UNIVERSITEIT•STELLENBOSCH•UNIVERSITY
jou kennisvennoot • your knowledge partner

MCTS Parallelisation

Author:
Francois van Niekerk

Supervisors:
Dr. Gert-Jan van Rooyen
Dr. Steve Kroon
Dr. Cornelia Inggs

Report submitted in partial fulfilment of the requirements of the module Project (E) 448 for the degree Baccalaureus in Engineering in the Department of Electrical and Electronic Engineering at the University of Stellenbosch.

October 2011

Abstract

Modern processors are making increasing use of parallel hardware to increase processing power. This has given increasing importance to parallelisation of computationally expensive algorithms, such as Monte-Carlo Tree Search (MCTS), in order to make use of the parallel hardware.

This project implemented pondering and parallelised an existing MCTS implementation for Computer Go on multi-core and cluster systems. Tree parallelisation was used for multi-core systems and root parallelisation was used for cluster systems.

Pondering functioned as designed and provided a strength increase in the order of 60 Elo. Multi-core parallelisation scaled close to perfectly on the tested hardware on 9x9 and 19x19 with virtual loss. Cluster parallelisation showed disappointing results on 9x9, but succeeded to scale well on 19x19 were it achieved a four core ideal strength increase on eight cores.

This project has implemented pondering and parallelisation techniques that enable an MCTS implementation to take advantage of parallel hardware.

Uittreksel

Moderne verwerkers maak toenemende gebruik van paralelle hardeware om verwerkingskrag te verhoog. Dit het toenemende belang vir die paralellisasie van computationeel duur algoritmes, soos die *Monte-Carlo Tree Search* (MCTS) gebring, met die doel om van paralelle hardeware gebruik te maak.

Hierdie projek het *pondering* en die paralellisasie van 'n bestaande MCTS implementering vir Rekenaar Go op *multi-core* en *cluster* stelsels geïmplementeer. *Tree* paralellisasie is wat gebruik word vir *multi-core* stelsels en *root* paralellisasie is wat gebruik word vir *cluster* stelsels.

Pondering funksioneer soos ontwerp en 'n krag toename in die orde van 60 Elo is gemeet. *Multi-core* paralellisasie skaal naby perfek op, op die toets hardeware op 9x9 en 19x19 met *virtual loss*. *Cluster* paralellisasie het teleurstellende resultate op 9x9 getoon, maar het goed op geskaal op 19x19 waar dit 'n vier *core* ideale krag toename op agt *cores* bereik.

Hierdie projek het *pondering* en paralellisasie tegnieke geïmplementeer wat 'n MCTS implementering in staat stel om die voordele van paralelle hardeware te trek.

Acknowledgements

I would like to express my gratitude to the following people for their support and help during the course of this project; without them this project would not be possible:

- All three of my supervisors, for their many hours of constant help and guidance.
- Hilgard Bell, for listening to and giving constructive criticisms on my rants.
- Ursula Straub, for putting up with my overly-complex explanations and mumblings.
- All my other friends and family, for their untiring support.
- The Computer Go community, for the valuable trove of information and ideas.
- Heine de Jager, for his support with my problems on the Rhasatsha cluster.

This material is based upon work supported financially by the National Research Foundation of South Africa.

Declaration

I, the undersigned, hereby declare that the work contained in this report is my own original work unless indicated otherwise.

Signature: .....
F. van Niekerk

Date: .....

Contents

List of Figures

List of Tables

List of Algorithms

Nomenclature

Acronyms

CHPC	Centre for High Performance Computing.
CPU	Central Processing Unit.
ECSA	Engineering Council of South Africa.
GGP	General Game Playing.
GPGPU	General Purpose Graphical Processing Unit.
GTP	Go Text Protocol.
HPC	High Performance Computing.
LGRF	Last Good Reply with Forgetting.
MCTS	Monte-Carlo Tree Search.
MPI	Message Passing Interface.
RAVE	Rapid Action Value Estimation.
SMP	Symmetric Multi-Processing.
SUN	Stellenbosch University.
TCP	Transmission Control Protocol.
UCT	Upper Confidence Bounds applied to Trees.

Terms

Elo	System used for quantifying the strength of Go players.
Oakfoam	MCTS Go implementation used as a base.
Rhasatsha	Stellenbosch Cluster.

