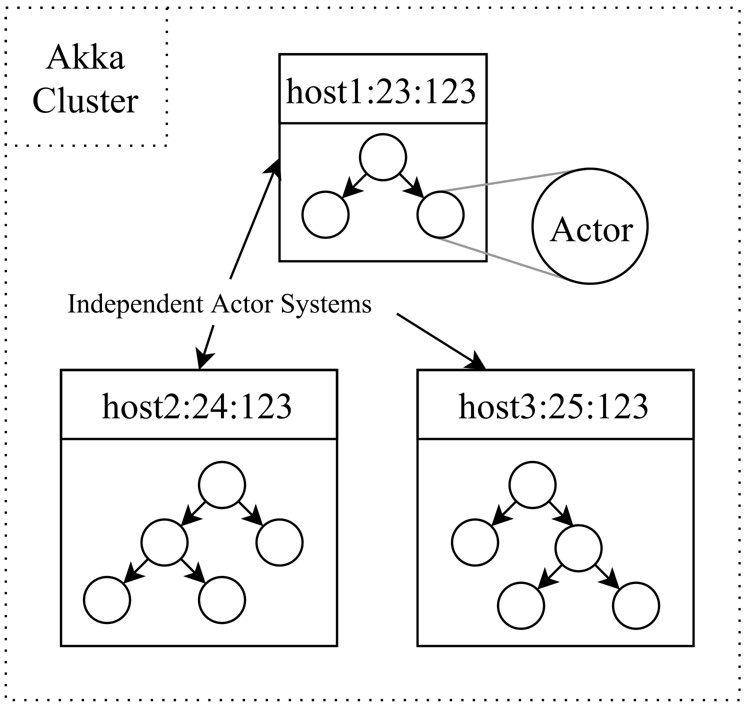

Figure 2.4: An Akka cluster consisting of three nodes. An actor may send a message to an actor in a different node as long as it has access to the recipient's `ActorRef` and the nodes have the same UID.

The formation of a cluster begins by launching one or more *seed nodes*—the initial contact points through which other nodes join the cluster. Additional nodes are added to the cluster by instantiating actor systems with the seed nodes' UID and providing the newly created actor systems with the hostname and port of one or more seed nodes. A newly created node uses this information to send a join command to a seed node, resulting in the node being added to the cluster.

2.4 Lines of Action

We choose to perform our comparison of distributed MCTS algorithms using the board game Lines of Action (LOA). The reason for this choice is the game's highly tactical nature and moderate branching factor (approximately 29) [59].

LOA is a connection-based combinatorial game played on an 8×8 board which initially contains 12 black pieces and 12 white pieces. The initial board layout is shown in Figure 2.5. The rules of LOA [46] are as follows:

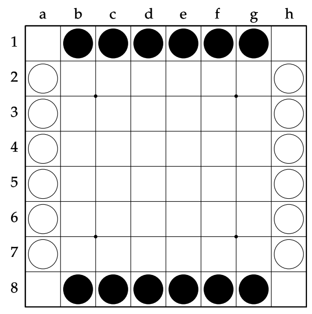
Figure 2.5: The initial board state.

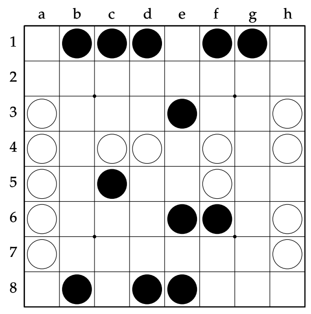
Figure 2.6: f6 may move to **b6**, **d4**, **f2** or **h8**.

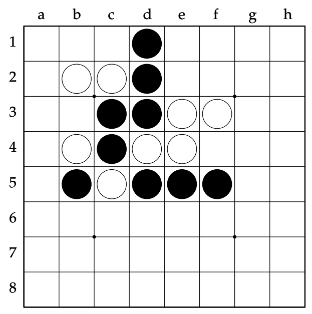
Figure 2.7: An example of a terminal state where black has won.

1. The players alternate moves, starting with Black.
2. The player to move must move one of their pieces in a straight line (horizontal, vertical or diagonal). The number of squares the piece moves is exactly equal to the number of pieces of *any* colour in the line of movement, including the moving piece. An example of this is given in Figure 2.6.
3. A player may jump over their own pieces.
4. A player may not jump over enemy pieces. However, it may land on top of an enemy piece, resulting in that piece's capture (removal from the board).
5. The first player to have all of their pieces in a single, connected component is the winner of the game. The connections between pieces may be either orthogonal or diagonal. The first player to achieve this is the

winner. An example of a terminal state where black has won is given in Figure 2.7.

6. If a move simultaneously causes both Black and White to achieve the winning condition, the game is drawn.

2.4.1 The Quad Heuristic

The rules of LOA dictate that the game is over once a player has moved all of their pieces into a single connected component. The most obvious technique for detecting a terminal state would involve the following:

1. Maintain variables for the number of black pieces (n_b) and white pieces (n_w) on the board, updating them after every move.

2. To check for a terminal state, perform a breadth-first search starting at an arbitrary black piece, and another starting at an arbitrary white piece to determine the sizes (s_b and s_w) of the respective connected components.

3. If $n_b = s_b$ or $n_w = s_w$, the state is terminal.

Since MCTS performs thousands of simulations per second and it is necessary to check for a terminal state after every move in each playout, it would be beneficial to optimise this procedure as much possible.

The *quad heuristic* is a procedure for detecting non-terminal positions in LOA that was first proposed and implemented by Dave Dyer in his LOA program `LoaJava` and subsequently formalised by Mark Winands [60]. The quad heuristic makes use of *quads*—a concept used widely in Optical Character Recognition—to compute the *Euler number* for a player's pieces on the board and use this information to determine connectivity.

The Euler number of a grid represents the number of connected groups in the grid minus the number of holes (an empty square surrounded orthogonally by filled squares) [38]. For example, in the board provided in Figure 2.8, black will have an Euler number of 2 since there are 2 connected black components and no holes. White will have an Euler number of 2 because there are 3 connected white components and one hole at **g6**.

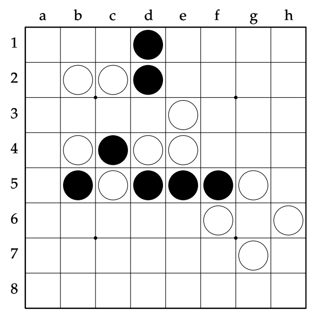

Figure 2.8: A state containing a hole at **g6**.

The quad heuristic is able to detect non-terminal board states by making use of 2×2 quads imposed on the board. Figure 2.9 shows the six possible quad types in LOA, taking rotational equivalence into account. Each quad (except Q_0) contains a *vertex* (the black points), *line segments* (the thick black lines) and *filled regions* (squares occupied by a player's pieces). Vertices and line segments are always adjacent to a filled region. A standard 8×8 LOA board consists of 81 quads, including those which only partially cover the board. Squares containing opponent pieces or those that are not on the board are considered empty.

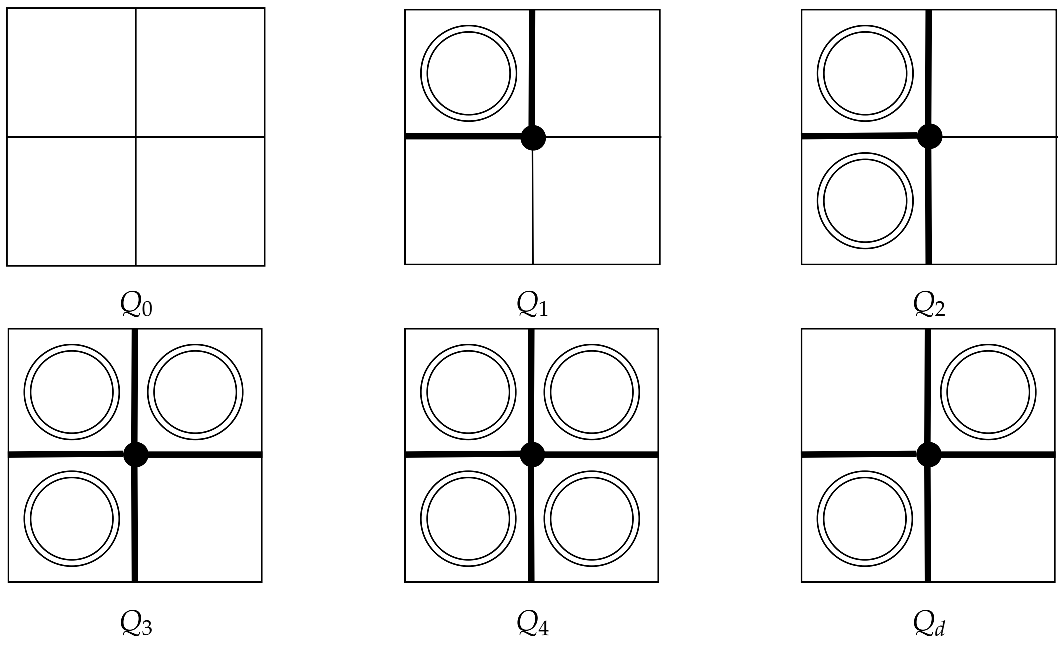

Figure 2.9: Possible quads for a LOA board.

Letting n_v represent the number of vertices on the board, n_l the number of line segments, and n_f the number of filled regions, the Euler number E of the player's pieces on the board can be computed as follows:

$$E = n_v - n_l + n_f.$$

Let Δn_v^q, Δn_l^q and Δn_f^q denote the number of vertices, line segments and filled regions, respectively, for a single quad q. Since each vertex occupies only one quad, each line segment occupies two quads, and each filled region occupies four, the contribution ΔE_q to the Euler number E for the quad is given by:

$$\Delta E_q = \Delta n_v^q - \frac{1}{2}\Delta n_l^q + \frac{1}{4}\Delta n_f^q.$$

Using this, we can determine the contributions to the Euler number for each quad type shown in Figure 2.9. The contributions are as follows:

Type	ΔE
Q_0	0
Q_1	1/4
Q_2	0
Q_3	−1/4
Q_4	0
Q_d	−1/2

The Euler number for the board can then be determined by counting the number of quads of each type and using the following formula:

$$E = \frac{1}{4}\left(\sum Q_1 - \sum Q_3 - 2\sum Q_d\right).$$

If $E > 1$ for both colours, the board is certainly not in a terminal state, since both players must have at least two connected components. However, if $E \leq 1$ for either colour, there can be one connected component or n connected components with $h \geq n - 1$ holes. Therefore, when the Euler number for either colour is less than one, another method (such as the one outlined at the beginning of this section) must be used to determine terminality.

By using the quad heuristic with incremental quad updates (see Section 4.1.2 for a more detailed discussion) in his LOA program `MIA`, Mark Winands found a 10-15% speedup in terminal state evaluation over breadth-first search.

2.5 Summary

In this chapter, we provided the necessary background required to understand the remainder of the thesis. We defined combinatorial games and discussed how the classical combinatorial search algorithms—minimax and $\alpha\beta$ pruning—use game trees to make optimal decisions in these games.

We introduced MCTS, and discussed how the algorithm is able to approximate the values of non-terminal nodes in the game tree by performing simulations in the search space, and defined UCT—the most prevalent selection policy for MCTS which models the selection process as a multi-armed bandit problem.

Following this, we examined three enhancements to standard UCT, defined them as domain-dependent or domain-independent, and discussed the use of transposition tables to efficiently store the search tree without having multiple nodes represent the same game state.

In Section 2.2.4, we introduced three MCTS parallelisation techniques in an environment-agnostic setting: leaf parallelisation, root parallelisation and tree parallelisation. We highlighted that leaf and root parallelisation are easy to implement and increase the playout rate, but do not build larger trees than sequential MCTS in the same amount of time. Tree parallelisation can build a larger tree, but requires careful steps to be taken to prevent data corruption caused by simultaneous memory updates.

We concluded the chapter with a summary of the rules of our test domain—Lines of Action—and introduced the quad heuristic for detecting non-terminal states.

In the following chapter, we will discuss the state of the art for extending leaf, root and tree parallelisation to distributed-memory clusters.

Chapter 3

Related Work

The leaf, root and tree parallelisation techniques discussed in Section 2.2.4 were initially proposed for symmetric multiprocessing (SMP) systems that consist of a single machine with a single processor consisting of two or more identical cores that share the machine's memory. In general, distributed memory clusters provide far greater CPU and memory resources than a single SMP machine, and parallelising MCTS for these environments has the potential for significant performance gains.

A compute cluster consists of multiple machines, or *compute nodes* (CNs) connected to each-other through a local area network (LAN). Each CN in a cluster has one or more processors, each with one or more cores, that share the CNs memory. Network communication is necessary for inter-CN message passing, and mitigating the resulting communication overhead is one of the most significant concerns in distributed MCTS. In light of this, a good distributed MCTS implementation should:

1. Take advantage of increased CPU resources to perform more playouts per second (PPS) when more CNs are available.

2. Take advantage of increased memory resources by building a larger tree when more CNs are available.

3. Produce stronger play when more CNs are available.

In this chapter, we outline the existing literature for applying the parallel MCTS algorithms presented in Section 2.2.4 to distributed memory environments. In order to do this, we begin this chapter with a section on performance measures and scalability (Section 3.1), where we discuss the metrics used to measure the performance of distributed MCTS algorithms. Following this, we discuss the existing work on distributed leaf parallelisation (Section 3.2), distributed root parallelisation (Section 3.3) and distributed tree parallelisation (Section 3.4) in terms of the metrics defined in Section 3.1.

3.1 Performance Measures and Scalability

As discussed in Section 2.2.4, the effectiveness of a parallel algorithm is determined by its scalability. In a distributed setting, a parallel program is said to scale to N CNs if the program performs better with N CNs than $N-1$ CNs. In the literature, the effectiveness of the distributed MCTS algorithms presented in this chapter were measured in terms of either PPS or playing strength (win-rate), or both.

In addition to scalability, some authors present their results in terms of *speedup*: a metric that measures the relative improvement in performance of a distributed algorithm with N CNs over the same algorithm with 1 CN. PPS speedup for an algorithm with N CNs is determined by dividing the PPS achieved by the algorithm with N CNs by the PPS achieved by the algorithm with 1 CN. Although a method to measure speedup in terms of playing strength has been proposed and used in other MCTS literature [17, 55], it was not used as a metric for testing in any of the publications discussed in this chapter.

In summary, scalability measures how many CNs a distributed algorithm can run on with a noticeable improvement over fewer CNs while speedup measures the extent of that improvement.

3.2 Distributed Leaf Parallelisation

The application of leaf parallelisation to distributed memory environments was first proposed by Cazenave and Jouandeau [15] as an improvement on their previous work [14], and was the first occurrence of MLMP leaf parallelisation (see Section 2.2.4) in the literature.

Their implementation consists of a single-threaded traverser running on one CN and up to four simulators running on each of the remaining CNs. At the start of the search, the traverser descends the tree, expands a node, and sends the position associated with the new node to a single simulator until all simulators have received a position from which to perform a simulation. Once a simulator receives a position and completes a simulation, it sends the result of the simulation to the traverser. Upon receiving a simulation result from some simulator S, the traverser backpropagates the result, descends the tree, adds a new node, sends the newly expanded position to S and waits for another result.

Using Go as a test domain, their experimental results show a significant performance increase up to 16 simulators against `GNUGo` [13] version 3.6 on a 9×9 board, with a speedup of 14 with 16 simulators and a win-rate improvement from 40.5% with one simulator to 70.5% with 16 simulators at 5 seconds per move. The scalability of this approach is limited by the communication overhead incurred by the traverser, and they note that better results could be obtained by optimising the traverser.

Another approach to distributed leaf parallelisation was investigated by Kato and Takeuchi [31] with a focus on clusters of inexpensive personal computers and game consoles that may be added or removed from the system at any time. When their traverser expands a node, the position is broadcast to all available simulators and a new playout is immediately started from the root, as opposed to Cazenave's approach which only starts a new playout upon receiving a simulation result.

Their experimental setup consisted of a single CN housing the traverser and 3 simulators, and up to 16 CNs housing four simulators each. Using Go as a test domain, they saw improvements in winning rate against `GNUGo` version 3.7.11 up to 8 CNs on a 9×9 board and up to 16 CNs on a 13×13 board.

3.3 Distributed Root Parallelisation

Extending root parallelisation to distributed memory environments is a relatively simple process, with the seminal work on root parallelisation being tested on a "parallel virtual machine" – a cluster that is simulated on a single machine [14].

Root parallelisation was first applied to a physical compute cluster by Gelly et al. [25]. Their approach was later termed slow-tree parallelisation and extended by Bourki et al. [8].

In slow-tree parallelisation, each CN maintains an independent tree and node statistics are periodically synchronised with frequency f as follows:

1. Select a subtree at each CN consisting of all nodes with depth at most d that were involved in at least p percent of the total playouts at that CN
2. Broadcast the statistics for the nodes selected above to all other CNs.
3. Sum the total rewards and visit counts received for each of these nodes on all CNs. If the node in question does not exist on a CN, it is initialised with the provided statistics.

Gelly's implementation made use of up to 4 cores per CN by applying tree parallelisation with local mutexes (see Section 2.2.4) on each CN. Their approach was only tested on up to 9 CNs, but was shown to scale linearly up to this point on 19×19 Go with $f = 3$Hz and $d = 1$ (only the statistics for children of the root are shared, so it is more similar to the slow-root parallelisation discussed in Section 2.2.4).

Bourki et al. found linear scaling up to 64 single-threaded CNs for $f = 3$Hz, $p = 5\%$ and $d = 3$ on 19×19 Go, and showed that the performance gained when moving from $d = 1$ to $d = 2$ is slight (only up to 2.3%), indicating that $d = 1$ is sufficient in some cases. Their slow-tree parallelisation achieved a 94% win rate against a version of root parallelisation without periodic sharing of node statistics, suggesting that optimal performance is not possible without mitigating the search overhead incurred when statistics are not shared.

Van Niekerk et al. [56] performed additional experiments on distributed root parallelisation, and observed scaling up to 64 single-threaded CNs with $p = 5\%$, $d = 3$ and $f = 5$Hz or $f = 10$Hz.

3.4 Distributed Tree Parallelisation

While root and leaf parallelisation are relatively easy to implement in a distributed memory setting and are successful in increasing playout rate, they do not search deeper given more resources, and therefore fail to take full advantage of a cluster's memory by building a larger tree than sequential MCTS.

Out of the three MCTS parallelisation techniques, tree parallelisation is the only one to build a larger tree than sequential MCTS given the same time constraints. However, tree parallelisation is difficult to implement in a distributed memory environment because sharing the tree among CNs incurs a massive communication overhead [47, 62].

Nevertheless, two approaches for extending tree parallelisation to clusters have been developed: Depth-First UCT (df-UCT) [62] and UCT-Treesplit [47]. Both of these use Transposition Table Driven Scheduling (TDS) [44, 32] to share the tree.

This section begins with an overview of TDS (Section 3.4.1), followed by an outline of df-UCT (Section 3.4.2 and UCT-Treesplit (Section 3.4.3).

3.4.1 Transposition Table Driven Scheduling

Implementing an efficient distributed transposition table (2.2.3.4) is difficult because a CN may not always have local access to the transposition table entries it requires. In the past, the transposition table would be partitioned among the CNs in the cluster, and a CN would perform a remote lookup for any entry that is not present in its local memory by sending a request over the network [24]. Since these requests are synchronous, the time that passes between the request being sent and the transposition table entry becoming available is wasted. Additionally, this approach can lead to thousands of inter-CN messages being passed per second, which introduces a large communication overhead [44].

Transposition table driven scheduling (TDS) was introduced in an attempt to mitigate these issues by integrating the search and the transposition table accesses. Instead of having a CN remotely request the transposition table entry for the node to be searched, the CN sends the node to be searched to the CN that manages its transposition table entry. The CN with the transposition table entry then performs the search. In this way, TDS makes use of a distributed transposition table to share a single tree representation among the CNs in a compute cluster and eliminates the need for synchronous remote lookups [44, 32].

In TDS, each CN manages a disjoint subset of the transposition table and a hash function is used to map each table entry to a single CN (the *home processor* for the entry). If the total size of the transposition table is $|T|$ in a cluster with N CNs, each CN holds approximately $M = \frac{|T|}{N}$ entries. For a compute cluster where each CN has a unique index in the range $0 \ldots N-1$, the home processor index $p(s)$ for a game state s with hash $h(s)$ (see Section 2.2.3.4 for details) is determined using the following formula:

$$p(s) = \left\lfloor \left(\frac{h(s)}{M} \right) \right\rfloor \bmod N \tag{3.1}$$

In TDS, work is moved to the CN where the relevant transposition table entry is located by sending it a job message containing the hash of s. For example, if a CN encounters a game state s, it asynchronously sends a message to the home processor of s so that the transposition table entry can be looked up and used to move the search forward. Since all messaging is asynchronous, a CN can continue processing messages immediately after sending a job message to a remote CN. This mitigates the synchronisation overhead incurred by the work-stealing approach.

The number of inter-CN messages generated in a single playout can be as large as the number of tree nodes that are encountered during the playout. This implies that TDS may perform poorly due to a considerable network communication overhead. However, since message-passing is asynchronous, a CN is able to start working on a new task immediately after sending a message. This allows network latency to be hidden by overlapping communication and computation [44].

In spite of this, TDS suffers from high communication overhead at frequently accessed near-root nodes [32]. Two independently developed techniques to solve this problem have been proposed–df-UCT [62] and UCT-Treesplit [47].

3.4.2 Depth-First UCT

Yoshizoe et al. [62] proposed df-UCT to alleviate some of the overhead incurred by network communication at near-root nodes in TDS. df-UCT is based on the observation that once UCT finds a promising part of the tree, it frequently performs playouts in that part. In order to take advantage of this, df-UCT does not propagate results back to the root node at each iteration. Instead, it delays backpropagation until it can determine that it is no longer on the most promising path.

For each node n encountered during selection, df-UCT determines n's best move m_b and second-best move m_s according to 2.3 and pushes the reward and visit count for each of these moves, as well as n's visit count onto a stack. After expansion and simulation, df-UCT iterates through the stack and updates the visit counts and rewards that are stored in the stack. If, after the update, UCT(m_s) > UCT(m_b) holds for some node n encountered during selection, it means that m_s has become the best move at n, and df-UCT performs backpropagation of the collected results until reaching n, from where selection begins again.

The operation of df-UCT is based on the assumption that the second best move m_s will always become the best move when m_b becomes too weak. This is not always the case, as more inferior moves do sometimes become the best move instead of m_s, but the authors noted that this occurs infrequently, and that investigating the effect of storing more than the best two moves on the stack is a possible avenue for future work [62].

Consider the tree shown in Figure 3.1. Assume that it satisfies UCT(B) > UCT(C), UCT(D) > UCT(E) and UCT(F) > UCT(G) i.e. the left-most child is the best for each node. Now assume that df-UCT has descended the tree and performed a simulation starting at F. It checks its local stack to determine whether it is still on the best possible path or if it must backpropagate simulation results to an ancestor. This process is as follows:

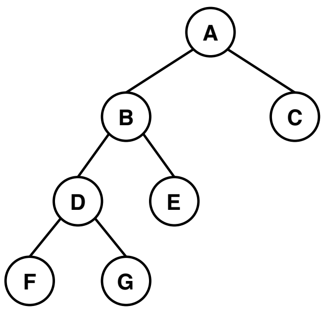

Figure 3.1: A simple tree where the left-most child of every node has the best UCT value

- If UCT(C) > UCT(B), the simulation result is backpropagated to A and C is selected.
- If UCT(E) > UCT(D), the simulation result is backpropagated to B and E is selected.
- If UCT(G) > UCT(F), the simulation result is backpropagated to D and G is selected.
- If none of the above statements hold, df-UCT is still on the best path F is expanded.

By delaying backpropagation in this way, the communication overhead incurred by TDS at the home processor of near-root nodes is mitigated. However, there is also an implicit search overhead introduced by df-UCT: although some parallel searches may determine that a node has little value, others may waste time searching the node since they only have their local stack statistics available to them.

Yoshizoe et al. performed experiments with artificial P-game [53] game trees, which simulate real game trees and support varying branching factors and time per playout. PPS scalability was tested on a cluster of up to 100 CNs with 12 CPU cores per CN and an independent instance of their program running on each CPU core. In order to make sure that the CPU cores are always busy, 20N searches are run simultaneously, where N is the number of CPUs in the cluster.

For all combinations of branching factor $b \in \{8, 40, 150\}$ and time per playout $t \in \{0.1, 1\}$, their implementation scaled up to 100CNs (1200 cores), and vastly outperformed vanilla TDS. df-UCT achieved a speedup of 741.2 with 100CNs while vanilla TDS only managed a speedup of 69.5, indicating that the communication overhead incurred by vanilla TDS is significantly reduced when applying the df-UCT modification.

3.4.3 UCT-Treesplit

Where df-UCT reduces the update frequency of near-root nodes by delaying backpropagation, UCT-Treesplit reduces communication overhead by duplicating frequently visited nodes on all CNs in the cluster so that messages only need to be sent to the home processor of a node when it has not been duplicated [47]. This prevents the home processors of near-root nodes from being overwhelmed with messages.

Similarly to root parallelisation (see Sections 2.2.4 and 3.3), statistics for duplicated nodes are periodically synchronised so that CNs always have reasonably up-to-date statistics for duplicated nodes. In order to facilitate the duplication and synchronisation of tree nodes, UCT-Treesplit maintains an additional reward and visit count (v_i^{Δ} and n_i^{Δ}, respectively) for each node that represent the statistics that are yet to be synchronised. Therefore, the actual accumulated, local reward for child i of a shared node is $v_i^{\Delta} + v_i$ and the actual local number of visits for child i is $n_i^{\Delta} + n_i$.

Four additional parameters are used to assist duplication and synchronisation of near-root nodes:

- n_{dup}: The minimum number of times a node must be visited before it can be considered for duplication
- $n_{\text{sync}}(i) = \alpha(n_i + n_i^{\Delta})$: The minimum value that at least one n_i^{Δ} must have for synchronisation to take place for the parent of i, where α is an adjustable constant.
- $n_{\text{sync}}^{\text{min}}$: A lower bound for $n_{\text{sync}}(i)$.
- $n_{\text{sync}}^{\text{max}}$: An upper bound for $n_{\text{sync}}(i)$.

This means that the statistics for a node are synchronised when there is at least one child i of the node that satisfies the following inequality:

$$n_i^{\Delta} \geq \min\left(n_{\text{sync}}^{\text{max}}, \max\left(n_{\text{sync}}^{\text{min}}, \alpha\left(n_i + n_i^{\Delta}\right)\right)\right) \tag{3.2}$$

The value of α can be chosen to ensure a uniform reduction per synchronisation of the standard error SE_i of a child i's mean estimated reward, which leads to less frequent synchronisation of more settled statistics and more frequent synchronisation at nodes with volatile statistics. Given a reduction rate $p \in (0, 1)$ of the standard error, Schaefers and Platzner derive that $\alpha = (1/p^2) - 1$ [47].

Node duplication and synchronisation are handled by additional CNs referred to as *broadcast ranks*, while *search ranks* are responsible for managing the transposition table and performing the search. The use of additional CNs to handle broadcasting and synchronisation greatly reduces the workload at search ranks—Schaefers and Platzner empirically determined that there should be one broadcast rank for every four search ranks for their domain and hardware setup [47]. Each search rank is assigned a single broadcast rank to send synchronisation messages to, and synchronisation messages originating at other CNs are communicated to the search rank via this broadcast rank.

Upon receiving a synchronisation message from another CN, a broadcast rank does not immediately send this information to its associated search ranks. Instead, the broadcast rank delays sending the synchronisation message for a tree node so that, if it receives additional synchronisation messages for the same tree node, the statistics can be merged before being sent to the search ranks. This significantly reduces the number of synchronisation messages that a search rank receives and processes, thereby improving scalability [47]. Schaefers and Platzner used single-threaded broadcast ranks as a proof of concept but note that the implementation of multi-threaded broadcast ranks is a topic for future research.[4]

Their experiments involved matches against open-source Go programs as well as UCT-Treesplit itself with differing configurations (self-play). Their implementation was able to scale—in terms of playing strength—up to 128 search ranks with 16 CPU cores per CN and 32 broadcast ranks in self-play experiments using Go as a test domain (playing on a 19×19 board against a version with 8 search ranks and 2 broadcast ranks). Experiments

[4]We present our implementation of multi-threaded broadcast ranks in section 4.4.3.3

against the open source programs `Fuego` and `Pachi` showed scaling up to 32 search ranks and 8 broadcast ranks – significantly worse than self-play. The difference in scalability is attributed to the increased likelihood that node statistics accumulated during the opponent's turn can be re-used when the opponent performs a similar search [47].

3.5 Summary

In this chapter, we presented the state of the art for distributed MCTS. We discussed the existing techniques for extending leaf, root and tree parallelisation to distributed memory environments. We noted that a good distributed MCTS algorithm should make effective use of the increased compute resources in a cluster by increasing the playout rate, building a larger tree, and mitigating the communication overhead incurred by inter-CN message passing.

We noted that the distributed leaf parallelisation agents developed by Cazenave and Jouandeau [15] and Kato and Takeuchi [31] each scaled to 16 simulators, with Cazenave and Jouandeau stating that the scalability of their approach was limited by a single-threaded traverser.

We discussed three studies of distributed root parallelisation and showed that root parallelisation scales better than distributed leaf parallelisation in all cases, with two of the analyses showing scalability up to 64 CNs.

The remainder of the chapter discussed distributed tree parallelisation. While distributed root parallelisation shows impressive scalability, it fails to build a larger tree than serial MCTS because CNs tend to expand the same tree nodes. TDS was introduced as a technique to efficiently share a search tree among CNs so that a larger tree can be built with an increase in CNs, but it was shown to suffer from network communication overhead incurred at near-root nodes.

df-UCT was the first technique proposed to limit the impact of this communication overhead, and proved to be an effective modification to vanilla TDS. Yoshizoe et al. showed that df-UCT managed a PPS speedup of 741.2 with 100CNs when applied to artificial game trees while vanilla TDS only managed a speedup of 69.5.

The second technique to mitigate the communication overhead incurred by TDS—UCT-Treesplit—showed playing strength scalability up to 128 search ranks and 32 single-threaded broadcast ranks, making this approach the best distributed MCTS algorithm developed so far in terms of scalability.

In the following chapter, we will discuss our implementation of these distributed MCTS algorithms, as well as some modified versions, in preparation to present our experimental findings in Chapter 5, where we will compare our implementations on a common domain.

Chapter 4

Design and Implementation

In this chapter, we discuss the design decisions and implementation details of our test domain, Lines of Action, and each of our distributed MCTS agents. Section 4.1 presents the data structures and algorithms we implement for move generation, terminal state detection and incorporation of knowledge in Lines of Action. In Section 4.2, we discuss our MCTS tree node structure and transposition table implementation. In Section 4.3, we provide background on the tools we use to facilitate distributed computing and a high-level overview of our test framework. The implementation of each of our distributed MCTS agents is detailed in Section 4.4.

4.1 Lines of Action

In this section, we discuss the implementation of our test domain, Lines of Action. Our MCTS implementations require access to a data structure that represents a LOA game state and allows for efficient move generation and terminal state detection. We implement a `LOABoard` class that achieves this through the use of data structures that are incrementally updated whenever a move is applied to the board.

Since our distributed MCTS agents are able to take advantage of domain knowledge through progressive unpruning and progressive bias (see Sections 2.2.3.2 and 2.2.3.3, respectively), we implement a static evaluation function and move categories to facilitate these enhancements.

In Sections 4.1.1 and 4.1.2, we discuss the data structures we implement to efficiently generate legal moves for a given `LOABoard` instance and detect terminal board positions, respectively. In Section 4.1.3, we discuss our evaluation function implementation and the techniques we use to determine move categories and their values.

4.1.1 Move Generation

In MCTS, node expansion (Section 2.2.1) generally entails generating a list of legal moves for the game state associated with the node. Additionally, in the simulation phase (Section 2.2.1), a random move must be generated for every game state encountered in the simulation. Since strong MCTS engines typically expand thousands of nodes per second and every expansion is followed by a simulation, it is important to implement data structures and algorithms to facilitate efficient move generation.

In LOA, move generation is not a straightforward process, since pieces may move in any direction and the distance a piece may move in a given direction is dependent on the number of pieces in the line of motion.

Our approach to move generation is adapted from the LOA program *MIA* [58], and is built on the idea that the legality of a move along some line is independent of where the line is positioned on the board. For example, in Figure 4.1, the configuration of rank *b* and rank *d* are identical. Since b2b6 is a legal move for black, d2d6 is implicitly legal as well.

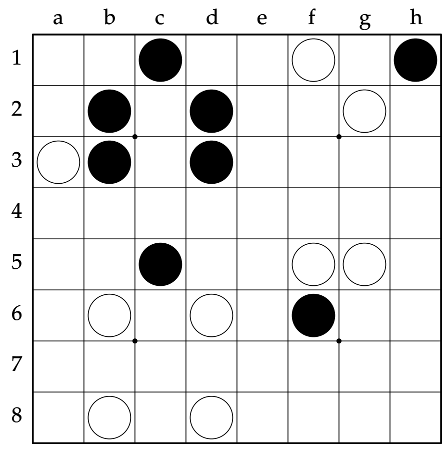

Figure 4.1: Since the move b2b6 is legal and rank d has the same configuration as rank b, it is implied that d2d6 is legal as well.

We take advantage of this property by populating a `MoveMap` at program initialisation that we use to retrieve legal moves throughout the search. Once populated, the `MoveMap` consists of a four-dimensional array where each entry holds a list of legal moves for a particular line on the board, from the perspective of a particular colour.

The properties we use to index into the `MoveMap` are as follows:

1. An integer representing the colour of the player to move.
2. An integer representing the orientation of the line for which legal moves should be retrieved (horizontal, vertical, positive-gradient diagonal or negative-gradient diagonal).
3. An integer representing the position of the line on the board.
4. An integer encoding of the configuration of the pieces on the line. We map the configuration of a line of length n to an integer hash as follows:

$$\text{encoding} = \sum_{i=0}^{n-1} (p_i \times 3^i) \tag{4.1}$$

where i is the position of the piece on the line as depicted in Figure 4.2 and p_i is the integer representation of the colour at i (0 for an empty square, 1 for black and 2 for white).

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4
5	5	5	5	5	5	5	5
6	6	6	6	6	6	6	6
7	7	7	7	7	7	7	7

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	6
0	1	2	3	4	5	5	5
0	1	2	3	4	4	4	4
0	1	2	3	3	3	3	3
0	1	2	2	2	2	2	2
0	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0

7	6	5	4	3	2	1	0
6	6	5	4	3	2	1	0
5	5	5	4	3	2	1	0
4	4	4	4	3	2	1	0
3	3	3	3	3	2	1	0
2	2	2	2	2	2	1	0
1	1	1	1	1	1	1	0
0	0	0	0	0	0	0	0

Figure 4.2: Line decompositions for horizontal, vertical and diagonal line orientations.

The encodings for each line on the board are maintained by the `LOABoard` and updated every time a move is applied to the board. This requires us to update a maximum of 8 line encodings per move, which is a significant improvement over the naïve approach of recomputing every line's encoding whenever a move is made. Additionally, updating a line encoding when a piece is added or removed does not require a full re-computation of the encoding as shown in Equation 4.1 since each piece in each position is represented by an individual term that can be added or subtracted from the encoding.

Although the `MoveMap` contains all possible legal moves in LOA, we do not need to explicitly compute a list of legal moves for every possible game state. Instead, we compute the legal moves for each possible line configuration and insert these moves into the appropriate `MoveMap` entries.

For example, once we have computed the two legal white moves for the line configuration shown in Figure 4.3, we can insert them into the `MoveMap` entries for each horizontal and vertical line position, as well as the two main diagonals.

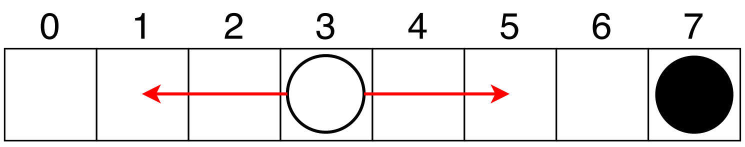

Figure 4.3: Given this line configuration, white may move two places in either direction along the line.

Then, if we were to encounter this line configuration on rank c, we would retrieve the list of legal moves for white along the rank using the following statement:

Listing 4.1: Retrieval of legal moves for the line configuration in Figure 4.3 at rank c

```
List<Move> movesForLine = moveMap
            [Colour.WHITE]
            [Orientation.HORIZONTAL]
            [Rank.C]
            [line.getIntEncoding()];
```

4.1.1.1 Generating All Legal Moves

The `MoveMap` allows us to retrieve a list of legal moves for a single line on the board in constant time. However, in order to populate a list of all legal moves for a given game state, we must consider all lines that intersect at least one of the moving player's pieces. Therefore, the `LOABoard` maintains a set of co-ordinates for each colour that indicates where its pieces are located.

In order to generate all legal moves for a colour, we concatenate the legal moves for every line that intersects one of the colour's pieces. An abbreviated version of this procedure is provided in Listing 4.2. Line 6 ensures that we do not add duplicate moves if the same line is encountered more than once and lines 7-10 retrieve the legal moves for the current line from the `MoveMap`.

Listing 4.2: A function that returns a full list of legal moves for a game state

```
1  List<Move> getAllLegalMoves(){
2    List<Move> allLegalMoves = new ArrayList<Move>();
3    Set<Line> linesAlreadyChecked = new HashSet<Line>;
4    for (Coord pieceCoord : getCurrentPlayerPieces()){
5      for (Line line : getIntersectingLines(pieceCoord)){
6        if (linesAlreadyChecked.add(line)) {
7            List<Move> movesForLine = moveMap
8                                  [getCurrentPlayer()]
9                                  [line.getOrientation()]
10                                 [line.getPosition()]
11                                 [line.getIntEncoding()];
12           allLegalMoves.addAll(movesForLine);
13       }
14     }
15   }
16   return allLegalMoves;
17 }
```

4.1.1.2 Generating A Random Legal Move

During the simulation phase of MCTS, our agents apply a random legal move to the board until a terminal position is reached. This type of move generation occurs far more frequently than the one described in Section 4.1.1.1, so it is important that it is performed as efficiently as possible.

One approach to generating a single legal move for a given game state is to generate all legal moves as described in Section 4.1.1.1 and select a random move from the resulting list. This would be time-consuming due to list concatenation and iteration through piece sets.

In order to reduce the time spent generating random moves, we first attempt to select a random piece from the piece set, retrieve the legal moves for that piece, and select a random one. If there are no legal moves for the piece, we repeat this process. If this approach fails three times we use full move generation, although this occurs so infrequently that the resulting performance degradation is negligible.

4.1.2 Terminal State Detection

Whenever a move is applied to a game state during the search, an MCTS agent must determine whether the resulting state is terminal. This can occur hundreds of times in a single simulation, so it is important that the agent has an efficient mechanism for determining when the game is over.

As discussed in Section 2.4.1, the quad heuristic is an efficient technique for identifying non-terminal LOA game states. We leverage the quad heuristic by implementing a `QuadTable` class that can be used to determine a colour's Euler number. The `LOABoard` stores a `QuadTable` for each player and updates them every time a move is applied to the board.

A `QuadTable` consists of two 2-dimensional arrays of quads. A quad is represented as a 4-bit number where a bit is on if and only if the corresponding corner of the quad is occupied by one of the player's pieces. Figure 4.4 depicts each bit's position in the quad for the bit-string $b_0b_1b_2b_3$.

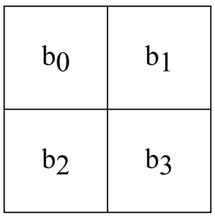

Figure 4.4: The quad represented by the bit-string $b_0b_1b_2b_3$

When a move is applied to the board, both `QuadTables` must be updated to reflect the change in state. When a piece is added to or removed from a quad at corner b_x, the quad is xored with a mask that has b_x on and all other bits off. This process is depicted in Figure 4.5.