Chapter 1

Introduction

> "If there are sentient beings on other planets, then they play Go."
>
> — Emanuel Lasker, Chess World Champion

Due to physical constraints, modern processors are making increasing use of parallel hardware to increase processing power [1]. This has given increasing importance to parallelisation of computationally expensive algorithms, such as Monte-Carlo Tree Search (MCTS) [2], in order to make use of the parallel hardware.

The main aim of this project is the parallelisation of MCTS, as used for Computer Go [3]. This parallelisation will be aimed at multi-core and cluster systems. In addition, pondering (thinking during the opponent's thinking time) will be investigated. This project will allow an MCTS implementation to take advantage of today and tomorrow's increasingly parallel hardware.

1.1 The Game of Go

Go (otherwise known as Weiqi, Baduk, and Igo) is an ancient game in which two players alternate in placing black and white stones on empty intersection of a board [4]. Orthogonally adjacent stones of the same colour form chains and chains that have no orthogonally adjacent empty intersections are removed from the board. The winner is the player whose stones control the largest area at the end of the game. Go can be played on different board sizes, with the most popular being 19x19 [5]. Figure 1.1 shows a small Go board with a number of moves already played. The △ stone only has one empty adjacent intersection, to the right. If black plays on that empty intersection, the △ stone will be removed from the board. In the top left, white is controlling two intersections. Black cannot play on either of these intersections as the black stone would immediately

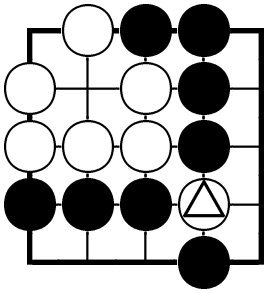


Figure 1.1: Small Go board

be removed, and such suicide moves are forbidden. Those white stones can therefore not be removed from the board by black and are said to be *alive*. By making groups of stones that are alive, a player is able to control sections of the board.

Go rules can be stated in a simple and elegant manner, yet they lead to complex situations (see Appendix E for a complete set of rules). This emergent complexity is what makes Go enjoyable for many Go players, but also contributes to Computer Go being so difficult [6].

To play Go at a non-trivial level, humans can create tree-like structures in their minds [7]. These trees consist of board positions as nodes in the tree, with children of a node being positions that occur after valid moves. The act of forming and evaluating such a tree for Go is called "reading" [7]. Due to the additive nature of Go — pieces are generally added to the current board position, not moved — it is relatively easy for humans to read ahead [7], compared to games like Chess, in which pieces move around. Professional Go players have been known to read up to 30 moves ahead in a complex situation in the middle of a game and further ahead closer to the end of a game [8].

1.2 Computer Go

Computer Go is the research field concerned with using computer programs to play Go. In the past, Computer Go programs could be beaten by most weak amateurs, but recently Computer Go has taken a number of large steps forward and can now compete with high level amateurs on 19x19 board sizes and professionals on 9x9 [9]. Much of this recent advancement is due to MCTS and related work [9].

One of the advantages of MCTS over previous techniques for Computer Go is that MCTS can make more effective use of more powerful processing power by using parallelisation [10]. This project focuses on parallelisation of MCTS.

There are three major methods of parallelisation for MCTS [11]. This project will make use of two of these these methods to parallelise an existing MCTS implementation, namely tree parallelisation for multi-core systems and root parallelisation for cluster systems. Leaf parallelisation, the other major parallelisation method, will not be used in this project.

1.3 Objectives

This project has a number of objectives which it accomplishes, all of which revolve around extending an existing MCTS implementation. The objectives are to make the following extensions: pondering (thinking during opponent thinking time) to make use of all available time; and parallelisation on multi-core and cluster systems to make use of parallel hardware. These objectives will be designed and implemented prior to being extensively tested and evaluated.

1.4 Outline

The rest of this report is structured as follows: Chapter 2 describes the state of the art of Computer Go, focusing on parallelisation. In Chapter 3, the design of pondering and parallelisation solutions are presented. In Chapter 4, the implementation of pondering, multi-core parallelisation and cluster parallelisation is described. In Chapter 5, a number of pondering and parallelisation tests are performed, and their results are presented and analysed. Finally, Chapter 6 provides a conclusion and overview of the pondering and parallelisation that this project has focused on.

Chapter 2

Background and Related Work

2.1 Introduction

Computer Go is the field of using computer programs to play the game of Go. In a number of other games, such as Chess and Othello, computers have surpassed human players [6]. However, Computer Go has not yet achieved the same dominance experienced in those games [9]. This is partially due to the complexity of Go — the branching factor of Go is in the order of 100 on a 19x19 board for most of the game, whereas the branching factor of Chess is closer to 20 [6].

"Traditional" Computer Go algorithms have tried to replicate the thought process that humans use. This was not very successful, as the amount of expert knowledge that must be hand-coded and maintained quickly grows too large [12]. In the last decade, Monte-Carlo Tree Search has found great popularity in the Computer Go field and is currently the dominant algorithm [9].

2.2 Go Strength

As with most games, there is no absolute rating for playing strength that can be assigned to a Go player based on their Go knowledge. Rather, ratings are awarded to Go players based on their performance in previous games. However, various systems that attempt to measure Go strength do exist.

One system that is used to quantify strength in Go, as well as other games such as Chess, is the Elo system [13]. This system uses a single number, an Elo score, to represent the strength of a player. The details of this system were presented in [14]. A useful property of the Elo system is that, given the difference in Elo scores between two players, the probability of either player winning a game between them can be determined. The inverse is also true: given a winning probability between two players, an Elo difference can be determined. A possible scenario is: if a player with an Elo score of 1000 plays

against a player with an Elo score of 900, the difference of 100 Elo points indicates that the former player should win 64% of the time (see Table G.1 in Appendix G). Elo score differences are often used in Computer Go to quantify the difference in strength between two players as they are able to measure much large differences more accurately than winning percentages [13]. Furthermore, Elo differences are relative and therefore useful for comparing results from different sources. A table of winning probabilities for given Elo differences is given in Appendix G.

The kyu-dan ranking system is another system that is often used to describe Go strength [15]. In this system amateur players are assigned a rank in the range 30 kyu[1], 29 kyu, ..., 1 kyu, 1 dan, 2 dan, ..., 7 dan (in ascending order of strength). A separate scale for professional players starts after amateur 7 dan. The ranking system used in martial arts uses the same terms and structure [15]. This system is widely used amongst human Go players [15], however it is very coarse, with only single rank granularity, while the Elo system can have arbitrary precision. The Elo system was therefore preferred in this project. For comparison between the two systems, a difference of 100 Elo points is approximately equivalent to the difference between successive kyu or dan ranks (this comparison is an approximation and does not apply at the extremes) [13].

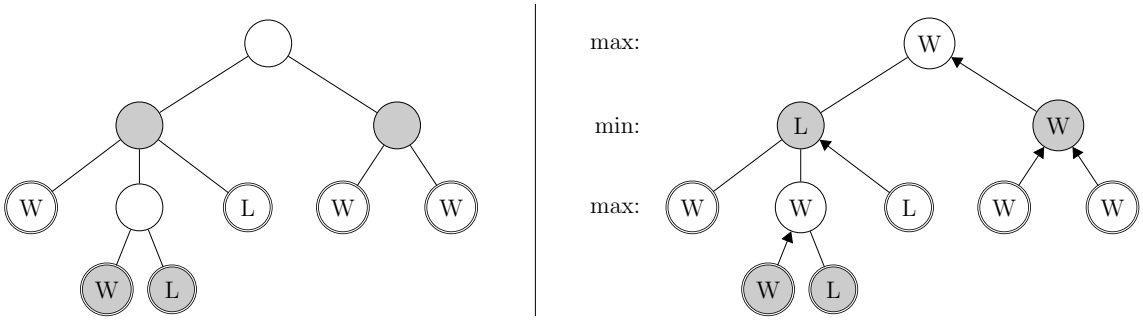
In a game of Go between two players of different levels of strength, they can make use of a handicap in the form of handicap stones or komi (a number of additional points awarded to the white player) to equal the odds [16]. Komi is frequently used to reduce the advantage of moving first, but handicap stones are typically not made use of in competitive play, but frequently used in casual play. All games played in the testing of this project were without a handicap, unless otherwise noted.

2.3 Game Trees

Game trees are tree data structures, with nodes representing positions and edges representing moves that lead to other positions [17]. Game trees can also have additional information stored at nodes, such as a winner or evaluation score. In this way, game trees can be used to perform searches for moves.