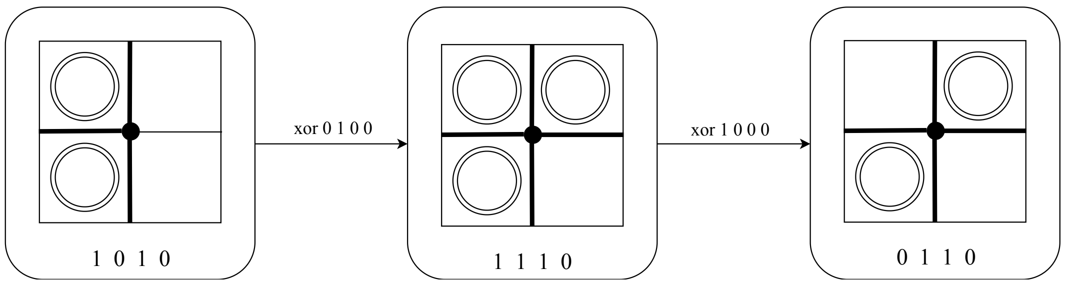

Figure 4.5: A quad being updated by xoring in a piece at b_1, then xoring out a piece at b_0.

By incrementally updating each colour's `QuadTable` in this way, the `LOABoard` is able to use Euler numbers to determine if the game state is non-terminal in most cases, as described in Section 2.4.1. If the quad heuristic is inconclusive, we use the depth-first search technique presented in Section 2.4.1 to determine terminality.

4.1.3 Incorporation of Knowledge

The progressive bias and progressive unpruning MCTS enhancements function by attaching values to moves before they are explored by the agent. These values are determined by incorporating domain knowledge—pre-existing knowledge of the test domain that is supplied to the agent before program execution.

In this section, we describe the two forms of heuristic knowledge we incorporate: a static evaluation function and move categories.

4.1.3.1 Evaluation Function

The purpose of an evaluation function is to assign a heuristic value to a game state from the perspective of the player to move. Since the progressive strategies we implement must attach values to moves rather than game states, we determine the value of a move by applying the move and evaluating the resulting state from the perspective of the player that applied the move.

We implement a simple, non-incremental evaluation function that makes use of the concentration component of Mark Winands' LOA evaluation function [61]. Our evaluation function computes the value of a move m from the perspective of a player p by performing the following steps:

1. Apply m to the game state.
2. Compute the centre of mass for each player's pieces.
3. For each colour, compute the distance of every piece from the centre of mass for that colour and sum the distances. This is referred to as the sum-of-distances (SOD).
4. For each number of pieces N, there is a value we call the sum-of-minimal-distances (SOMD(N)) that denotes the smallest possible SOD when there are N pieces on the board. We retrieve the SOMD(N) for both players from a pre-populated table.
5. Determine the excess-of-distances (EOD = SOD – SOMD(N)) for both players.
6. Determine each player's concentration—the reciprocal of its EOD.
7. Subtract the concentration of the opponent's pieces from the concentration of p's pieces to determine the value of m from the perspective of p.

4.1.3.2 Move Categories

The move category approach to incorporating domain knowledge assigns values to moves by partitioning them into pre-determined categories and using a database of expert matches to determine a *transition probability* for each category. The transition probability $P(c)$ of a move category c is given by the following formula [59]:

$$P(c) = \frac{n_{\text{played}}(c)}{n_{\text{available}}(c)}$$

where $n_{\text{played}}(c)$ is the number of game positions in the database where a move belonging to category c was played, and $n_{\text{available}}(c)$ is the number of positions in the database where a move belonging to category *c* could legally be played.

We partition the board into the four regions depicted in Figure 4.6: the four corners of the board (region A), the remainder of the 8x8 outer rim (region B), the borders of the 4x4 and 6x6 inner squares (region C) and the 2x2 inner square (region D). Moves are categorised according to the regions that their source and destination co-ordinates occupy. Moves are further classified into capture and non-capture moves. This results in a total of 32 possible move categories, and we assign a transition probability to each of them by parsing match logs that were obtained from Darse Billings' LOA page [6]. These transition probabilities can be found in Appendix A.

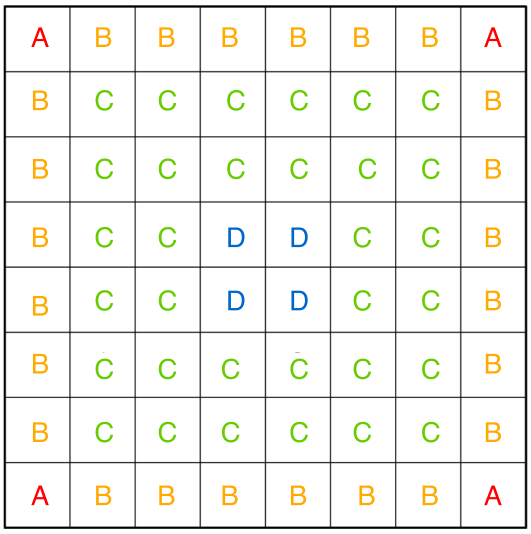

Figure 4.6: Board regions used for move classification.

4.2 Game Tree Representation

As described in Section 2.2.2, MCTS constructs and traverses a game tree with nodes representing game states and outgoing edges representing possible moves from a state. When an agent visits a node during tree descent, it must have access to the total reward and number of simulations accumulated for every move from the node so that the UCT formula can be applied.

All of our implementations use a transposition table to store the game tree. Each entry in the transposition table is a single tree node and contains the statistics necessary to apply the UCT formula. We discuss the details of our tree node and transposition table implementations in Section 4.2.1 and Section 4.2.2, respectively.

4.2.1 Tree Node

We implement a `TreeNode` interface to define the necessary operations that are performed on a node during the search. Since some of our implementations have multiple threads that simultaneously modify a shared tree, we implement a `VolatileTreeNode` that synchronises access to node statistics for these agents while other agents make use of a `SimpleTreeNode`. This prevents agents that do not require volatile statistics from incurring an overhead through frequent memory accesses.

Since UCT-Treesplit and root parallelisation frequently synchronise node statistics, they must maintain those statistics that have been communicated to remote ranks and those that have not in separate data structures. Instead of adding more `TreeNode` implementations to facilitate this, we separate synchronised and un-synchronised statistics for all agents. Agents that do not perform node synchronisation simply do not instantiate the secondary data structures.

Our `TreeNode` implementations consist of the following fields:

1. `nodeHash`: A Zobrist hash (Section 2.2.3.4) that identifies the game state associated with the node.

2. `allMoves`: An array consisting of all moves that may be legally made from the game state associated with the `TreeNode`.

3. `nodeVisits`: An integer that keeps track of the total number of times the node has been visited.

4. `unsynchronisedVisits`: An array that holds the number of visits that must still be communicated to remote CNs for every move in `allMoves`.

5. `synchronisedVisits`: An array that holds the number of visits that have already been synchronized for each move in `allMoves`.

6. `unsynchronisedRewards`: An array that holds the total accumulated reward that must still be communicated to remote CNs for each move in `allMoves`.

7. `synchronisedRewards`: An array that holds the total accumulated reward that has already been synchronised for each move in `allMoves`.

8. `knowledgeValues`: An array that holds the knowledge value for each move in `allMoves`. This may either be a transition probability as discussed in Section 4.1.3.2, or the output of our static evaluation function as discussed in Section 4.1.3.1.

Both `TreeNode` constructors take a `LOABoard` parameter that represents the game state associated with the `TreeNode`. This parameter is used to populate `allMoves` and `knowledgeValues` by employing the procedures discussed in Section 4.1.1 and Section 4.1.3, respectively. The remaining move statistic arrays are filled with zeroes at initialisation.

During node initialisation, we order the list of moves in descending order of knowledge value. Moves with the same knowledge value are ordered according to their positions in the `MoveMap` (Section 4.1.1) so that the ordering of `allMoves` is deterministic. This ordering policy has the following two purposes:

1. If progressive unpruning is used and the unpruning window is N, only the first N moves need to be considered during tree descent since they are implicitly the moves with the best knowledge values.

2. For implementations that require sharing of node statistics, we can be sure that the order of a node's children is the same on every CN, thereby simplifying the synchronisation procedure.

4.2.2 Transposition Table

All of our agents make use of a transposition table to store the game tree representation. The transposition table contains a one-dimensional array with each entry mapping to a single `TreeNode`.

Our transposition table implementation is derived from the one devised for UCT-Treesplit [47], and allows for automatic replacement of stale nodes during the search. This prevents the need for an explicit element deletion phase between turns.

The transposition table maintains an integer variable `lookupCount` that is incremented with every table lookup. Additionally, we store the value of `lookupCount` at the start of the current turn in an integer variable `start-Time`. At transposition table initialisation, the array of entries is filled with `TTEntry` objects that each contain a `TreeNode` reference and an integer variable `lastAccess` that stores the value of `lookupCount` when the `TreeNode` was last accessed. Before the search begins, all `TTEntry` objects have a null `TreeNode` and a `lastAccess` value of 1, indicating an empty index.

When an agent encounters a game state s, it performs a transposition table lookup that either finds and returns the `TreeNode` associated with s, or creates a new `TreeNode` for s and inserts it into an appropriate table index. This lookup procedure is divided into two phases: finding the best index for insertion/retrieval and updating the transposition table accordingly. We discuss the two phases in Section 4.2.2.1 and Section 4.2.2.2, respectively.

4.2.2.1 Best Index Retrieval

We use Zobrist hashing (Section 2.2.3.4) to map a game state to a 64-bit integer that is used to compute an index into the transposition table for the given game state. The size of our table is always a power of two so that table indices can be determined using a bitwise `and` operation instead of `modulo`. For example, to compute the table index for a game state s, we use the following formula:

$$\text{index}(s) = \text{hash}(s)\ \&\ (\text{ttsize} - 1)$$

The transposition table can not be made large enough to accommodate every possible LOA game state. This means that multiple game states may map to the same table index. Our table lookup operation uses linear probing to resolve such collisions. When performing a lookup for a game state s, we iterate through the table entries in the index range $[\text{index}(s), \text{index}(s) + K - 1]$ to either find the table index for the `TreeNode` associated with s or a viable index to insert a new `TreeNode` for s. In the original literature, K is set to 10. [47] We choose to use the same value so that we can conserve the computational resources required to tune this parameter.

If we find that the transposition table does not have an entry for s, we find the first unoccupied table index in the range $[\text{index}(s), \text{index}(s) + K - 1]$. If there is no free index, we use the index of the oldest element (the entry with the smallest `lastAccess` value) as long as it has not been accessed during the current turn (`lastAccess` < `startTime`). The reason we choose to overwrite the least recently accessed element is because it is more likely that more recently accessed nodes will be encountered in the future. If all entries in the range are occupied by a recently used node, the table is considered full, the node is not inserted into the table, and the search moves on to the simulation phase of MCTS.

The procedure we use to find the table index for a node with a given Zobrist hash is provided in Listing 4.3. It returns an `IndexProbeResult`—a pair containing the best index for insertion and the index of the entry with the given Zobrist hash, if one exists. This is used in the second phase of the lookup that we discuss in Section 4.2.2.2.

4.2.2.2 Table Update

After computing an `IndexProbeResult` for a game state s, the lookup procedure either adds a `TreeNode` to the transposition table or updates the `lastAccess` field of the `TTEntry` for s. The procedure returns a `LookupResult` that contains the index of the `TTEntry` for s if the lookup is successful as well as one of the following return states:

- `FOUND`: Indicates that the transposition table contains an entry for s at the provided index.
- `FULL`: Returned if `findIndex` was unable to find an entry for s, an empty table entry, or an entry old enough to be over-written.
- `BUSY`: Indicates that the entry has been updated by another thread.

Listing 4.3: A function to find the table index for a node with the given Zobrist hash, where K is a program parameter that determines the number of entries that are searched and `startTime` is the value of `lookupCnt` at the start of the current turn.

```
IndexProbeResult findIndex(long hash){
  int index = (int)(hash & (ttsize - 1));
  int bestFitIndex = 0;
  int foundIndex = 0;
  int oldestElement = Integer.MAX_VALUE;
    for (int i = 0; i < K; i++){
    // 0 is used as an implicit lock value
    index = Math.max(index, 1);
    TTEntry entry = entries[index];
    int lastAccess = entry.getLastAccess();
    if ((lastAccess == 1) && (oldestElement > 0)){
      // First free index
      bestFitIndex = index;
      oldestElement = 0;
    } else if ((lastAccess < startTime) &&
      (lastAccess < oldestElement)){
      // Old element
      bestFitIndex = index;
      oldestElement = lastAccess;
    }
    if ((entry.getHash() == hash) &&
    (lastAccess > 1)) {
      // Found node
      foundIndex = index;
      break;
    }
    index = (index + 1) & (ttsize - 1);
  }
  return new IndexProbeResult(bestFitIndex,
                               foundIndex);
}
```

Our lookup procedure is provided in Listing 4.4 and can be broken down as follows:

1. Lines 2-5: Call `findIndex` to determine the index of the `TTEntry` for the given Zobrist hash. If the `IndexProbeResult` has `foundIndex` == 0 and `bestFitIndex` == 0, we return `FULL`. If `foundIndex` != 0, we set `index` = `foundIndex`; Otherwise, we set `index` = `bestFitIndex`.

2. Lines 8-26: A loop that is executed when the entry at `index` is stale or empty. The loop repeats until the entry is successfully updated. If an entry with the given hash already exists, its `lastAccess` field is updated in line 17 with an atomic compare-and-swap operation to prevent corruption caused by simultaneous updates. If a new node is to be added to the `TTEntry` at `index`, we set its `lastAccess` to 0—preventing other threads from updating the same `TTEntry`—add a new `TreeNode` to the entry, and finally set its `lastAccess` to the current value of `lookupCnt`.

3. Lines 27-32: If the entry at `index` is not stale and points to a `TreeNode` with the correct hash, return the entry's index and `FOUND`. If not, return `BUSY` as the entry has been updated by another thread.

Listing 4.4: Our transposition table lookup procedure.

```java
LookupResult lookup(long hash, Board board){
  IndexProbeResult ipr = findIndex(hash);
  int index = ipr.foundIndex;
  if (index == 0) index = ipr.bestFitIndex;
  if (index == 0) return new LookupResult(0, FULL);
  incrementLookupCount();
  int lastAccess = entries[index].getLastAccess();
  while (lastAccess <= startTime){
    if (lastAccess == 0){
      // 0 is used as an implicit lock value
      lastAccess = entries[index].getLastAccess();
      continue;
    }
    if (ipr.foundIndex != 0){
      // An entry for this hash exists.
      // Update the entry's last access.
      entries[index]
          .CASLastAccess(lastAccess, getLookupCount());
    } else if (entries[index]
                    .CASLastAccess(lastAccess, 0)){
      // Set last access to 0 to prevent other threads
      // from writing to the index and insert a new
      // node
      TreeNode node = makeNode(board, hash);
      entries[index].setNode(node);
      entries[index].setLastAccess(getLookupCount());
    }
    lastAccess = entries[index].getLastAccess();
  }
  if (entries[index].getLastAccess() > startTime &&
      entries[index].getHash() == hash){
    entries[index].setLastAccess(getLookupCount());
    return new LookupResult(index, FOUND);
  }
  return new LookupResult(0, BUSY);
}
```

4.3 System Overview

Our system is implemented in Java and we make use of the Akka framework (see Section 2.3) for all inter-CN message passing and concurrency. In this section, we provide design and implementation details for our test framework. Additionally, we describe and depict the sequence of events that make up a match within the framework.

4.3.1 Cluster Design

We make use of Akka clustering (see Section 2.3) to implement a test framework that runs matches between two distributed LOA agents on a compute cluster with $N \geq 3$ CNs and M CPU cores per CN. Each agent a consists of an Akka cluster with one seed node and $N_a \geq 1$ member nodes—one for every CN available to the agent—that are used to perform the search.

At a high-level, our test framework consists of five components: `Referee`, `ClusterManager`, `ClusterMember`, `RankManager` and `Worker`. While the `Referee` is a simple Java class and the `ClusterMember` class is general to all our implementations, the other three components are abstract Akka actors whose implementations are specific to each of our distributed MCTS agents. This section does not concern itself with the details of every implementation, but rather focuses on the high-level responsibilities of each component and the interactions between them.

The structure of our test framework is depicted in Figure 4.7, the cluster initialisation procedure is presented in Section 4.3.2, and the individual responsibilities of the aforementioned components are as follows:

- `Referee`: A non-actor Java class that is responsible for initialising a seed node and `ClusterManager` for each agent and running a match by requesting moves from them in an alternating fashion until the match is over.

- `ClusterMember`: Each agent a has N_a `ClusterMember`s that are launched before the match starts. A single `ClusterMember` is executed on each of the CNs available to the agent. Before the `ClusterMember` actor is instantiated, this class is responsible for creating an actor system that automatically joins the agent's seed node. Once the actor system is created, a `ClusterMember` actor is added to the system and used to instantiate a specific `RankManager` implementation based on an initialisation message that it receives from the `ClusterManager`.

- `ClusterManager`: The sole actor in the seed node for an agent's cluster. Responsible for sending initialisation messages to `ClusterMembers` when they join the cluster, facilitating communication with the `Referee`, and shutting down the cluster upon match completion. Both `ClusterManagers` are instantiated by the `Referee`, and these three components run on a single CN. The types of initialisation messages that are sent by the `ClusterManager` depend on the type of agent being tested, and each of our distributed MCTS agents has a unique `ClusterManager` implementation. A detailed discussion of each implementation is provided in Section 4.4.

- `RankManager`: An actor that instantiates and manages the `Workers` on a CN. Different distributed MCTS implementations require their `RankManagers` to perform specific tasks. For example, leaf parallelisation requires one CN to manage the tree and $N_a - 1$ CNs to perform simulations while UCT-Treesplit requires some search ranks and some broadcast ranks. Therefore, every `ClusterMember` is assigned a role by the `ClusterManager` once it has successfully joined, and this role determines the type of `RankManager` that is instantiated on the CN. A detailed discussion of our `RankManager` implementations is provided in Section 4.4.

- `Worker`: The actors that perform the majority of an agent's work. The number and types of `Workers` that are instantiated by the `RankManager` depend on the number of CPU cores available on the CN, as well as the member's role. The details of each `Worker` implementation are discussed in Section 4.4.

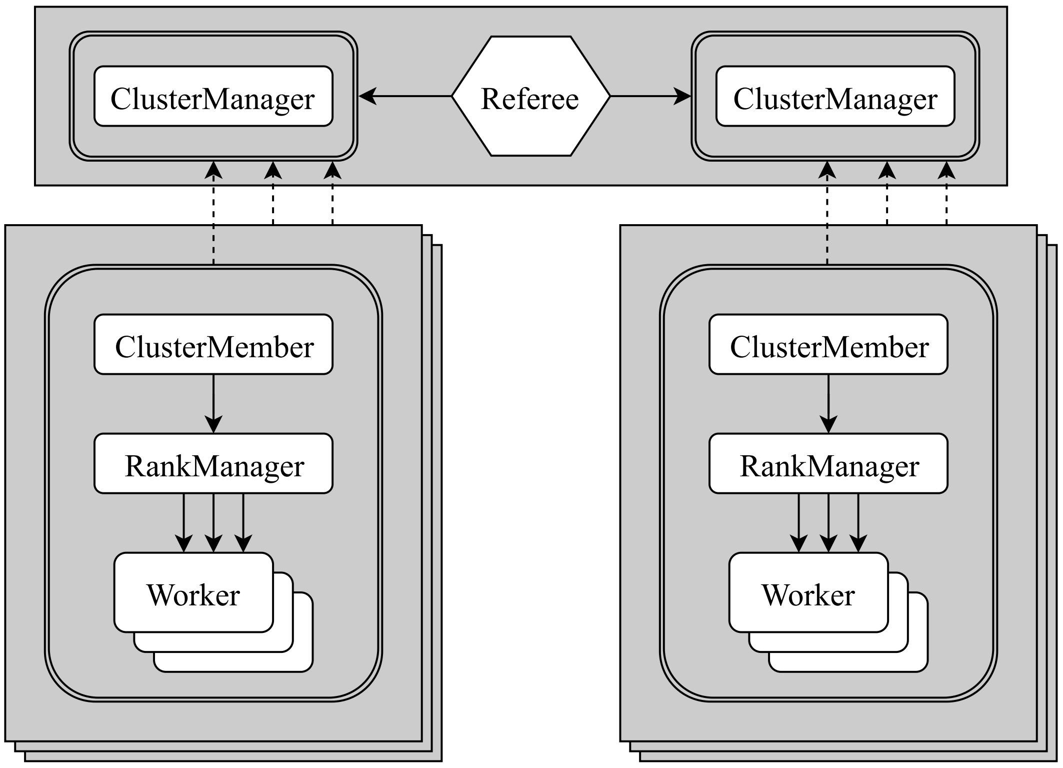

Figure 4.7: A high-level overview of our test framework. Dark rectangles are CNs, rounded single rectangles depict actors, rounded double rectangles are actor systems, and a hexagon represents a non-actor Java class. Solid arrows indicate instantiation and dotted arrows represent cluster join commands from remote CNs.

4.3.2 Cluster Initialisation and Match Execution

The process of running a match in our test framework begins by launching a `Referee` on one CN and a `ClusterMember` on each of the remaining CNs. The hostname, port and UID to be used for each agent's seed node is provided to the `Referee` as a program argument. When running a match between agents a and b, we provide N_a `ClusterMembers` with the hostname, port and UID of a's seed node, and the remaining N_b `ClusterMembers` are launched with the hostname, port and UID of b's seed node.

Once all `ClusterMembers` and the `Referee` have been launched, each agent's Akka cluster must be initialised before the match begins. We implement the following four message types to facilitate cluster initialisation:

- `ClusterInitRequestMessage`: An empty message, sent by the `Ref-`

`eree`, that a `ClusterManager` must respond to with an `InitCompleteMessage` when all `ClusterMembers` have joined the cluster.

- `ClusterMemberRegistrationMessage`: An empty message that a `ClusterMember` sends to the `ClusterManager` so that it has access to the member's `ActorRef`.
- `ClusterMemberInitMessage`: An interface whose implementation determines the type of `RankManager` that a `ClusterMember` creates, as well as any other information that is required by that `RankManager`. The specifics for each implementation of this interface are discussed in Section 4.4.
- `InitCompleteMessage`: An empty message signalling that a component has been fully initialised.

The initialisation process for a single agent is depicted as a sequence diagram in Figure 4.8 and proceeds as follows:

1. The `Referee` creates an actor system to be used as the agent's seed node and instantiates a `ClusterManager` within the newly created actor system. The type of `ClusterManager` that the `Referee` creates depends on the type of agent being tested, and is provided as a program argument.
2. Each `ClusterMember` executable creates an actor system, joins the seed node, and instantiates a `ClusterMember` actor which sends a `ClusterMemberRegistrationMessage` to the `ClusterManager`.
3. Upon receiving a `ClusterMemberRegistrationMessage`, the `ClusterManager` replies with an implementation-specific `ClusterMemberInitMessage`.
4. Depending on the type of `ClusterMemberInitMessage` it receives, a `ClusterMember` instantiates a `RankManager` implementation with the arguments provided in the `ClusterMemberInitMessage`.
5. The `RankManager` instantiates its `Workers`. The number of `Workers` that are created depends on the number of CPU cores available at each CN and the type of the `RankManager`.

6. Each `Worker` sends an `InitCompleteMessage` to its `RankManager` once it is fully initialised. When all `Workers` are initialised, the `RankManager` sends an `InitCompleteMessage` to the `ClusterMember`, which forwards it to the `ClusterManager`.

7. Once the `ClusterManager` has received `InitCompleteMessages` from all `ClusterMembers`, it sends an `InitCompleteMessage` to the `Referee`, signalling that the agent is ready for the match.

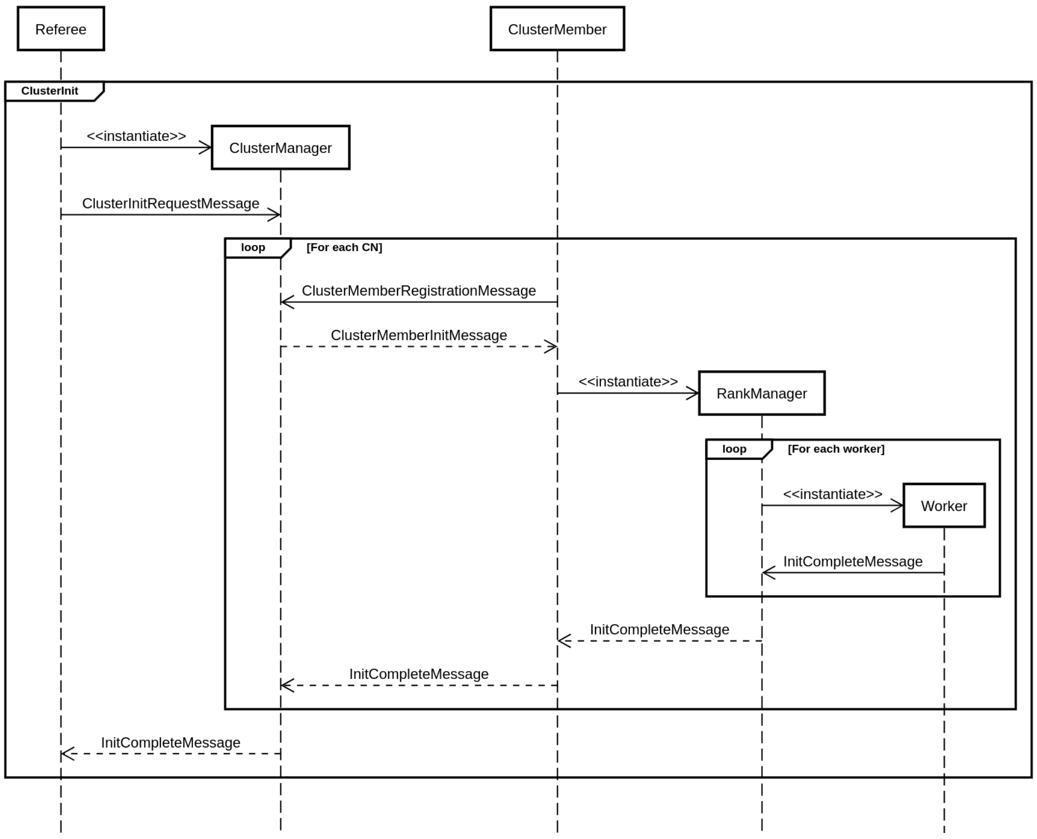

Figure 4.8: A sequence diagram depicting the initialisation process for an agent's cluster.

Once the `Referee` has received an `InitCompleteMessage` from both `ClusterManagers`, the match begins. We supply the referee with an initialisation timeout parameter that determines how long it waits for an `InitCompleteMessage` after sending a `ClusterInitRequestMessage`. If there is no response within this time frame, it indicates that there was a problem during initialisation and the system is shut down.

If both clusters are successfully initialised, the `Referee` sends both `ClusterManagers` a `MatchStartMessage`—an empty message that signals to an agent that it should begin searching from the starting LOA position.

Once an agent receives a `MatchStartMessage`, its search is not interrupted until the match is over i.e. our agents *ponder* during an opponent's thinking time. It is the responsibility of the agent to keep an up-to-date representation of the current game state at all `RankManagers` that maintain the search tree representation. We call this type of `RankManager` a `SearchRank`.

The `Referee` requests moves from agents in an alternating fashion, starting with the agent playing black. We make use of the following four message types during an agent's turn in a match:

- `GenMoveMessage`: A message that signals to an agent that its turn has begun. Contains the previous move made in the match, if there is one. Upon receiving this message, an agent must apply the move contained in the message to its game state representation and provide the `Referee` with a legal move from the resulting board state within some pre-defined time constraint (provided as a program argument).

- `FinalMoveMessage`: The `Referee` expects this message in response to a `GenMoveMessage`. It contains the agent's chosen move and the total number of simulations it performed since receiving the `GenMoveMessage`

- `SearchResultMessage`: A message containing the statistics that a `SearchRank` has accumulated for every move from the root, as well as the number of simulations it performed since the last `GenMoveMessage`. A `ClusterManager` uses these messages to determine the agents final move and construct the `FinalMoveMessage`.

- `ApplyMoveMessage`: Communicates the final move selected by the `ClusterManager` to all `SearchRanks`.

A sequence diagram depicting a single turn in a match is provided in Figure 4.9 and proceeds as follows:

1. The `Referee` sends a `GenMoveMessage` to the `ClusterManager` of the agent that must make a move. If it is the first move in the match, the message is empty, otherwise it contains the move that was received in the previous `FinalMoveMessage`.

2. The `ClusterManager` forwards the `GenMoveMessage` to all `SearchRanks` in the agent's cluster.

3. `SearchRanks` apply the move contained in the `GenMoveMessage` to their internal game state representation and send an `ApplyMoveMessage` to each of their `Workers`. Instead of sharing the `SearchRank`'s game state representation, `Workers` maintain their own internal game state representations to prevent simultaneous updates from corrupting the `SearchRank`'s game state representation.

4. When the agent's allotted time per move is near completion, each `SearchRank` retrieves the statistics it has accumulated for moves from the root, constructs a `SearchResultMessage`, and sends it to the `ClusterManager`. If the `SearchRank` does not have statistics for the root node, it sends an empty `SearchResultMessage`.

5. When the `ClusterManager` has received all `SearchResultMessages`, it combines the results if necessary, sends a `FinalMoveMessage` to the `Referee`, and sends an `ApplyMoveMessage` to all `SearchRanks`. The manner in which results are combined is specific to each of our implementations and is discussed in Section 4.4.

6. Each `SearchRank` applies the move contained in the `ApplyMoveMessage` to their internal game state representation and forwards the message to each of its `Workers`. The `Workers` then apply the move to their own game state representations.

This process is repeated until an agent either: makes a move that leads to a terminal position, fails to make a move within its allotted time, or makes an illegal move. At this point, the `Referee` sends a `MatchOverMessage` to both `ClusterManagers` and logs the winner of the match, the reason for termination, and the average simulations per second that each agent achieved at every turn.

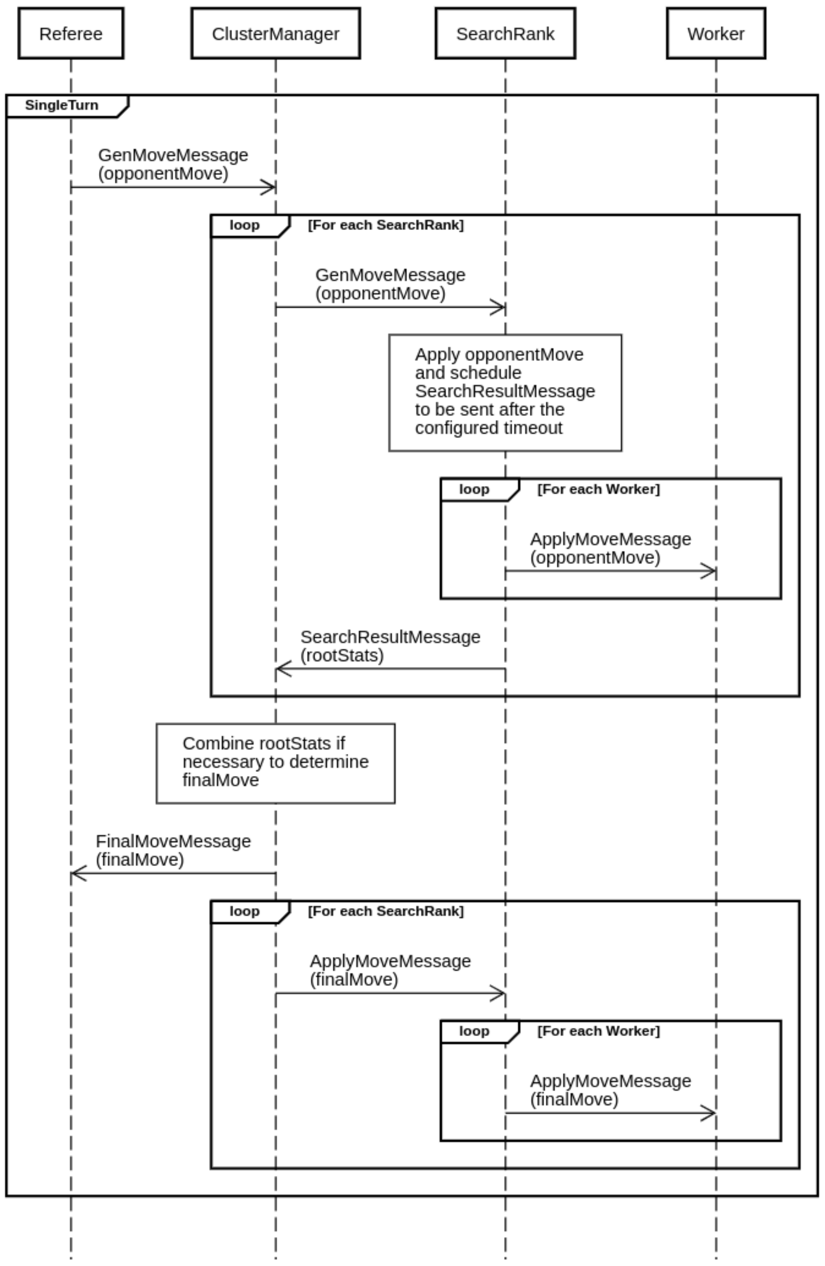
Referee
ClusterManager
SearchRank
Worker
SingleTurn
GenMoveMessage (opponentMove)
loop [For each SearchRank]
GenMoveMessage (opponentMove)
Apply opponentMove and schedule SearchResultMessage to be sent after the configured timeout
loop [For each Worker]
ApplyMoveMessage (opponentMove)
SearchResultMessage (rootStats)
Combine rootStats if necessary to determine finalMove
FinalMoveMessage (finalMove)
loop [For each SearchRank]
ApplyMoveMessage (finalMove)
loop [For each Worker]
ApplyMoveMessage (finalMove)

Figure 4.9: A sequence diagram depicting a single move being made in a match.

Upon receiving a `MatchOverMessage`, a `ClusterManager` forwards the message to all cluster nodes, causing them to terminate their actor systems. Finally, the last running components are shut down when both `Cluster-Managers` terminate their respective actor systems. A sequence diagram depicting a full match—from cluster initialisation to shut down—is provided in Figure 4.10.

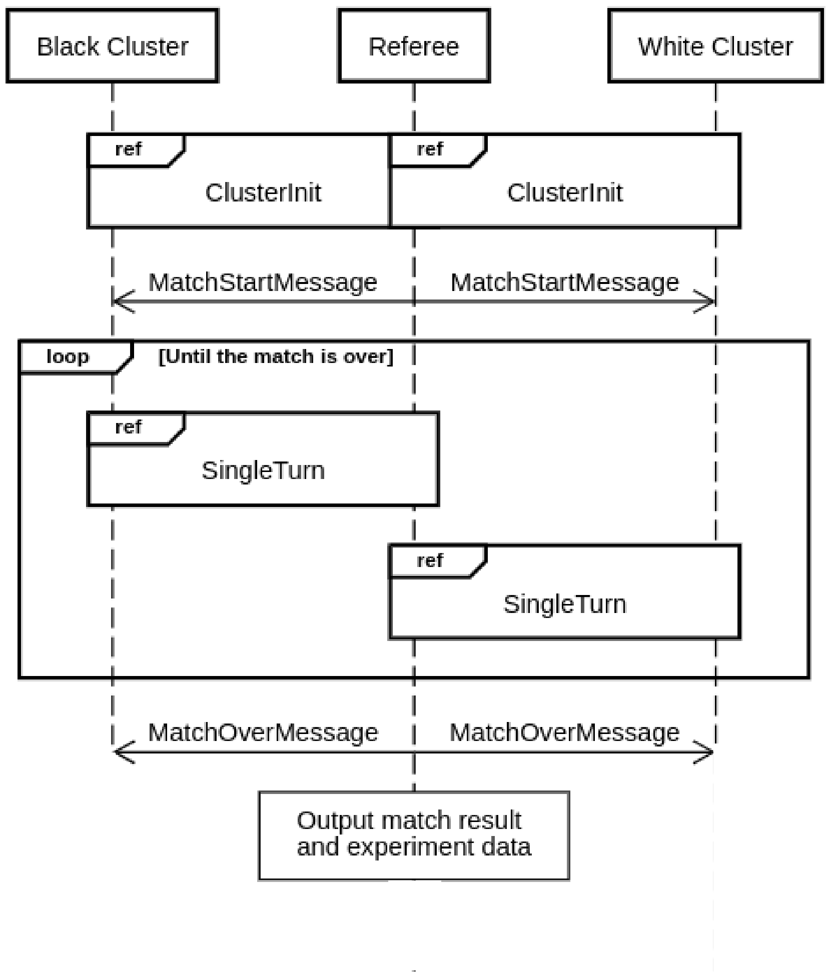

Figure 4.10: A sequence diagram depicting a full match being run.

4.4 Distributed MCTS Agents

As discussed in Section 4.3, an agent a consists of one `ClusterManager` and N_a `ClusterMembers` that each instantiate a `RankManager` based on the type of `ClusterMemberInitMessage` it receives from its `ClusterManager`. `RankManagers` instantiate and manage a number of `Workers` that carry out the majority of the search operations.

In Section 4.3, we provided an overview of the system in terms of these abstract components without providing agent-specific implementation details. In this section, we discuss the implementation of the `ClusterManagers`, `RankManagers`, `Workers` and `ClusterMemberInitMessages` that make up each of our distributed MCTS agents.

4.4.1 Distributed Leaf Parallelisation

Our distributed leaf parallelisation implementation is based on the MLMP strategy developed by Cazenave and Jouandeau [15] that is discussed in Section 2.2.4.1. While the initial proposal of MLMP leaf parallelisation consisted of a single-threaded traverser and a number of single-threaded simulators, our implementation uses parallelised traversers and simulators to take advantage of multiple CPU cores on a single CN.

A leaf parallelisation agent a with N_a CNs and $M_a \geq 2$ CPU cores per CN consists of one `TraverserRank`—the single `SearchRank` in our leaf parallelisation agent—and $(N_a - 1)$ `SimulatorRanks`. The responsibilities of these `RankManager` implementations are as follows:

- `TraverserRank`: Responsible for managing the transposition table and performing selection, expansion and backpropagation. It is initialised with a `TraverserRankInitMessage` that contains the `ActorRef` of all the `SimulatorRanks` and instantiates $M_a - 1$ `TraverserWorkers` that share the transposition table. The remaining CPU core is responsible for communicating with `SimulatorRanks` and delegating search tasks to `TraverserWorkers`.

- `SimulatorRank`: Responsible for performing the simulation phase of MCTS. It is initialised with a `SimulatorRankInitMessage` that contains the `ActorRef` of the `TraverserRank` and instantiates M_a `SimulatorWorkers` that perform M_a simulations in parallel when a simulation request is received. When the simulations are complete, the sum of the rewards is sent to the `TraverserRank`.

Once the cluster has been successfully initialised and the `LeafClusterManager`—the `ClusterManager` implementation for leaf parallelisation—sends a `MatchStartMessage` to the `TraverserRank`, the search progresses as follows:

1. For each `SimulatorRank` r, the `TraverserRank` descends the tree, adds a new `TreeNode` containing the final game state $s(r)$, and sends a `SimulationRequestMessage` to r that contains the Zobrist hash of every game state encountered during descent (the *descent path*—used for backpropagation in step 5) as well as the `LOABoard` for $s(r)$.

2. Upon receiving a `SimulationRequestMessage`, the `SimulatorRank` r forwards the message to each of its `SimulatorWorkers` so that M_a simulations can be performed in parallel for $s(r)$. When a `SimulatorWorker` has completed a simulation, it sends a `SimulationResponseMessage` to its `SimulatorRank`.