Minimax is one such search algorithm [18]. At each level of the tree, the value of the node alternates between the minimum and maximum value of the node's children [18]. The alternating between minimum and maximum values ensures that the optimal strategy for both players is chosen. Figure 2.1a shows a small game tree with the terminal nodes marked and the win status indicated. Figure 2.1b shows the same tree after the minimax algorithm has been applied. It is now shown that the optimal play is the move that arrives at the position on the right of the tree, whereas the other move will lead to a lost game if the opponent plays optimally.

[1]The weakest kyu rank varies, depending on the implementation of the system.



(a) Before Minimax **(b)** After Minimax

Figure 2.1: Example of minimax algorithm. Node values are wins (W) or losses (L) and are all from the perspective of one player. Shaded nodes indicate that the opponent will play from this position.

A variation on minimax is the *negamax* algorithm [19]. With negamax, every tree node's value is from the perspective of the player to move in that position. This implies that the value of a node in a negamax tree is the negative value of the maximum child [19]. Negamax is used as it simplifies a number of implementation details. This project makes use of negamax in the implementation, but this report makes use of minimax in some explanations to make them more elegant and understandable.

Game trees are used in computer players, for games such as Go, to select moves. A complete game tree (one that contains all possible game sequences) will reveal perfect play after the minimax or negamax algorithm is used on it [18, 19]. However, this will require too much memory and processing power for most games, including Go and Chess, so minimax and negamax are typically not used on a complete tree in practice [6]. Instead, techniques such as alpha-beta pruning are used to reduce the computational resources required and make the tree search feasible [18]. However, partially due to the huge branching factor of Go, alpha-beta pruning is not enough to create a strong Go player [12].

2.4 Computer Go Techniques

Computer Go is the research field concerned with making computers play Go. Typically, this is done with the use of a game tree and algorithms that process the tree and perform a tree search on the tree [12]. In order to perform a strong tree search, a good evaluation function is required. This is where the core difficulty of Computer Go lies — the creation of an evaluation function for a Go position is a notoriously difficult task [12].

Various programs have attempted to evaluate a position in a human-like way, but this requires extensive expert knowledge that is very difficult to maintain and improve on [12]. These programs have not been successful in surpassing a novice amateur level [12]. Other techniques such as neural networks, have shown some interesting results, but have also not been very successful [12].

2.5 Monte-Carlo Tree Search

Monte-Carlo simulations are stochastic simulations of a model [20]. Through repetition, they can provide valuable information and they are especially useful for problems which do not have a known deterministic solution [20].

Monte-Carlo simulations were first applied to Go in the form of simulated annealing [21, 22]. Simulated annealing is a process whereby a system of pieces have an initial temperature that dictates "movement" and are slowly "cooled down" until they arrive in a local optimum, like a liquid cooling and forming a crystal [23]. In this case, a sequence of Go moves was "cooled down" and managed to achieve a strength above random play without any expert knowledge [21, 22].

Later, Monte-Carlo simulations were again applied to Go, but in a different form [24]. These simulations, often referred to as *playouts*, simulate a game of Go from some initial position until the end of the game, by selecting player moves at random[2] [24]. Once the end of the game is reached, it is trivial to score and determine the winner. A number of playouts are performed, starting from the same position and the ratio of wins to losses forms an evaluation of that position.

Even though these playouts only make use of the rules of Go to select moves, through repetition they are able to provide valuable information regarding which moves are better [2, 24].

In order to improve the strength of these Monte-Carlo methods, they were combined with game tree search to form Monte-Carlo Tree Search (MCTS) [2]. An early application of MCTS was in Computer Go [2]. Since its inception, MCTS has enjoyed widespread use, especially in the field of General Game Playing (GGP) [25].

The MCTS tree is initialised with the current game position as the root node and all nodes in this tree will store a number of wins and losses [2]. Figure 2.2 shows an example MCTS tree with each node showing the results of playouts through that node. All leaf nodes and nodes that can add another valid child[3] form the *frontier* of the MCTS tree.

The MCTS algorithm can be broken-down into the following four sequential parts [26]:

1. Selection
2. Expansion
3. Simulation
4. Backpropagation

Selection is the process of descending down the tree (see Figure 2.3a). Each node through which the descent travels is selected according to a selection policy. This policy's main

[2]More advanced playouts make use of heuristics to select moves [9].
[3]There is a finite number of valid Go moves for any given position.