3. When the `SimulatorRank` has received all M_a `SimulationResponseMessages`, the sum of the simulation rewards and the descent path is placed in a new `SimulationResponseMessage` and sent to the `TraverserRank`.

4. Upon receiving a `SimulationResponseMessage` from a `SimulatorRank`, the `TraverserRank` forwards the message to the `TraverserWorker` with the fewest messages in its message queue via Akka's built-in `SmallestMailboxPool`.

5. When a `TraverserWorker` receives a `SimulationResponseMessage` from a `SimulatorRank` r, it backpropagates the total simulation reward and M_a visits to every `TreeNode` on the descent path, performs selection and expansion again, and sends a new `SimulationRequestMessage` to r.

4.4.2 Distributed Root Parallelisation

Our distributed root parallelisation implementation performs a decoupled search on every CN available to the agent while using the slow-tree parallelisation approach to sharing node statistics. As discussed in Section 3.3, slow-tree parallelisation periodically shares statistics for nodes up to a certain depth that have been involved in a pre-defined percentage of the total playouts. The maximum depth for sharing d, the minimum percentage of total playouts p, and the sharing frequency f, are provided as program arguments.

The search performed on each CN is parallelised using either leaf parallelisation or tree parallelisation to take full advantage of the available CPU cores. The `ClusterManager` implementation used by a root parallelisation agent is called a `RootClusterManager`, and it assigns the same role to every `ClusterMember` during initialisation.

We implement two `RankManagers` for root parallelisation. The `RankManager` implementation that an agent uses depends on the parallelisation technique used at each CN. Tree parallelisation is performed by a `RootTreeSearchRank` while leaf parallelisation is performed by a `RootLeafSearchRank`. The functionality of each `RankManager` implementation for a root parallelisation agent a with N_a CNs and M_a CPU cores per CN is as follows:

- `RootLeafSearchRank`: Instantiated when a `ClusterMember` receives a `RootLeafRankInitMessage` containing the `ActorRef` of every other `RootLeafSearchRank` in the cluster. Spawns a single `TraverserWorker` and $M_a - 2$ `SimulatorWorkers` that perform MLMP leaf parallelisation as described in Section 2.2.4.1 and Section 4.4.1. The remaining CPU core is responsible for sending node statistics to other `RootLeafSearchRanks`, as well as receiving node statistics from other `RootLeafSearchRanks` and updating the tree accordingly.

- `RootTreeSearchRank`: Instantiated when a `ClusterMember` receives a `RootTreeRankInitMessage` containing the `ActorRef` of every other `RootTreeSearchRank` in the cluster. Spawns $M_a - 1$ `TreeWorkers` that perform individual searches on a shared transposition table as described in Section 2.2.4.3. The remaining CPU core is responsible for communicating node statistics to other `RootTreeSearchRanks` and incorporating remote statistics.

During the search, periodic sharing of node statistics is achieved through the use of an Akka scheduler that enables an actor to periodically send a message with some interval between messages. The scheduler is managed by the search rank actor and sends the search rank an empty `StartStatShareMessage` message every $\frac{1}{f}$ seconds. Upon receiving this message, the search rank performs a depth-first search (DFS) on the search tree up to depth D. When a node is encountered with a playout contribution of at least p, the search rank broadcasts a `NodeStatShareMessage` containing the node's hash, the `LOABoard` associated with the node, and both arrays of unsynchronised statistics (see Section 4.2.1) to all other search ranks and incorporates these values into the `TreeNode`'s synchronised statistic arrays.

Upon receiving a `NodeStatShareMessage`, a search rank retrieves the `TreeNode` with the given hash from the transposition table. If it does not exist, a new `TreeNode` is added to the transposition table using the `LOABoard` contained in the message. The visit and reward arrays contained in the message are then added to the `TreeNode`'s synchronised statistic arrays.

4.4.3 Distributed Tree Parallelisation

Our distributed tree parallelisation agents rely on the Transposition Table Driven Scheduling (TDS) approach to sharing the tree as discussed in Section 3.4.1. We implement vanilla TDS, depth-first UCT (see Section 3.4.2) and UCT-Treesplit (see Section 3.4.3). This section begins with a discussion of our vanilla TDS implementation in Section 4.4.3.1. Sections 4.4.3.2 and 4.4.3.3 focus on our implementation of df-UCT and UCT-Treesplit as enhancements to vanilla TDS, respectively.

4.4.3.1 Vanilla TDS

As discussed in Section 3.4.1, TDS partitions the transposition table among an agent's CNs. When a game state s is encountered during the search, a message is sent to the CN that manages the transposition table entry for s so that the necessary operation can be performed on the `TreeNode` associated with s.

The `ClusterManager` implementation used by vanilla TDS—the `TDSClusterManager`—assigns the same role to every CN in the cluster. The `RankManager` implementation that is instantiated at every CN is called a `TDSSearchRank`. For a vanilla TDS agent a and M_a CPU cores per CN, every `TDSSearchRank` instantiates $M_a - 1$ `TDSWorkers` that are responsible for performing the search. The remaining CPU core is responsible for communicating with remote CNs and delegating work to its `TDSWorkers`.

Our vanilla TDS implementation uses the the following two messages to perform playouts:

- `TDSSearchMessage`: Signals to a `TDSSearchRank` that it must perform the selection, expansion or simulation phase of MCTS. Contains a `LOABoard` that represents the current board state, as well as a list of `TDSSelections` that constitute the path in the tree leading up to the current state. Each `TDSSelection` consists of a Zobrist hash of a board state and the index of the selected move. The deterministic nature of our approach to move generation allows us to pass move indices between CNs instead of full move representations (see Section 4.1.1).

- `TDSReportMessage`: When a `TDSSearchRank` performs a random simulation and obtains a reward, a `TDSReportMessage` is sent to the home processor of every node encountered during tree descent so that these nodes' reward statistics can be updated. Each message contains a `TDSSelection` that indicates the node whose statistics should be updated and the specific move index to update, as well as the reward obtained after simulation.

The search begins when the `TDSSearchRank` that manages the transposition table entry for the starting LOA position receives a `MatchStartMessage`. For an agent with N_a CNs and M_a CPUs per CN, it initiates $N_{\text{par}} \times N_a \times (M_a - 1)$ playouts, where N_{par} is a parameter that determines the number of parallel searches the agent performs. The choice of N_{par} has a substantial influence on agent performance, and we provide an outline of our tuning process in Section 5.2.3.

The `TDSSearchRank` initiates a playout by sending a `TDSSearchMessage` with an empty selection list and a `LOABoard` for the starting LOA game state to its `TDSWorker` with the fewest messages in its message queue via Akka's built-in `SmallestMailboxPool`.

When a `TDSWorker` receives a `TDSSearchMessage`, it performs a transposition table lookup for the Zobrist hash of the `LOABoard` contained in the message. If the node exists, the selection procedure finds the move with the highest UCT value (or an unexplored move if one exists) and adds a new `TDSSelection` with the node's hash and the chosen move index to the list contained in the message. It then applies the chosen move to the `LOABoard` and forwards the message to the `TDSSearchRank` that manages the transposition table entry for the resulting game state.

If there is no transposition table entry for the `LOABoard` contained in the `TDSSearchMessage`, a new `TreeNode` is added for the `LOABoard` and a simulation is performed. After simulation, the `TDSWorker` iterates through the list of `TDSSelections` in the `TDSSearchMessage` and sends a `TDSReportMessage` to the home processor of each Zobrist hash in the list.

Upon receiving a `TDSReportMessage`, a `TDSSearchRank` forwards the message to its `TDSWorker` with the smallest number of messages in its message queue so that the node's statistics can be updated. If the `TDSReportMessage` is targeted at the root of the tree, the `TDSSearchRank` initiates a new playout from the root.

4.4.3.2 df-UCT

df-UCT inherits most of its functionality from vanilla TDS. As discussed in Section 3.4.2, df-UCT aims mitigates the network communication overhead incurred at the home processors of near-root nodes in TDS by implementing a depth-first variation of UCT that delays backpropagation until the search is no longer focused on the most promising path.

The `ClusterManager` and `RankManager` implementations used by df-UCT—`DFUCTClusterManager` and `DFUCTSearchRank`—behave in the same way as vanilla TDS. However, the `DFUCTSearchRank` instantiates `DFUCTWorkers` that incorporate functionality to manage the df-UCT stack described in Section 3.4.2. As in vanilla TDS, the df-UCT search is driven by search messages and report messages. While df-UCT uses the same report message type as vanilla TDS—the `TDSReportMessage`—for backpropagation, we implement a `DFUCTSearchMessage` that contains the full df-UCT stack instead of a list of `TDSSelections`.

When a `DFUCTWorker` receives a `DFUCTSearchMessage` and performs selection on a node n, it constructs a `DFUCTSelection` and pushes it onto the df-UCT stack contained in the message. Each `DFUCTSelection` contains the following data:

- The Zobrist hash of n.
- The index of the move with the highest UCT value, as well as its visit count, total reward and knowledge value.
- The index of the move with the second highest UCT value, as well as its visit count, total reward and knowledge value.

When a `DFUCTWorker` adds a new `TreeNode` to the transposition table and performs a simulation, it does not immediately backpropagate the reward to every node encountered in the playout. Instead, the `DFUCTWorker` performs the following tasks:

1. For each `DFUCTSelection` in the stack, update the reward of the best move and increment its visit count.

2. If the UCT value of the second best move becomes greater than that of the best move for any `DFUCTSelection` s in the stack, or if a new move should be unpruned in the case of progressive unpruning, pop every `DFUCTSelection` on the path from the leaf up to and including s, send a `TDSReportMessage` to their respective `DFUCTSearchRanks`, and send a `DFUCTSearchMessage` to the home processor of s with the remaining stack so that selection can resume there.

3. If the best move remains the same for every entry in the stack, send a `TDSReportMessage` to the parent of the newly expanded `TreeNode` and initiate selection there.

4.4.3.3 UCT-Treesplit

Similarly to df-UCT, UCT-Treesplit inherits most of its functionality from vanilla TDS, but mitigates the communication overhead incurred by TDS by duplicating and periodically synchronising statistics for near-root nodes (see Section 3.4.3). The initial UCT-Treesplit proposal made use of dedicated broadcast ranks to handle node duplication and synchronisation. However, their broadcast ranks are single-threaded, and are presented as a proof-of-concept as opposed to an ideal solution [47]. We implement UCT-Treesplit with broadcast ranks that take advantage of all the CPU cores available at a CN in order to increase the number of search ranks that an agent may use.

The `ClusterManager` and `SearchRank` implementations used by a UCT-Treesplit agent are called `TreesplitClusterManager` and `TreesplitSear-chRank`, respectively. Additionally, some CNs in a UCT-Treesplit agent's cluster are `TreesplitBroadcastRanks`. A `TreesplitSearchRank` spawns `TreesplitSearchWorkers` and a `TreesplitBroadcastRank` instantiates `Tr-eesplitBroadcastWorkers`.

The initial UCT-Treesplit implementation assigned one broadcast rank for every four search ranks [47]. Since we implement multi-threaded broadcast ranks, the number of `TreesplitBroadcastRanks` in our UCT-Treesplit cluster is dependent on the number of CPU cores available at each CN, and we assign one `TreesplitBroadcastWorker` for every four search ranks.

In addition to the subset of the transposition table that is maintained at every `TreesplitSearchRank`, we implement a separate cache to hold duplicated nodes that are shared by all of the search rank's `TreesplitSearchWorkers`.

The search performed by UCT-Treesplit is identical to that of vanilla TDS until a node becomes eligible for duplication or synchronisation (see Section 3.4.3). When this happens, the `TreesplitSearchWorker` that encountered the node sends a `NodeStatShareMessage` to it's dedicated `TreesplitBroadcastWorker`. This message is identical to the one used by our root parallelisation implementation discussed in Section 4.4.2.

As discussed in Section 3.4.3, broadcast ranks in UCT-Treesplit artificially delay forwarding node statistics to remote search ranks so that messages originating from different search ranks can be merged, thereby reducing the number of messages processed by search ranks. In our implementation, `NodeStatShareMessages` received by a `TreesplitBroadcastWorker` are stored in a `HashMap` with Zobrist hashes as a keys and a list of `NodeStatShareMessages` as values. The `TreesplitBroadcastWorker` periodically merges the messages for each Zobrist hash in the map to construct new `NodeStatShareMessages` that are then sent to all other `TreesplitSearchRanks` via their dedicated `TreesplitBroadcastWorkers`.

When a `TreesplitSearchRank` receives a `NodeStatShareMessage`, it adds a new cache entry for the provided game state if none exists and incorporates the node's statistics as discussed in Section 4.4.2.

4.5 Summary

In this chapter, we presented the design and implementation of our test domain, Lines of Action, as well as our distributed MCTS agents.

In Section 4.1, we discussed our implementation of Lines of Action. This included our approach to move generation via incremental updates to line configurations and terminal state detection via the quad heuristic in Sections 4.1.1 and 4.1.2, respectively. We also discussed our incorporation of knowledge through static evaluation and move categories derived from expert play databases in Sections 4.1.3.1 and 4.1.3.2, respectively.

We presented our game tree implementation in Section 4.2, beginning with a summary of the `TreeNode` interface in Section 4.2.1. We noted that we required two `TreeNode` implementations: a `VolatileTreeNode` for agents that share the game tree representation, and a `SimpleTreeNode` for agents that do not. We provided a list of data structures that are stored in each `TreeNode` in order to maintain the statistics required to perform the search, and concluded the section with a brief overview of our node initialisation procedure.

We provided an overview of our test framework in Section 4.3. We noted that our system is implemented in Java, and that we make use of the Akka framework for message passing and concurrency. In Section 4.3.1, we provided a high-level overview of the Akka cluster that enables our test framework and distributed MCTS agent implementations. We presented the five high-level components that make up our cluster, described their responsibilities, and discussed how they are executed on a physical compute cluster. We depicted and discussed our cluster initialisation and match execution procedures in Section 4.3.2. We provided a detailed description of the messages that are passed between members of the cluster in order to execute a match, and supplied supporting sequence diagrams for clarity. Additionally, we noted that each of our distributed MCTS agents must implement a `ClusterManager`, a `RankManager`, a `Worker` and a `ClusterMemberInitMessage` in order to function within the cluster.

Finally, in Section 4.4, we present the `ClusterManagers`, `RankManagers`, `Workers` and `ClusterMemberInitMessages` that make up each of our distributed MCTS agents. We gave details on the design and implementation of distributed leaf parallelisation, root parallelisation and tree parallelisation in Sections 3.2, 3.3 and 3.4, respectively. Section 3.4 was partitioned into discussions of our TDS, df-UCT and UCT-Treesplit implementations in Sections 4.4.3.1, 4.4.3.2 and 4.4.3.3, respectively.

In the following chapter, we will provide experimental results and analyse the scalability of each of our distributed MCTS implementations.

Chapter 5

Experiments and Results

In Chapter 3, we mentioned that there are at least three scaling properties that a good distributed MCTS algorithm should satisfy:

1. It should take advantage of increased CPU resources to perform more playouts per second (PPS) when more CNs are available.
2. It should take advantage of increased memory resources by building a larger tree when more CNs are available.
3. Its playing strength should improve when more CNs are available.

In this chapter, we present our experimental setup as well as the scalability of each of the distributed MCTS implementations discussed in Chapter 4 in terms of the three metrics above. Section 5.1 outlines our experimental setup. Section 5.2 discusses the parameter tuning procedure and the results obtained. Finally, Section 5.3 presents and analyses the scalability of each of our implementations in terms of the three properties above.

5.1 Experimental Setup

The goal of our experiments is to compare the scalability of the distributed MCTS implementations discussed in Chapter 4 up to 128 CNs. Specifically, we consider leaf parallelisation (Section 4.4.1), root/leaf parallelisation (Section 4.4.2), root/tree parallelisation (Section 4.4.2), TDS (Section 4.4.3.1), df-UCT (Section 4.4.3.2) and UCT-Treesplit (Section 4.4.3.3). Since prior implementations of these algorithms were presented and evaluated on disparate domains and different network architectures, and are therefore difficult to compare, we aim to provide a fair comparison of these algorithms on a common domain and compute infrastructure.

In our experiments, each of the implementations plays 100 Lines of Action (LOA, Section 2.4) matches (50 as black, 50 as white) per CN allotment against a reference opponent with each player having one second allowed per move. In order to conserve computational resources, our choice of reference opponent should ideally use only 1 CN. Our initial experiments revealed that root parallelisation combined with tree parallelisation is our strongest implementation when running on 1 CN, so we use this as our reference opponent.

For each turn in a match, the system logs playouts per second, unique nodes searched, and, for TDS, df-UCT and UCT-Treesplit, the number of `ReportMessages` received at the home processor of the root (Section 4.4.3) for each agent. At the end of the match, the system outputs the winner of the match, and this data is used to generate the graphs depicted in Section 5.3.

We performed our experiments on the Lengau cluster of the Centre for High Performance Computing (CHPC) at the Council for Scientific and Industrial Research (CSIR) [1]. This homogeneous cluster consists of 1368 nodes, each with two 12-core CPUs and 128GB RAM, all interconnected using FDR 56 Gb/s InfiniBand. In order to adhere to the CHPC's usage limitations, we execute two `ClusterMembers` (Section 4.4) on a node—one per CPU—and assign half the node's memory to each of these `ClusterMembers`. For the remainder of this chapter, we refer to the hardware that a `ClusterMember` has available to it as a CN. Thus, a CN has access to 12 CPU cores and 64GB RAM.

5.2 Parameter Tuning

Most strong game-playing AIs have a large number of algorithm parameters that can drastically effect their performance [19, 22]. Therefore, choosing good values for these parameters—or *tuning*—is imperative to maximising the playing strength of a game-playing AI.

Although it is often possible to make educated guesses on good values for some parameters, the results will, in general, not be optimal [19]. An alternative approach to such manual tuning is to simulate matches against a reference opponent with varying parameter value combinations, record the win rates achieved by each combination, and choose the one that yields the best results. Since the number of simulated matches that is required in order to obtain a reliable win rate estimate for each parameter combination is large [19], such a *gridsearch* approach is a time-consuming and computationally expensive process that many authors of game-playing AIs try to avoid [22].

In order to avoid wasting time and computational resources on tuning using this approach, we automate the tuning process using Rémi Coulom's optimization program, CLOP [22].

5.2.1 Tuning with CLOP

Our distributed MCTS implementations have a number of parameters that can be tuned to adjust performance. These include UCT parameters (Section 2.2.2), MCTS enhancement parameters (Section 2.2.3) and parameters for individual parallelisation techniques (Section 2.2.4). Due to time and resource constraints, we limited the parameters that we tuned using CLOP to the following set:

- the UCT constant C_{uct} (Equation 2.3);
- the FPU constant (Section 2.2.3.1);
- the progressive bias constant C_{pb} (Equation 2.5); and
- the initial window size U_0 and unpruning rate μ to be used for progressive unpruning (Equation 2.4).

The remaining parameters that are present in this thesis are all specific to some MCTS parallelisation technique, and were set to the values presented in their respective publications, with a single exception: the constant N_{par} that determines the number of parallel searches performed for TDS, df-UCT and UCT-Treesplit (Sections 4.4.3.1, 4.4.3.2 and 4.4.3.3, respectively). The tuning of N_{par} is discussed later in Section 5.2.3.

An issue with tuning our chosen parameters is that some of them cannot be used simultaneously. More specifically, FPU and progressive unpruning are attempted solutions for the same problem and are therefore incompatible. Also, move categories and static evaluation cannot simultaneously be used to determine H_i for progressive bias. Therefore, we used CLOP to tune the following four agents independently:

- A serial agent with FPU that uses our evaluation function to determine H_i (**FPU-EF**)
- A serial agent with FPU that uses move categories to determine H_i (**FPU-MC**)
- A serial agent with progressive unpruning that uses our evaluation function to determine H_i (**PU-EF**)
- A serial agent with progressive unpruning that uses move categories to determine H_i (**PU-MC**)

We chose to limit our tuning to serial versions of each agent in order to limit the computational expense and time spent tuning. We made the assumption that the parameters chosen for the serial agents would be effective choices for the distributed agents as well, but we note that they may not be optimal, so that individually tuning each agent is a potential avenue for future work.