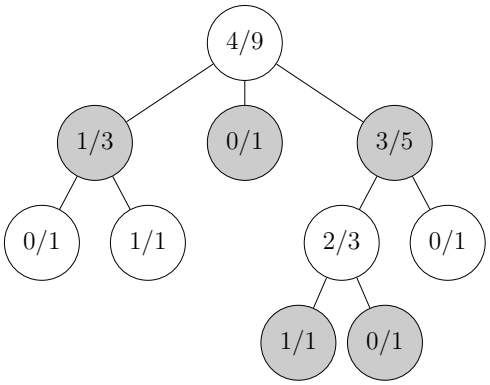


Figure 2.2: Example MCTS tree. All nodes show the number of playout wins over the total number of playouts through that node from the perspective of one player. Shaded nodes indicate the opponent will play next from this position.

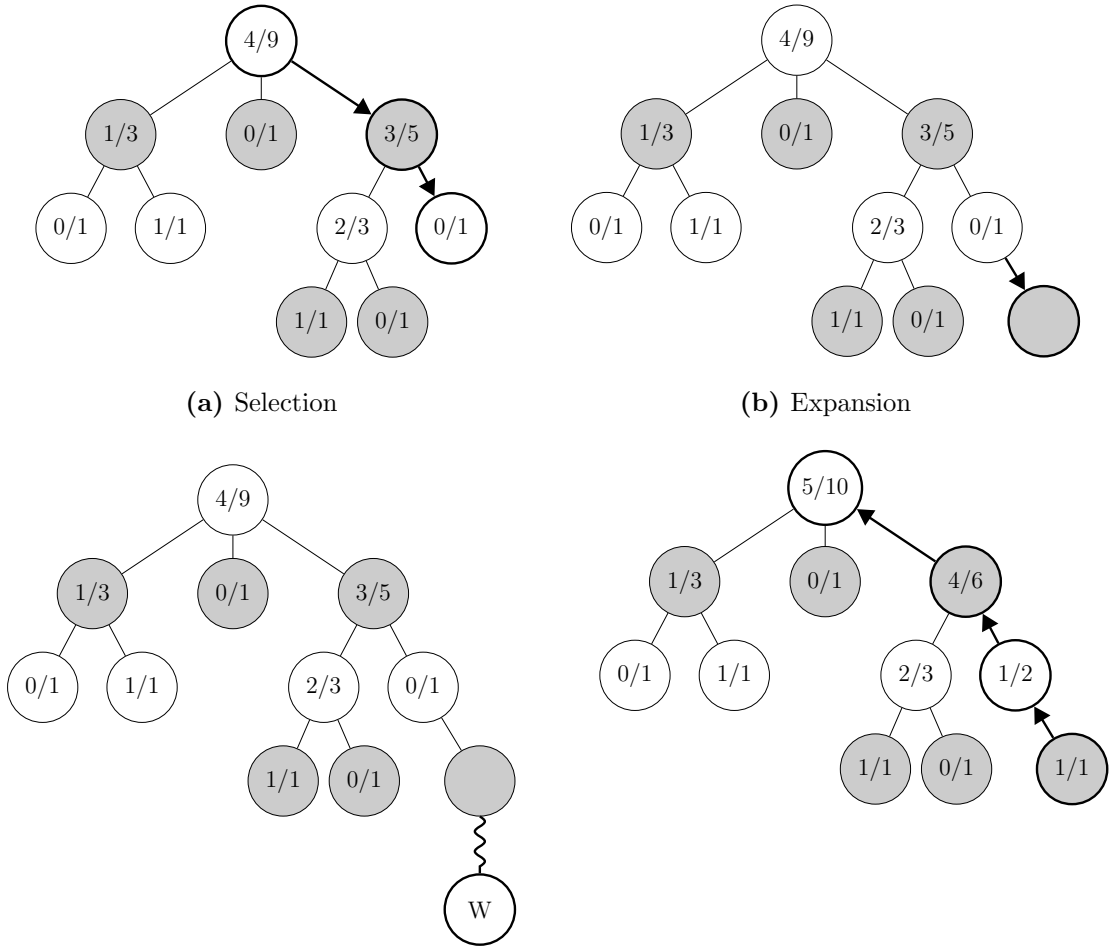
aim is to reduce *regret* (the time spent considering non-optimal moves) and this involves trading off exploration versus exploitation. Exploitation is the act of focusing on the currently best node, while exploration is the act of considering other, currently worse (but possibly ultimately better) nodes. The most commonly used selection policy for Computer Go is Upper Confidence Bounds (UCB) or a variant on it. According to one variant, UCB1, the node that maximises Equation 2.1 [2] is selected. This equation is made up of two terms: the first term is the winrate of the node, while the second term is an approximation of the upper bound on a confidence interval. In other words, maximising this equation selects the node that is possibly the ultimately best. The c constant is selected and tuned to optimise the balance between exploration and exploitation (a larger c creates more exploration). The variant of MCTS that uses UCB is often referred to as "Upper Confidence Bounds applied to Trees" (UCT).

$$\text{urgency(node)} = \frac{\text{wins(node)}}{\text{playouts(node)}} + c \cdot \sqrt{\frac{\log(\text{playouts(parent)})}{\text{playouts(node)}}} \tag{2.1}$$

Once the selection has arrived at a node, *expansion* is the process of adding a new node to a frontier node of the tree (see Figure 2.3b). In this way, the frontier of the tree is constantly expanding, looking "deeper" into the future. Expansion increases the accuracy and relevance of the tree by making it a more realistic representation of possible outcomes. Once a node has been added to the tree, *simulation* is the process of performing a playout starting from this node (see Figure 2.3c).

After the playout is finished and the result is available, *backpropagation* is the process of propagating this result from the initial playout position's node, back up the tree to the root (see Figure 2.3d). Typically the new result is combined with previous results such that all results through that node have equal weight, but other "backup operators" can be considered [27].

The process of descending through a node and later adding a result to that node is



(a) Selection **(b)** Expansion

(c) Simulation **(d)** Backpropagation

Figure 2.3: Example of MCTS algorithm showing selection, expansion, simulation and backpropagation. All node results are from the perspective of one player.

similar to sampling from a distribution. However, as the tree is expanded, the distribution being sampled from changes and is therefore non-stationary. This non-stationarity plays an important role in parallelisation.

The entire MCTS algorithm is outlined in Algorithm 2.1.

```
while time left do
    node ← SelectFromTree()
    leaf ← ExpandNode(node)
    result ← PlayoutFromLeaf(leaf)
    UpdateFromLeaf(leaf,result)
end
```

Algorithm 2.1: Monte-Carlo Tree Search

After a specific number of playouts have occurred, or a specific time has passed, the MCTS

search can be stopped and the best move selected according to some criteria, such as the node with the most playouts. When this occurs, the sub-tree with the resultant position as root can be retained as the current MCTS tree, while the rest is discarded. This allows subsequent searches to start with this tree as their initial tree and increases the overall number of playouts' results in the tree.

Given sufficient processing power, MCTS performs much better than other Computer Go techniques [9]. As such, it has become the paradigm for Computer Go [9].

It has been shown that, with infinite time, MCTS will converge to optimal play [2] and that given an increase in the number of playouts, MCTS increases in strength [10]. This increase in playouts can be achieved through an increase in thinking time or an increase in the rate of playouts. The latter can be accomplished through optimisation or parallelisation.

2.5.1 Pondering

In Computer Go, pondering is defined as the act of thinking during an opponent's thinking time. This increases the overall thinking time available for the program. MCTS makes this easy, as an additional search can be performed on the current MCTS tree during this time and then, when the opponent makes a move, the additional playouts will hopefully aid the following move search. Previous results have show pondering to add 60 Elo of strength to a Go engine [28].

The pondering search can be guided in some way, or perform the same as a normal search. One way of guiding the pondering is to select a number of the most likely opponent moves and focus on evaluating followups to only these moves. It has been shown that the unguided pondering search performs just as well or better than most attempts at guiding the search [28] therefore guided pondering was not considered in this project.

2.6 Parallelisation

Parallelisation of MCTS attempts to increase the strength of MCTS by increasing the rate of playouts, thereby increasing the total number of playouts done within a fixed thinking time. Parallelisation does this by making use of a number of processing nodes simultaneously. These nodes can be Central Processing Unit (CPU) cores on a Symmetric Multi-Processing (SMP) (multi-core) machine or spread out over a number of machines in a cluster. Other processing nodes, such as General Purpose Graphical Processing Units (GPGPUs), are also possible processing nodes, but have not yet been as effective as CPU cores [29] and were not considered in this project. Typically, a single execution thread is run per processing node, but this can change in situations, such as when some

nodes are required for other work or some nodes have a technology such as Intel's Hyper-Threading.