The parameter sets obtained by CLOP for the four agents above are shown in Table 5.1.

Table 5.1: Parameters obtained for each of our candidate agents with CLOP.

Agent	C_{uct}	C_{pb}	FPU	U_0	μ
PU-MC	1.0748	12.3619	-	1	2.5365
PU-EF	0.8610	14.2020	-	2	2.6011
FPU-MC	0.6838	16.0458	1.1050	-	-
FPU-EF	1.1077	10.8757	1.1734	-	-

5.2.2 Final Parameter Choices

Once CLOP had tuned each of the agents' parameters, we ran a round-robin tournament to determine which combination of parameters to use in our final parameter set. In the tournament, each agent played 1000 matches—500 as black and 500 as white—against each other agent. The results of this tournament are shown in Table 5.2.

The results of the tournament show clear dominance by progressive unpruning over FPU and a slight advantage of move categories over the evaluation function. Since the agents with progressive unpruning will unprune the children of a node in decreasing order of transition probability/position evaluation (Section 4.1.3), the dominance of these agents over FPU is to be expected. The advantage that move categories have over our static evaluation function may be a result of the simplicity of our evaluation function and/or the computational overhead incurred by a large number of evaluation function calls. Determining the precise reason for this advantage is a possible avenue for future work, as are experiments with a more comprehensive evaluation function.

Since the combination of progressive unpruning and move categories performed the best in the round-robin tournament, we use these enhancements with the parameters obtained in Section 5.2.1 for all of our subsequent experiments.

Table 5.2: Results of the round-robin tournament with the four agents from Section 5.2.1. Cell values indicate the number of times the agent in the row defeated the agent in the column.

	PU-MC	**PU-EF**	**FPU-MC**	**FPU-EF**
PU-MC		640	963	992
PU-EF	360		933	972
FPU-MC	37	67		710
FPU-EF	8	28	290	

5.2.3 Tuning N_{par}

As discussed in Section 4.4.3.1, the number of simultaneous searches performed by our TDS, df-UCT and UCT-Treesplit agents are determined by the value of N_{par}—a parameter that, when multiplied by the number of CPUs available to the agent, determines the number of parallel searches performed. In their original papers, the authors of UCT-Treesplit and df-UCT presented their results with $N_{\text{par}} = 5$ and $N_{\text{par}} = 20$, respectively, while the authors of TDS did not specify a value [47, 62, 44].

In our initial experiments, we used $N_{\text{par}} = 20$, but found the resulting scalability of these agents to be unsatisfactory, which led us to experiment with other values of N_{par}.

When we initially decided to tune N_{par}, we wanted to limit the computational resources used for the tuning process. Since playing strength is a computationally expensive metric to accurately estimate (see Section 5.2.1), we decided to use playout rate scalability as a proxy metric for tuning. The results of these initial experiments are presented in Section 5.2.3.1.

Once we had tuned N_{par} using the results of these PPS experiments, we continued with the playing strength scalability experiments provided in Section 5.3.3. However, we found that setting N_{par} to the relatively large values obtained through tuning did not lead to stronger play. Therefore, we decided that it would be necessary to tune the parameter using playing strength scalability after all. The results of these experiments are presented in Section 5.2.3.2.

5.2.3.1 Playout Rate

As discussed in Section 5.2.1, using playing strength as a metric for manual tuning is incredibly time-consuming and computationally expensive, and this becomes an even greater deterrent when each experiment involves a large number of compute nodes. In order to limit the resources used for tuning N_{par}, we initially decided to use scalability in terms of playouts per second (PPS) from the opening LOA position as a proxy metric for tuning.

Our initial experiments indicated that an agent's playout rate from the starting LOA position does not vary significantly given constant computational resources and time constraints. Therefore, the average PPS achieved by an agent with some N_{par} across 10 one-second searches from the initial LOA position was considered sufficient to gauge the effectiveness of the given N_{par}.

We depict the turn 1 PPS scalability of TDS, df-UCT and UCT-Treesplit with varying values of N_{par} in Figures 5.1, 5.2 and 5.3, respectively.

All three of our implementations achieve better PPS scalability with greater values of N_{par} than those presented in the literature. Most notably, out of our candidate values, our implementation of UCT-Treesplit performs best when $N_{\text{par}} = 500$—an order of magnitude larger than in the original paper. There are a number of possible reasons for our implementations benefiting from greater N_{par}. One such possibility is that we suffer a higher network communication overhead due to Akka's lack of RDMA support—something that the original authors of UCT-Treesplit made use of [47]. Other possibilities include Akka's message serialization being less performant than MPI and large message payloads forcing our agents to spend more time serializing and de-serializing messages. The exact reason is not obvious and difficult to investigate while also controlling for hardware and implementation differences. Determining the relationship between hardware/software specifications and good choices of N_{par} is a possible avenue for future work.

Another interesting result is the poor performance of TDS and df-UCT for large N_{par}—we note that these agents are unable to complete a single playout with 64 CNs and above with $N_{\text{par}} = 500$. This can be attributed to the sheer number of `SearchMessages` that the home processor of the root must handle at the start of the match. UCT-Treesplit does not suffer from this because its `SearchMessages` are distributed among the available `SearchRanks` (Section 4.4.3).

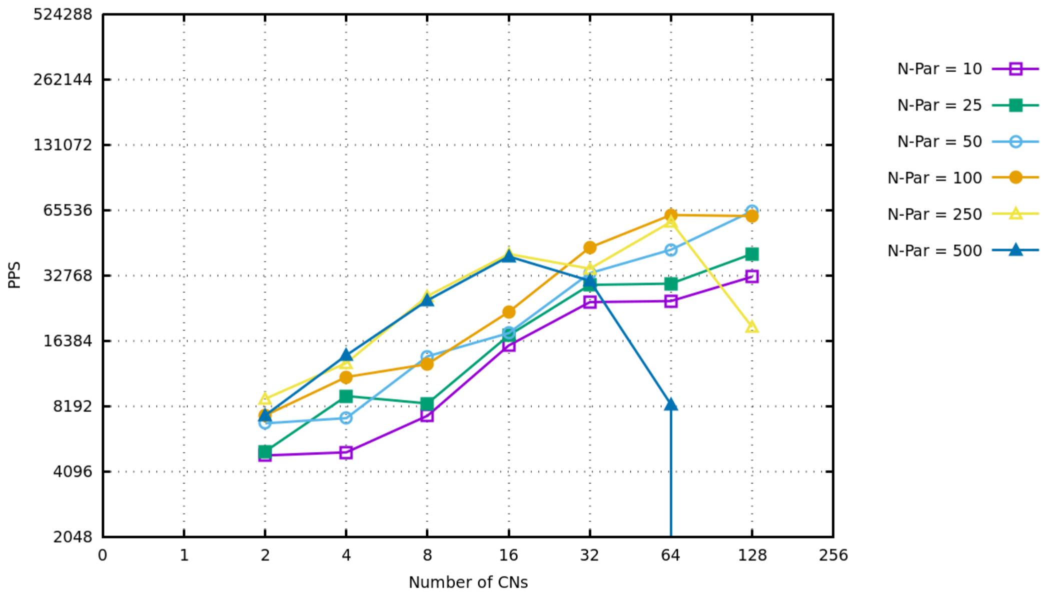

Figure 5.1: PPS achieved by TDS at turn 1 for varying values of N_{par} and an increasing number of CNs.

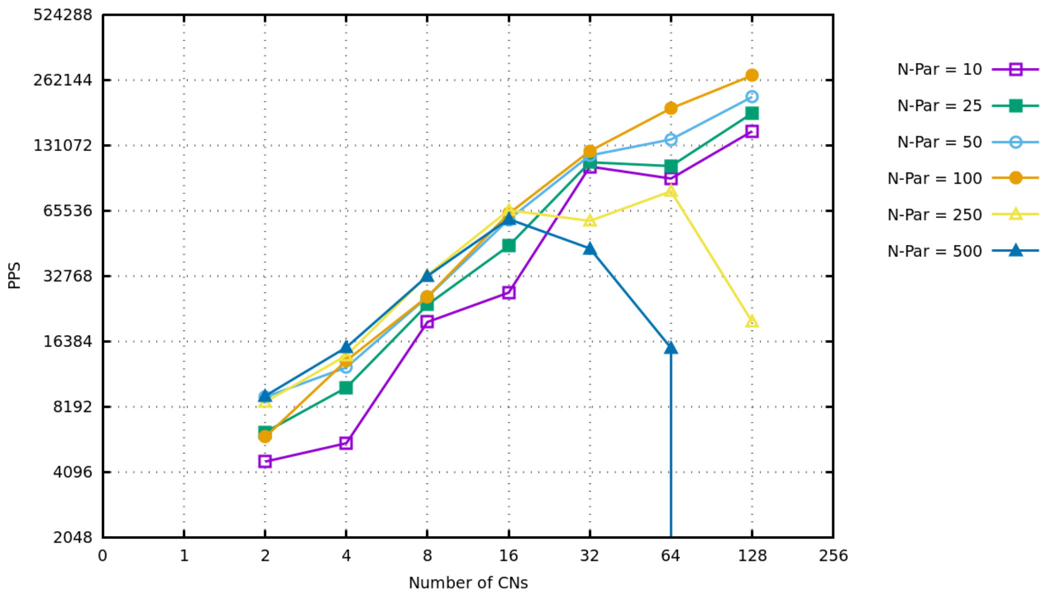

Figure 5.2: PPS achieved by df-UCT at turn 1 for varying values of N_{par} and an increasing number of CNs.

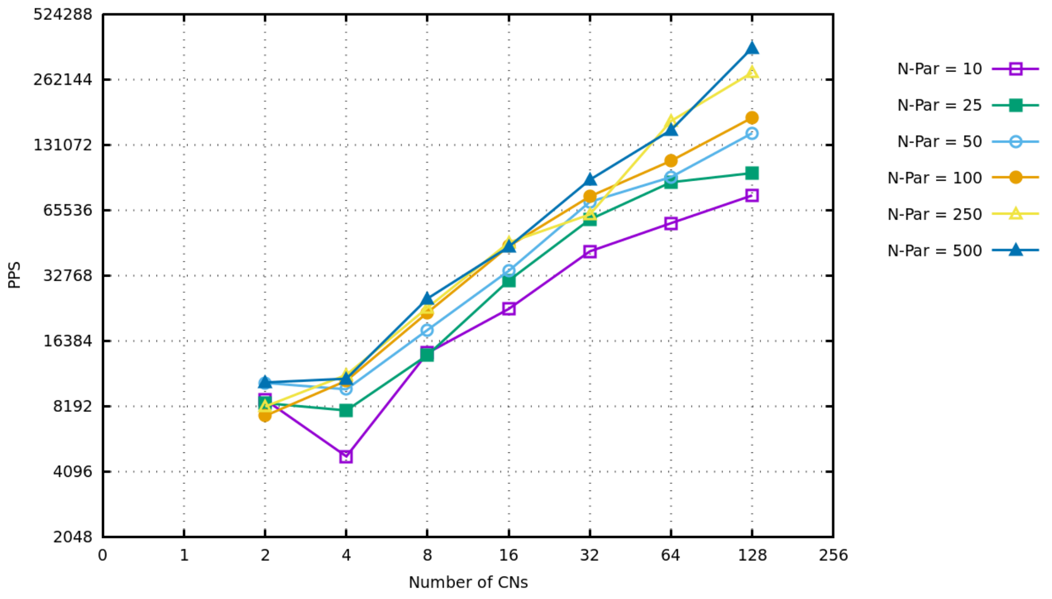

Figure 5.3: PPS achieved by UCT-Treesplit at turn 1 for varying values of N_{par} and an increasing number of CNs.

5.2.3.2 Playing Strength

The results discussed in Section 5.2.3.1 indicate that larger N_{par} leads to improved PPS scalability as long as the home processor of the root node is able to handle the initial set of `SearchMessages`. However, in our initial playing strength experiments with N_{par} set to these large values, we found that all three of the agents performed very poorly in terms of playing strength, with none of the agents managing a win-rate of greater than 10% for any number of CNs greater than 1 against a serial agent.

We believe that the reason for this is that tree parallelisation suffers from an inherent search overhead. If multiple threads perform playouts starting from the same tree node and the node is found to be unfavourable by one of the threads, the work done by the other threads are wasted. This is known as the *parallel effect*, and the extent to which this effect hinders playing strength is largely determined by the degree of parallelism present in the system [56] and the delay between starting a search and incorporating the results for future selection.

These disappointing initial results led us to rethink our tuning strategy and sacrifice some computational resources to tune N_{par} using playing strength instead of using PPS as a proxy for playing strength. We ran 50 matches per agent (25 as white and 25 as black) against a reference serial opponent with each player having 1 second per move. The resulting playing strength results for TDS, df-UCT and UCT-Treesplit are provided in Figures 5.4, 5.5 and 5.6 respectively. We did not perform experiments using 1 CN because the poor playing strength we observed in our initial experiments were obtained with configurations that had more than 1 CN. Therefore, we omitted the 1 CN agents from these experiments to conserve computational resources.

In order to conserve computational resources, we limited our candidate N_{par} to 1, 5, 10, 20, 50. We did not include the larger values of N_{par} attained in Section 5.2.3.1 because we had already determined that they yield poor playing strength scalability. As seen in Figures 5.4, 5.5 and 5.6, playing strength scalability is already limited with $N_{\text{par}} = 50$, so using valuable computational resources to test with larger values would have been wasteful.

For $N_{\text{par}} > 1$, TDS scales up to 16 CNs, with more CNs incurring a decline in playing strength. However, our implementation scales up to 128 CNs with $N_{\text{par}} = 1$, with a dip in win-rate at 64 CNs that may be attributed to chance. Since the playing strength scalability of TDS declines consistently with increased N_{par}, we attribute the improved scalability at $N_{\text{par}} = 1$ to reduced communication overhead at the home processor of the root node. Another possible explanation is search overhead incurred by more parallelism, but the cause cannot be determined for certain without further experimentation.

For $N_{\text{par}} = 1$, df-UCT scales up to 128 CNs, and has no scaling to speak of for higher N_{par}. A possible reason for this is the inherent search overhead already present in df-UCT being exacerbated by the massive parallelism for $N_{\text{par}} > 1$. Since the seminal paper on df-UCT did not present playing strength experiments, their choice of $N_{\text{par}} = 20$ did not negatively influence their results, and analysing the search overhead inherent to the algorithm—as well as the effect that N_{par} has on this overhead—is a possible avenue for future research.

UCT-Treesplit scales very well up to 128 CNs for Npar = 5, with higher values having poor scalability after 32 CNs. The dip in playing strength from 2 CNs to 4 CNs can be attributed to the introduction of a second `SearchRank`, which necessitates inter-CN communication throughout the search (our 2 CN UCT-Treesplit implementation consists of one `SearchRank` and one `BroadcastRank`). We can see the same dip in the PPS and NPS and playing strength graphs provided in Figures 5.7, 5.8 and 5.9, respectively.

Based on these results, we choose $N_{\text{par}} = 1$ for TDS and df-UCT and $N_{\text{par}} = 5$ for UCT-Treesplit.

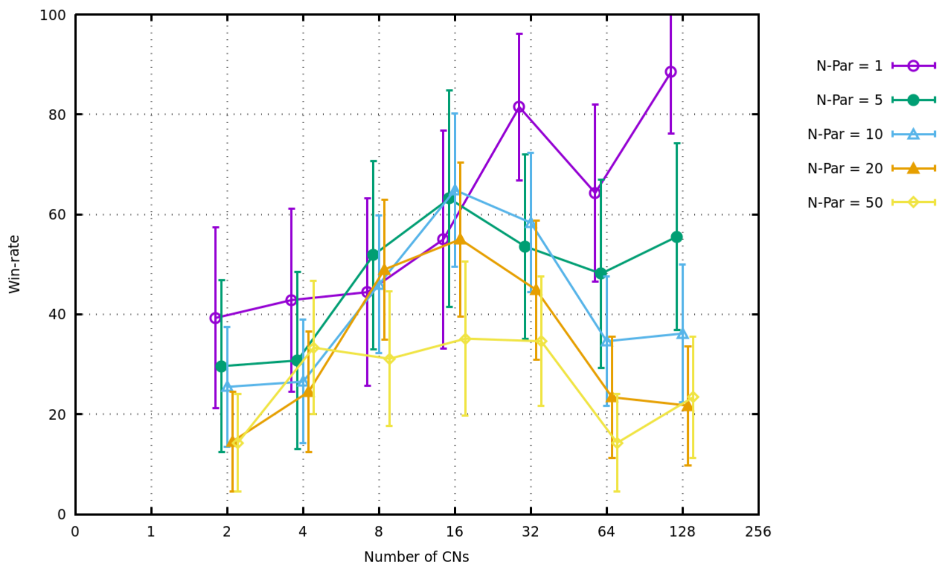

Figure 5.4: Win-rate achieved by TDS after 50 matches versus a serial agent for varying values of N_{par} and an increasing number of CNs. Each win-rate is slightly offset in order to improve legibility and provided with a 95% confidence interval.

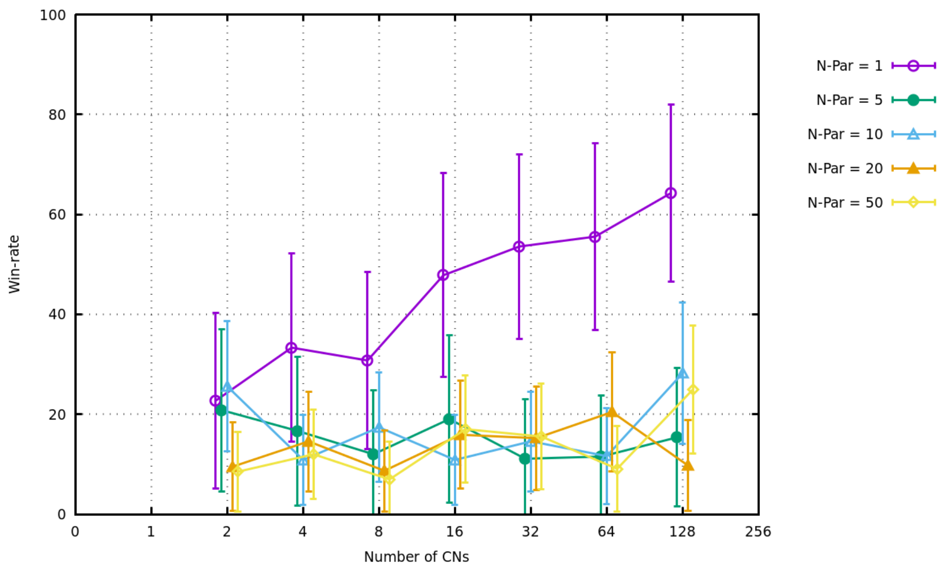

Figure 5.5: Win-rate achieved by df-UCT after 50 matches versus a serial agent for varying values of N_{par} and an increasing number of CNs. Each win-rate is slightly offset in order to improve legibility and provided with a 95% confidence interval.

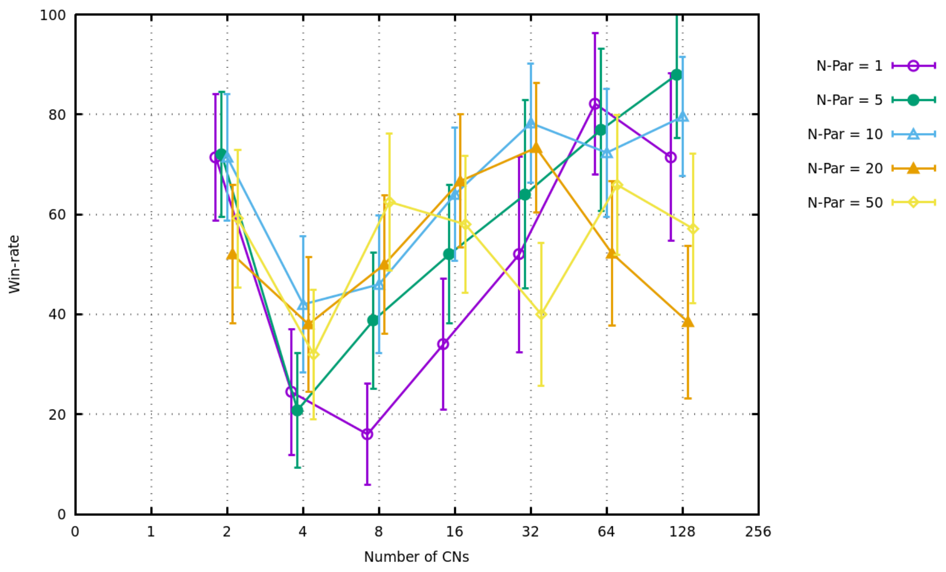

Figure 5.6: Win-rate achieved by UCT-Treesplit after 50 matches versus a serial agent for varying values of N_{par} and an increasing number of CNs. Each win-rate is slightly offset in order to improve legibility and provided with a 95% confidence interval.

5.2.4 Transposition Table Size

As mentioned at the beginning of this chapter, one of our criteria for comparing the distributed MCTS implementations is whether or not they take advantage of increased memory resources by building a larger tree when more CNs are available. The simplest way to do this is to artificially limit the memory available to an agent by reducing the size of its transposition table (Section 4.2.2) such that it fills up throughout the course of the match and then measure the size of the tree the agent is able to construct with increasing CNs.

By doing this, we expect to observe that the scalability of distributed root and leaf parallelisation are limited because, as long as each CN's transposition table is full, adding more CNs will not allow the agent to search any deeper in the tree. In contrast, we expect that TDS, df-UCT and UCT-Treesplit will be able search deeper when they are supplied with more CNs.

Since our transposition table implementation overwrites nodes encountered in previous turns, we must limit the number of entries in each CN's transposition table such that we expect it to be filled with new nodes every turn. Additionally, we would like our distributed tree parallelisation implementations to overtake our root/tree implementation in terms of unique nodes searched at some CN setting so that we can determine the effect this has on playing strength. In order to fulfill these criteria, we limit the number of entries in each CN's transposition table to 2^{13}. Although this number is small considering the large amount of memory available at each CN, this artificial memory limit is required to perform our comparisons.

5.3 Scalability Analysis

This section presents and analyses the scalability of distributed root parallelisation, leaf parallelisation and tree parallelisation in terms of the three scaling properties presented at the beginning of the chapter.

Playout rate scalability is discussed and analysed in Section 5.3.1 and an analysis of scalability in terms of tree size is presented in Section 5.3.2. Finally, in Section 5.3.3, we present the playing strength scalability for each of the distributed MCTS implementations and consider how playout rate and tree size contribute to playing strength.

5.3.1 Playout Rate

The playouts per second (PPS) achieved by an agent measures the number of full MCTS playouts (Section 2.2) that the agent is capable of performing in a second. This metric should provide a good indication of program performance since one would expect that an increase in playout rate will lead to faster node expansion, which in turn will allow the agent to search deeper in the tree within some time constraint and hopefully produce stronger play. Additionally, the value estimates of tree nodes will be more statistically significant, which will allow the agent to prioritise more beneficial parts of the tree and make better move choices.

Ideally, the PPS achieved by an agent will double with a doubling in CNs, but this is not always the case because some parallelization techniques may not be optimally efficient. Additionally, communication overhead in distributed implementations can prohibit PPS scalability [44, 62, 47].

Figure 5.7 depicts the average PPS achieved by each of our distributed MCTS implementations at turn 1 with an increasing number of CNs. It shows that the root parallelisation-based agents have linear PPS scaling. This is because, for root parallelisation, doubling the CN count will implicitly double the number of independent parallel searches performed. This, combined with minimal network communication—only three broadcasts per second in our implementation (Section 4.4.2)—leads to impressive PPS scalability.

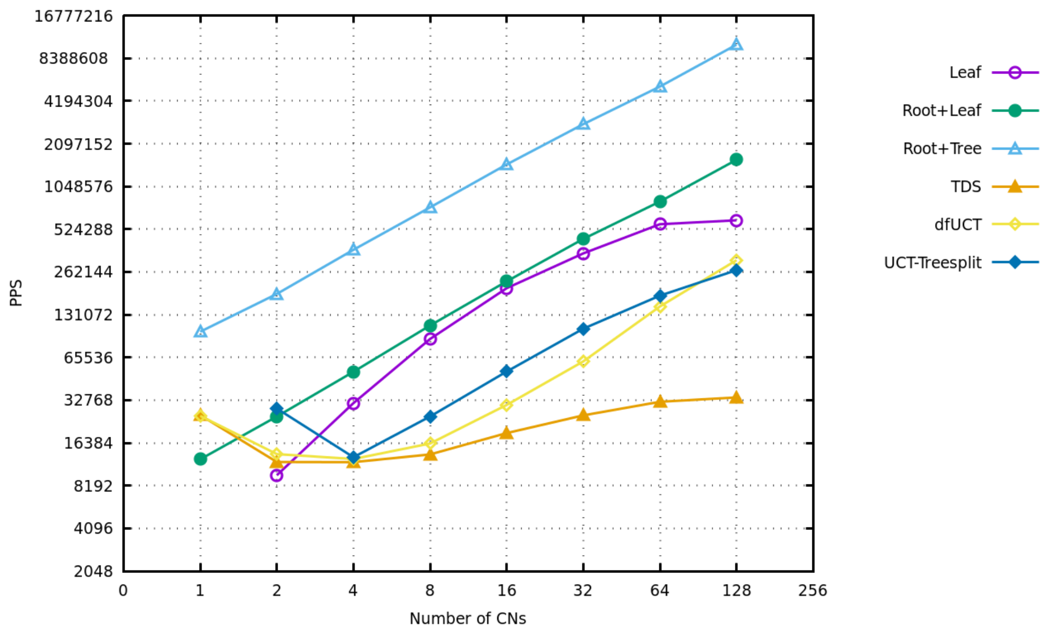

Figure 5.7: Playouts per second (PPS) achieved by our distributed MCTS agents with an increasing number of CNs.

The combination of root and tree parallelisation is dominant in terms of PPS. We believe that this is because root/tree parallelisation parallelises all four phases of MCTS, and all of the workers on a CN can perform playouts completely independently of each other (Section 4.4.2). This can be contrasted with root/leaf parallelisation, which only parallelises the simulation phase and consists of a single worker that must wait for simulation results before performing backpropagation, selection and expansion (Section 4.4.2).

The strong PPS scalability of root parallelisation can be contrasted with our TDS agent, which performs poorly in terms of raw PPS. As discussed in Section 3.4.1, TDS suffers from massive communication overhead incurred at the home-processors of near-root tree nodes, which we believe is the cause of the agent's limited scalability.

The improved PPS scalability of our df-UCT and UCT-Treesplit agents over TDS highlights the benefit of mitigating the aforementioned overhead. When comparing the overhead reduction strategies of UCT-Treesplit and df-UCT, we note that UCT-Treesplit has a slight advantage over df-UCT. Nevertheless, these agents perform relatively poorly in terms of PPS even though they all scale linearly after 4 CNs. Since the major difference between these agents and the root and leaf parallelisation agents is the massive number of messages passed over the network during the search, we believe that their relatively poor PPS performance can be attributed to one or more of the following reasons:

- Large message payloads causing serialization and de-serialization to take up time that would otherwise be used for searching
- Our choice of serialization framework being less performant than the MPI implementation that is widely used in the literature (Section 3.4)
- Akka's lack of RDMA support (also discussed in Section 5.2.3)

The theory that the relatively poor PPS scalability of distributed tree parallelisation can be attributed to a communication-related overhead is further supported by the dip in PPS achieved by these agents when a second `SearchRank` (Section 4.4) is introduced (2 CNs for TDS and df-UCT and 4 CNs for UCT-Treesplit). When there is more than one `SearchRank`, `SearchMessages` and `ReportMessages` must be transferred between `SearchRanks` in order to perform the search (Section 3.4), which we believe leads to this dip in PPS.

Finally, we note that our distributed leaf parallelisation agent scales well in terms of PPS even though it performs poorly with few CNs. With 2 CNs, leaf parallelisation consists of one traverser rank and one simulator rank (Section 4.4.1). This configuration only allows for a single sequential search, with the traverser rank being completely idle while it waits for simulation results from the simulator rank, leading to poor PPS. With 4 CNs, there are three simulator ranks and three parallel searches. Since the number of parallel searches is actually tripled with a doubling of CNs from 2 to 4, we see a steep increase in PPS here.

This behaviour continues until 16 CNs, at which point the increase in PPS with each doubling in CNs tapers off drastically. We believe that this decline in PPS improvement is caused by the traverser rank becoming a bottleneck because it only has 11 traversers available to perform more than 11 parallel searches (Section 4.4.1). We believe that adding more traverser ranks to the configuration when this bottleneck is encountered will improve the scalability of leaf parallelisation—testing this theory is a possible avenue for future work.

5.3.2 Tree Size

As discussed in Section 5.3.1, increasing the PPS achieved by an MCTS agent strengthens the statistics of nodes in the tree. This, in turn, should allow the agent to make better decisions during the search and better moves in a match. However, it is not always the case that increasing an agent's playout rate increases its playing strength [17]. Another potential indication of an agent's performance is the size of the tree that the agent constructs [62, 47].

For a standard serial implementation of MCTS where a single tree node is expanded per playout, the PPS achieved by an agent will be very similar to the number of nodes in the tree (there will be a slight difference due to the selection phase sometimes ending on a terminal tree node, which will prevent a new node from being expanded) (Section 2.2). However, this is not the case for parallel implementations since some compute entities will expand nodes that have already been explored by other compute entities, thereby incurring some search overhead (Section 2.2.4).

In order to measure and compare this overhead, we count the number of unique nodes expanded per second by each of our agents. We achieve this by introducing a Java `HashSet` at each `SearchRank` (Section 4.4) that contains the Zobrist hashes (Section 2.2.3.4) of states that the `SearchRank` has encountered during the search. The contents of these maps are periodically flushed and forwarded to a CN that is dedicated to counting the unique nodes encountered for the agent—the `NodeCounter`. Upon receiving a set of Zobrist hashes from each of the agent's CNs, the `NodeCounter` adds these sets to its own `HashSet` that is used to determine the number of unique nodes encountered across the agent's cluster. Although this process should introduce some communication overhead, we found it to be insignificant and we did not see any noticeable performance degradation with this feature enabled.

We depict the number of unique nodes expanded per second (NPS) by each of our implementations in Figure 5.8. If the same game state is encountered by multiple CNs, it is only counted once. We note that TDS, df-UCT and UCT-Treesplit scale far better in terms of NPS as opposed to PPS, which highlights the benefit of distributing a single search tree across the cluster as opposed to constructing multiple independent trees (as in root parallelisation) or dedicating a single CN to tree management (as in leaf parallelisation).

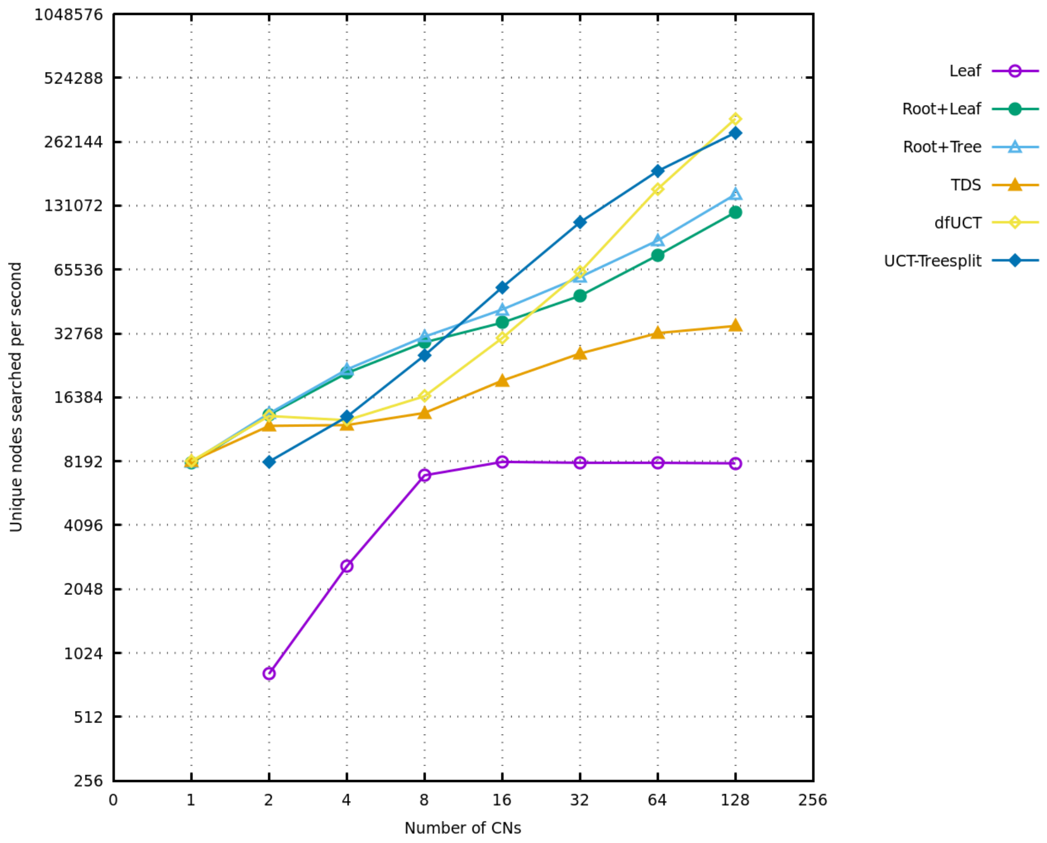

Figure 5.8: Unique nodes expanded per second by our distributed MCTS agents with an increasing number of CNs.

Root parallelisation scales worse than df-UCT and UCT-Treesplit in terms of NPS, with df-UCt and UCT-Treesplit achieving more NPS after 16 CNs even though root parallelisation is dominant when comparing PPS (Section 5.3.1). This can be attributed to the search overhead incurred by CNs constructing their trees in a similar fashion and thereby searching the same nodes, which in turn limits the number of unique nodes these agents encounter across the cluster. Although our experiments reveal somewhat limited NPS scalability for root parallelisation, we would have expected the NPS achieved by these agents to be even less than our results show. This indicates that the aforementioned search overhead does not have as much of an impact as we had initially assumed. Further experimentation with larger time constraints may reveal a more clear distinction between distributed tree parallelisation and root parallelisation. Performing such experiments and quantifying this search overhead is a possible avenue for future work.

As in our PPS graph (Figure 5.7), we see the limited scalability of TDS in comparison with df-UCT and UCT-Treesplit after 16 CNs, further highlighting the benefit of mitigating the communication overhead incurred by TDS at near-root tree nodes.

Finally, we note that leaf parallelisation performs particularly poorly in terms of NPS. It scales sharply up to 8 CNs then plateaus. As explained in Section 5.3.1, the performance of our leaf parallelisation implementation is inherently poor for few CNs. This effect is more noticeable when looking at NPS as opposed to PPS because the agent adds only one node to the tree for every 12 simulations it performs—one per `Simulator` (Section 4.4.1). The NPS scalability of leaf parallelisation flattens after 16 CNs because at this configuration, the `TraverserRank` begins to fill its transposition table each turn. We believe that this could be resolved by using configurations with more than one `TraverserRank`, and this is a possible avenue for future work.

5.3.3 Playing Strength

The ultimate goal of parallelising an MCTS algorithm is to take advantage of increased computational resources in order to make better decisions within some time constraint and produce stronger play. Although the PPS and NPS that an MCTS agent achieves is often used as a proxy metric to gauge playing strength, it is often the case that increasing PPS or NPS does not lead to stronger play [17]. In Sections 5.3.1 and 5.3.2, we presented the scalability of each of our implementations in terms of these proxy metrics. In this section, we present the playing strength scalability of each agent and compare these results with the PPS and NPS scalability presented earlier in the chapter. We depict the win-rate achieved by each of our agents versus our 1 CN root/tree reference opponent in Figure 5.9.

Firstly, we note that our root/tree agent has no discernable improvement in playing strength beyond 1 CN even though it achieves clear PPS and NPS scaling (see Figures 5.7 and 5.3.2, respectively). Nevertheless, root/tree parallelisation is our strongest implementation by a substantial margin. None of our other implementations were able to achieve a win-rate greater than 50% at any CN configuration versus the 1 CN root/tree agent. These two results indicate that our implementations do not benefit from an increase in PPS past the number achieved by our 1 CN root/tree agent. The reason for these diminishing returns is unclear, and further experimentation with different transposition table sizes and enhancement configurations will be needed in order to determine the root cause.

In contrast with root/tree, our root/leaf agent scales until 64 CNs, at which point it drops sharply at 128CNs. We believe that the relatively poor win-rates achieved by the 16 CN and 128 CN agents are due to chance. Although our root/leaf agent achieves very similar PPS as the reference opponent at 8 CNs and expands more nodes than the reference opponent at 2 CNs, it only achieves a near-50% win-rate at 32 CNs. A possible reason for this is that leaf parallelisation suffers more from the parallel effect than tree parallelisation. This assumption can be confirmed by comparing the degree to which each agent suffers from the parallel effect [56], and we leave this as a possible avenue for future work.

Our distributed leaf parallelisation scales well until 16 CNs, at which point it tapers off at a win-rate of approximately 50%. This behaviour is expected since this is the point at which the agent achieves the same PPS and NPS as our reference opponent. In addition to the aforementioned diminishing returns from increasing PPS, there are two reasons for the drop-off in scalability after 16 CNs:

1. This is the point at which the agent fills its transposition table every turn, so NPS does not increase at all after this point, and

2. PPS scaling starts to taper off after 16 CNs because the traverser rank becomes a bottleneck (see Section 2.2.4.1 for a more detailed explanation).

When compared with our other agents, distributed leaf parallelisation shows impressive playing strength scalability. We attribute the improved performance of our leaf parallelisation over previous implementations (Section 3.2) to the multi-threaded traverser. We believe that this performance could be further improved by using configurations with more than one multi-threaded traverser, and this is a possible avenue for future work.

Finally, we observe that our distributed tree parallelisation agents perform poorly in terms of playing strength. As mentioned in Section 5.2.3.2, UCT-Treesplit shows a dip in playing strength at 4 CNs due to the introduction of a second `SearchRank`, necessitating inter-CN communication to drive the search. This effect can also be seen for TDS and df-UCT for 2 CNs. Although TDS and UCT-Treesplit show scalability up to 64 CNs—similarly to the results presented in Section 5.2.3.2—they are only able to achieve a 20-40% win-rate at their strongest configurations. Although the differences in win-rate observed between the three distributed tree parallelisation agents might be a result of chance, we note that df-UCT's performance is poorer than TDS and UCT-Treesplit, with the agent achieving a win-rate of only 7-20% at its strongest configuration.

While the poor performance of TDS is expected since the agent never achieves more PPS or NPS than our reference opponent, we would expect to see df-UCT and UCT-Treesplit achieve a 50% win-rate at 32CNs since they achieve similar PPS and NPS as our reference opponent at this configuration.

While the seminal df-UCT paper did not present playing strength results [62], UCT-Treesplit has been shown to scale well in terms of playing strength [47]. We believe that the weakness of our implementation may stem from our choice to set the duplication and synchronisation parameters discussed in Section 3.4.3 to the values presented in the paper. It could be the case that, since our implementation does not perform as many playouts per second as their implementation, we are duplicating too few nodes and synchronising statistics too infrequently. This would lead to a communication overhead at near-root nodes and a search overhead, respectively, and may lead to the poor results shown in Figure 5.9. An important avenue for future work is to determine the effect that these parameters have on agent performance.

A possible explanation for the poor performance of df-UCT is the inherent search overhead incurred by delayed backpropagation (see Section 3.4.2). This could be verified by peforming experiments to determine the extent to which the agent suffers from the parallel effect [56]. Again we leave this as a possible direction for future work.