It is important to note that parallelising a well-tuned MCTS implementation will almost always create a loss of playing strength when the total number of playouts is fixed. This is due to the fact that node selection is non-stationary and therefore a parallelised implementation will select and perform playouts differently to the unparallelised version. The effect can be simplified to: a parallelised version will be the same strength or weaker than the unparallelised version, given that the parameters are tuned for the unparallelised version. This is defined as the *parallel effect* in this project. A real-life example of the parallel effect is: if you have one person to ask for game advice, it is easy to give that person a position to evaluate; however, if you have many people to ask for advice, it is much more difficult to *efficiently* ask them all for advice.

MCTS can be parallelised with a few different methods, the major ones being: tree parallelisation, leaf parallelisation, and root parallelisation [11] (see Figure 2.4).

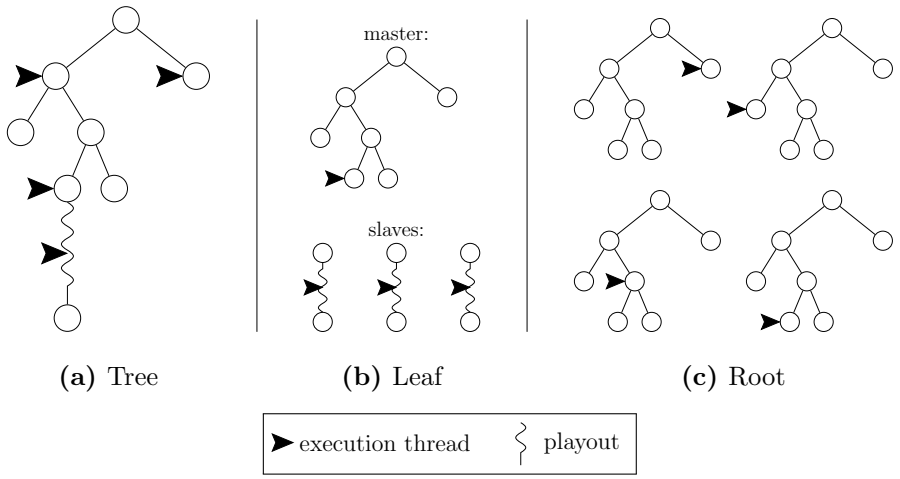


Figure 2.4: Parallelisation methods

2.6.1 Tree Parallelisation

In tree parallelisation, a shared MCTS tree is used by all execution threads (see Figure 2.4a). This implies that the tree is constantly shared amongst all the processing nodes. Tree parallelisation therefore lends itself to multi-core systems where memory is shared. This shared memory dependency presents an inherent problem for clusters, which are connected by comparatively high-latency connections.

The main problem with tree parallelisation is over-exploitation. Improvements which address this problem include virtual loss and lock-free modifications which have allowed tree parallelisation to scale up to 16 or more cores [11].

Virtual loss involves adding a loss to each node in the tree during the selection descent, and then removing it when propagating the result back up the tree [11]. This deters other

threads from descending down the same path in the tree if there is another path which is of similar quality, thus virtual loss encourage exploration.

To prevent race conditions with tree parallelisation, concurrency primitives such as mutexes are used. These mutexes control access to the data stored at each node and the mutexes can be local to each node or global. The use of mutexes means that data access to nodes is serialised. When a large number of threads is used, this can lead to time wasted spent waiting to acquire mutexes. The *lock-free* option dictates that tree node mutex locks are not acquired [30]. This means that multiple threads can access a node's data at the same time. However, this does not affect MCTS correctness, as the evaluation function is stochastic, so additional noise is not a problem. However, the lack of mutexes could lead to data loss, but the processing power freed from waiting for mutex locks can outweigh these losses [30].

2.6.2 Leaf Parallelisation

In *leaf parallelisation*, there is a single master node and multiple slave nodes (see Figure 2.4b). The master node maintains the MCTS tree and requests slave nodes to perform playouts starting from specific leaf positions. The master can broadcast the same position to all nodes or send different positions to different slaves. The master node then collects the results of these playouts and updates the tree. This method can be successful on clusters [31], but the single tree and master node can easily become a bottleneck [31]. It has been shown to scale to 13 nodes [31].