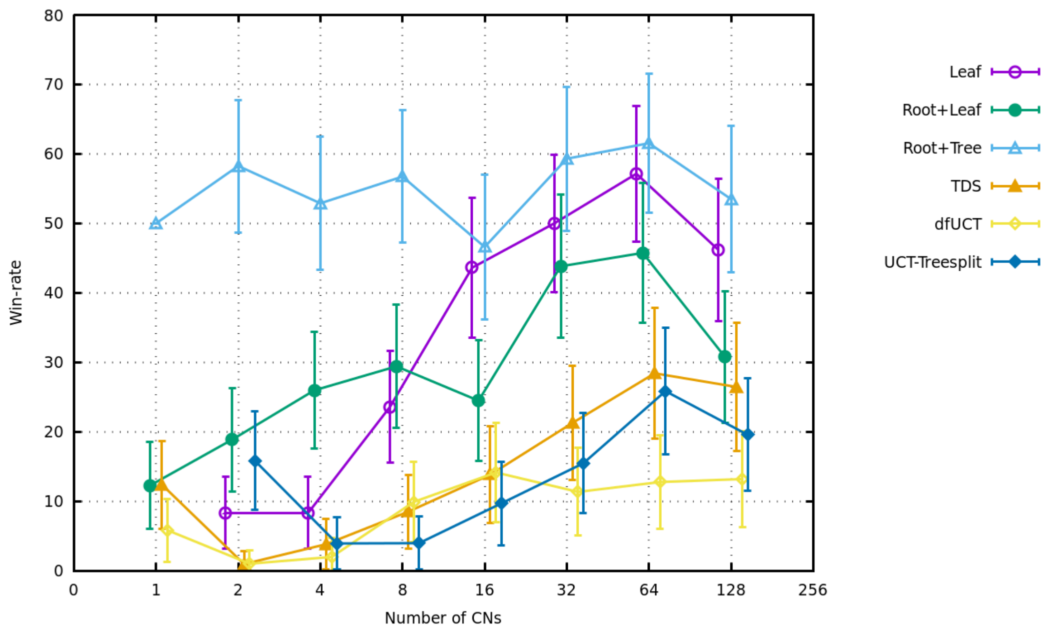

Figure 5.9: Win-rates achieved by our distributed MCTS implementations against our root/tree agent with 1CN. Each win-rate is slightly offset in order to improve legibility and provided with a 95% confidence interval.

5.4 Summary

In this chapter, we presented our experimental setup, parameter tuning procedure, and the scalability of our distributed MCTS implementations.

In Section 5.1, we presented our experimental setup. This included the number of matches we ran, the hardware we used, our reference opponent and the data logged by our system.

Section 5.2 discussed our parameter tuning process. We noted that we use a serial agent for MCTS enhancement tuning and present the results of these experiments in Sections 5.2.1 and 5.2.2. Following this, in Section 5.2.3, we discussed our procedure for tuning N_{par}. We began by using PPS as a proxy metric for playing strength in order to tune the parameter. Our initial results using the values obtained in Section 5.2.3.1 yielded poor results and we thus opted to tune N_{par} using playing strength in Section 5.2.3.2. To conclude the section, we provide reasoning for our choice of transposition table size in Section 5.2.4.

In Section 5.3, we presented and analysed the scalability of our leaf, root/leaf, root/tree, TDS, df-UCT and UCT-Treesplit implementations in terms of PPS, NPS and playing strength. PPS scalability was discussed in Section 5.3.1, with the root parallelisation agents showing dominance in terms of this metric and the tree parallelisation agents performing relatively poorly. Section 5.3.2 presented NPS scalability, and showed that distributed tree parallelisation outperforms root parallelisation in terms of unique nodes expanded. Finally, we presented playing strength scalability in Section 5.3.3. We noted that none of our agents consistently achieve a win-rate greater than 50% against a 1CN version of root/tree—not even root/tree with higher CN settings. We showed that leaf and root/leaf parallelisation scale well in terms of playing strength, and that our distributed tree parallelisation agents perform poorly even though they do show an increase in playing strength with an increase in CNs. We found our playing strength results for df-UCT and UCT-Treesplit to be particularly disappointing, and some ideas to develop our understanding of these results further were included.

Chapter 6

Conclusion

One way to improve the decision-making ability of an MCTS program is to maximise the number of simulations that the program performs. Parallelising MCTS therefore plays an important role in the development of stronger programs, especially since hardware that supports parallelism (multi-core CPUs, multi-CPU machines and large computer clusters) has become commonplace in recent years.

The three most widely implemented parallel MCTS algorithms—leaf parallelisation, root parallelisation and tree parallelisation—were initially proposed and analysed for SMP environments. Later in the development of parallel MCTS, these algorithms were successfully applied to large clusters of computers with distributed memory. However, since the implementations were tested on different domains and hardware setups, they were difficult to compare.

In light of this, we implemented the following distributed MCTS algorithms and compared their scalability in terms of playout rate, number of unique nodes expanded, and playing strength, using LOA as a common test domain:

- distributed leaf parallelisation,
- distributed root parallelisation combined with leaf parallelisation,
- distributed root parallelisation combined with tree parallelisation,
- distributed tree parallelisation using TDS,
- distributed tree parallelisation using df-UCT, and

- distributed tree parallelisation using UCT-Treesplit.

We implemented the first version of distributed leaf parallelisation with a multi-threaded traverser in order to address the bottleneck that has been discussed in previous publications. Additionally, we provide the first analysis of root parallelisation combined with leaf parallelisation. Other than these implementations, our distributed MCTS agents were implemented as described in the original literature.

In terms of PPS, we found that the root parallelisation-based agents are dominant. Root/tree parallelisation achieved approximately 2.5 times as many PPS as its closest competitor—root/leaf parallelisation—at all CN configurations. This, combined with the agent's exceptionally strong play (even at lower CN settings) leads us to recommend root/tree parallelisation for most distributed MCTS applications.

Root/leaf parallelisation was shown to scale similarly in terms of NPS as root/tree parallelisation. However, root/leaf parallelisation achieves far less PPS and shows weaker play than root/tree parallelisation. Therefore, we do not recommend combining root and leaf parallelisation for distributed environments when the combination of root and tree parallelisation is possible. However, we note that root/leaf parallelisation would be a good option in environments that do not support the shared mutable memory required for the lock-free tree parallelisation that we have implemented.

We saw that leaf parallelisation performs well in terms of PPS, and scales up to 128 CNs. However, we observed that, although its playing strength scalability was steep for lower CN settings, it was limited by the single `TraverserRank` that manages its tree. This is an expected result, and we believe that adding more than one `TraverserRank` will lead to much better results for leaf parallelisation. This would be an interesting avenue for future work.

We showed that all three distributed tree parallelisation implementations perform poorly in terms of PPS. Although df-UCT and UCT-Treesplit do alleviate the communication overhead incurred by TDS, our implementations are still limited due to network communication overhead—a problem which may be solved if RDMA is used. Although df-UCT and UCT-Treesplit show the best NPS scalability of our implementations, we cannot recommend either of these algorithms based on their disappointing playing strength results. We note that the poor performance of df-UCT and UCT-Treesplit ould be due to an inherent search overhead and infrequent duplication and synchronisation of node statistics, respectively. Performing a deeper analysis into the weakness of these algorithms is an important avenue for future work.

In addition to our scalability experiments, we analysed the effect of the N_{par} parameter on the PPS and playing strength scalability. We found that higher N_{par} invariably leads to more PPS until the agent is no longer able to begin searching due to an initial overload of `SearchMessages`. However, we determined that setting N_{par} to the maximum feasible value does not yield good results, with the playing strength scalability being limited by the influence of the parallel effect.

6.1 Future Work

While we were able to provide a fair comparison of distributed MCTS algorithms on a common domain, we identified the following avenues for future work that may yield better results:

- In order to explain the poor playing strength scalability achieved by distributed tree parallelisation, we propose that experiments are run in order to determine the extent to which the agents are influenced by the parallel effect. This can be achieved by keeping the number of playouts performed constant while increasing the number of CNs available to each agent. In doing so, one should see a decline in win-rate with an increase in CNs that will quantify the parallel effect.

- The parameters that determine the number of duplicated nodes and the frequency that those nodes' statistics are synchronised should be tuned for our implementation. We believe that a reason for the poor scaling of UCT-Treesplit may be the choice to set these parameters to the values presented in the original paper. Since our implementation achieves fewer playouts per second, it is possible that there are too few duplicated nodes and infrequent synchronisation, leading the agent to incur a communication overhead and search overhead, respectively.

- Our leaf parallelisation implementation was shown to scale well with a parallel `TraverserRank` until the rank's transposition table becomes full and the single rank becomes a bottleneck. This would be resolved by having more than one `TraverserRank`. Ideally, the `TraverserRanks` should share node statistics in a similar fashion to root parallelisation. The implementation and analysis of such a leaf parallelisation – root parallelisation hybrid should yield interesting results.

- It may be beneficial to tune the parameters that drive the MCTS enhancements we used in this work for each distributed MCTS algorithm. We only tuned these parameters for a serial agent, but there may be better options for the distributed implementations. It would be interesting to determine how each algorithm reacts to adjustments in these parameters, and if tuning for each algorithm individually is necessary at all. Additionally, it may be helpful to consider the behaviour of these algorithms with all enhancements turned off.

6.2 Final Remarks

This research aimed to provide a fair comparison of distributed MCTS algorithms on the same domain and hardware setup. Therefore, we performed experiments to determine the scalability of distributed leaf, root/leaf, root/tree, TDS, df-UCT and UCT-Treesplit in terms of PPS, NPS and playing strength.

We found that distributed root/tree parallelisation is the strongest of our implementations, and recommend this algorithm for most distributed MCTS implementations. We cannot recommend any of the distributed tree parallelisation algorithms due to their disappointing playing strength results. However, we note that further analysis must be performed in order to pinpoint the reason for these poor results. Finally, distributed leaf parallelisation showed impressive scalability due to our multi-threaded traverser implementation. However, we show that a single `TraverserRank` is not sufficient for scaling to higher CN settings, and further experimentation with more than one `TraverserRank` should yield interesting results.

Appendix A

Move Categories

Table A.1: Transition probabilities for the move categories described in Section 4.1.3.2

From Region	To Region	Capture	No Capture
A	A	0.0000	0.0000
A	B	0.1975	0.0546
A	C	0.3077	0.2233
A	D	0.3833	0.1772
B	A	0.0000	0.0052
B	B	0.0245	0.0163
B	C	0.2471	0.0453
B	D	0.2665	0.0686
C	A	0.0000	0.0005
C	B	0.0142	0.0051
C	C	0.1429	0.0231
C	D	0.2164	0.0430
D	A	0.0000	0.0011
D	B	0.0180	0.0035
D	C	0.1275	0.0146
D	D	0.0000	0.0339

Bibliography

[1] CHPC wiki. `http://wiki.chpc.ac.za/start`, 2019.

[2] Allen, J. *Effective Akka: Patterns and Best Practices.* " O'Reilly Media, Inc.", 2013.

[3] Anderson, T. E. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems 1*, 1 (1990), 6–16.

[4] Arneson, B., Hayward, R. B., and Henderson, P. Monte Carlo tree search in Hex. *IEEE Transactions on Computational Intelligence and AI in Games 2*, 4 (2010), 251–258.

[5] Auer, P., Cesa-Bianchi, N., and Fischer, P. Finite-time analysis of the multiarmed bandit problem. *Machine Learning 47*, 2-3 (2002), 235–256.

[6] Billings, D. Mona and yl's lines of action page. `https://webdocs.cs.ualberta.ca/~darse/LOA/`, 2002.

[7] Bonér, J., Klang, V., and Kuhn, R. Akka library. *http://akka. io*.

[8] Bourki, A., Chaslot, G., Coulm, M., Danjean, V., Doghmen, H., Hoock, J.-B., Hérault, T., Rimmel, A., Teytaud, F., and Teytaud, O. Scalability and parallelization of Monte-Carlo tree search. In *International Conference on Computers and Games* (2010), Springer, pp. 48–58.

[9] Breuker, D. M., Uiterwijk, J. W., and van den Herik, H. J. Replacement schemes for transposition tables. *ICGA Journal 17*, 4 (1994), 183–193.

[10] Brockington, M. G., and Schaeffer, J. Aphid game-tree search. *Advances in Computer Chess 8* (1997), 69–91.

[11] Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games 4*, 1 (2012), 1–43.

[12] Brudno, A. L. Bounds and valuations for abridging the search of estimates. *Problems of Cybernetics 10* (1963), 225–241.

[13] Bump, D. GNU Go home page, 2003.

[14] Cazenave, T., and Jouandeau, N. On the parallelization of UCT. In *proceedings of the Computer Games Workshop* (2007), pp. 93–101.

[15] Cazenave, T., and Jouandeau, N. A parallel Monte-Carlo tree search algorithm. In *Computers and Games* (Berlin, Heidelberg, 2008), H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands, Eds., Springer Berlin Heidelberg, pp. 72–80.

[16] Chaslot, G., Bakkes, S., Szita, I., and Spronck, P. Monte-Carlo tree search: A new framework for game AI. In *AIIDE* (2008).

[17] Chaslot, G. M.-B., Winands, M. H., and van Den Herik, H. J. Parallel Monte-Carlo tree search. In *International Conference on Computers and Games* (2008), Springer, pp. 60–71.

[18] Chaslot, G. M. J., Winands, M. H., HERIK, H. J. V. D., Uiterwijk, J. W., and Bouzy, B. Progressive strategies for Monte-Carlo tree search. *New Mathematics and Natural Computation 4*, 03 (2008), 343–357.

[19] Chaslot, I. S. G., Winands, M. H., and Van Den Herik, H. J. Parameter tuning by the cross-entropy method. In *European workshop on reinforcement learning* (2008), vol. 2008, p. 7.

[20] Coulom, R. Crazy stone. `https://www.remi-coulom.fr/CrazyStone/`.

[21] Coulom, R. Efficient selectivity and backup operators in Monte-Carlo tree search. In *International conference on computers and games* (2006), Springer, pp. 72–83.

[22] Coulom, R. Clop: Confident local optimization for noisy black-box parameter tuning. In *Advances in Computer Games* (2011), Springer, pp. 146–157.

[23] Enzenberger, M., and Müller, M. A lock-free multithreaded Monte-Carlo tree search algorithm. In *Advances in Computer Games* (2009), Springer, pp. 14–20.

[24] Feldmann, R., Mysliwietz, P., and Monien, B. Game tree search on massively parallel systems. *Advances in Computer Chess 7* (1993).

[25] Gelly, S., Hoock, J.-B., Rimmel, A., Teytaud, O., and Kalemkarian, Y. On the parallelization of Monte-Carlo planning. In *ICINCO* (2008).

[26] Gelly, S., and Silver, D. Combining online and offline knowledge in UCT. In *Proceedings of the 24th international conference on Machine learning* (2007), ACM, pp. 273–280.

[27] Gelly, S., and Wang, Y. Exploration exploitation in Go: UCT for Monte-Carlo Go. In *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop* (2006).

[28] Hearn, R. A., and Demaine, E. D. *Games, puzzles, and computation*. CRC Press, 2009.

[29] Hewitt, C. Actor model of computation. *arXiv preprint arXiv:1008.1459* (2010).

[30] Ikeda, K., and Viennot, S. Efficiency of static knowledge bias in Monte-Carlo tree search. In *International Conference on Computers and Games* (2013), Springer, pp. 26–38.

[31] Kato, H., and Takeuchi, I. Parallel Monte-Carlo tree search with simulation servers. In *International Conference on Technologies and Applications of Artificial Intelligence* (2010), pp. 491–498.

[32] Kishimoto, A., and Schaeffer, J. Transposition table driven work scheduling in distributed game-tree search. In *Conference of the Canadian Society for Computational Studies of Intelligence* (2002), Springer, pp. 56–68.

[33] Knuth, D. E., and Moore, R. W. An analysis of alpha-beta pruning. *Artificial intelligence 6*, 4 (1975), 293–326.

[34] Kocsis, L., and Szepesvári, C. Bandit based Monte-carlo planning. In *European conference on machine learning* (2006), Springer, pp. 282–293.

[35] Lai, T. L., and Robbins, H. Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics 6*, 1 (1985), 4–22.

[36] Marsland, T. A., and Popowich, F. Parallel game-tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 4 (1985), 442–452.

[37] Metropolis, N. Monte carlo method. *From Cardinals to Chaos: Reflection on the Life and Legacy of Stanislaw Ulam* (1989), 125.

[38] Minsky, M., and Papert, S. A. *Perceptrons: An introduction to computational geometry*. MIT press, 2017.

[39] Moyer, C. How Google's AlphaGo beat a Go world champion. *The Atlantic 28* (2016).

[40] Müller, M. Computer Go. *Artificial Intelligence 134*, 1-2 (2002), 145–179.

[41] Nilsson, N. J. *Principles of artificial intelligence*. Morgan Kaufmann, 2014.

[42] Pearl, J. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Pub. Co., Inc., Reading, MA, 1984.

[43] Piccione, P. A. *In Search of the Meaning of Senet*. Archaeological Institute of America, 1980.

[44] Romein, J. W., Plaat, A., Bal, H. E., and Schaeffer, J. Transposition table driven work scheduling in distributed search. In *AAAI/IAAI* (1999), pp. 725–731.

[45] Russell, S. J., and Norvig, P. *Artificial Intelligence: A Modern Approach*. Malaysia; Pearson Education Limited, 2016.

[46] Sackson, S. *A gamut of games*. Courier Corporation, 1992.

[47] Schaefers, L., and Platzner, M. Distributed monte carlo tree search: A novel technique and its application to computer Go. *IEEE Transactions on Computational Intelligence and AI in Games 7*, 4 (2015), 361–374.

[48] Schaeffer, J. Distributed game-tree searching. *Journal of Parallel and Distributed Computing 6*, 1 (1989), 90–114.

[49] Schäfers, L. *Parallel Monte-Carlo Tree Search for HPC Systems and its Application to Computer Go*. Logos Verlag Berlin GmbH, 2014.

[50] Shannon, C. E. Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science 41*, 314 (1950), 256–275.

[51] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., and Lanctot, M. Mastering the game of Go with deep neural networks and tree search. *Nature 529*, 7587 (2016), 484–489.

[52] Slate, D. J., and Atkin, L. R. Chess 4.5—the northwestern university chess program. In *Chess skill in Man and Machine*. Springer, 1983, pp. 82–118.

[53] Smith, S. J., and Nau, D. S. An analysis of forward pruning. In *AAAI* (1994), pp. 1386–1391.

[54] Soejima, Y., Kishimoto, A., and Watanabe, O. Evaluating root parallelization in Go. *IEEE Transactions on Computational Intelligence and AI in Games 2*, 4 (2010), 278–287.

[55] Teytaud, F., and Teytaud, O. On the huge benefit of decisive moves in Monte-Carlo tree search algorithms. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games* (2010), IEEE, pp. 359–364.

[56] Van Niekerk, F., van Rooyen, G.-J., Kroon, S., and Inggs, C. P. Monte-Carlo tree search parallelisation for computer Go. In *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference* (2012), ACM, pp. 129–138.

[57] Wang, Y., and Gelly, S. Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In *2007 IEEE Symposium on Computational Intelligence and Games* (2007), IEEE, pp. 175–182.

[58] Winands, M. H. Analysis and implementation of lines of action. *Master's thesis, Department of Computer Science, Universiteit Maastricht* (2000).

[59] Winands, M. H., Bjornsson, Y., and Saito, J.-T. Monte carlo tree search in lines of action. *IEEE Transactions on Computational Intelligence and AI in Games 2*, 4 (2010), 239–250.

[60] Winands, M. H., Uiterwijk, J. W., and van den Herik, H. J. The quad heuristic in lines of action. *ICGA Journal 24*, 1 (2001), 3–15.

[61] Winands, M. H., van den Herik, H. J., and Uiterwijk, J. W. An evaluation function for lines of action. In *Advances in Computer Games*. Springer, 2004, pp. 249–260.

[62] Yoshizoe, K., Kishimoto, A., Kaneko, T., Yoshimoto, H., and Ishikawa, Y. Scalable distributed Monte-Carlo tree search. In *Fourth Annual Symposium on Combinatorial Search* (2011).

[63] Zobrist, A. L. A new hashing method with application for game playing. *ICCA journal 13*, 2 (1970), 69–73.