2.6.3 Root Parallelisation

In *root parallelisation*, each node maintains its own tree with periodic sharing of information (see Figure 2.4c). When information is shared, only a portion of the tree is shared in order to minimise the communication overhead of sharing. A possible sharing frequency is 3Hz and a possible portion of the tree to be shared is all nodes that are in the top 3 levels and have at least 5% of the total playouts through them [32]. If root parallelisation updates at an infinite frequency, shares the whole tree, and takes zero time to update, then it will be equivalent to tree parallelisation. Root parallelisation has been shown to scale to 40 nodes [32].

Chapter 3

Design

3.1 General

This project builds on the existing MCTS implementation of *Oakfoam* [33] (see Appendix D). The system design is not changed, but extensions are designed and later implemented. Section 4.1 elaborates on the usage of Oakfoam.

Figure 3.1 shows a system diagram of Oakfoam, with the additions of this project shaded and a connection present prior to this project shown as a dashed line. The non-shaded parts were present in Oakfoam prior to this project. The subsystems that this project adds to Oakfoam have been inserted in between existing subsystems as the figure shows, while abstracting their implementation details from other parts of the whole system.

The list of Oakfoam features prior to this project include:

- Functional MCTS implementation.
- GTP interface.
- Additional MCTS heuristics such as:
  - Rapid Action Value Estimation (RAVE);
  - Progressive Widening and Biasing; and
  - Last Good Reply with Forgetting (LGRF).
- Support for opening books.

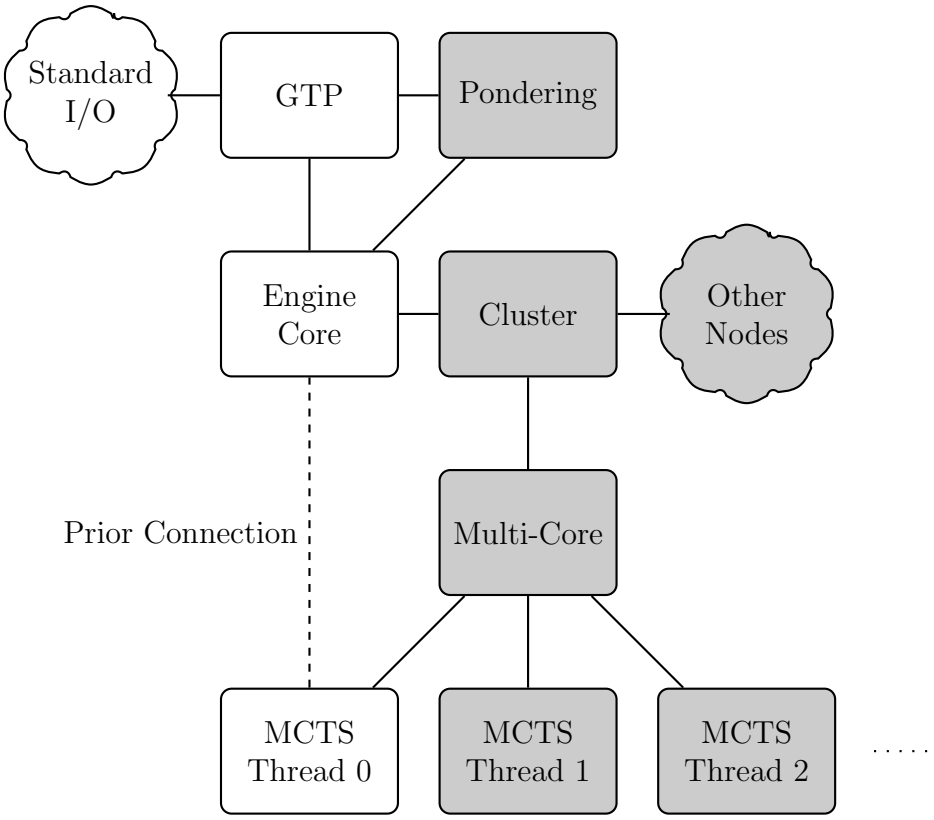


Figure 3.1: System Diagram for Oakfoam, with this project's additions shaded. A connection present prior to this project is shown as a dashed line.

3.2 Pondering

Pondering is the act of thinking during the opponent's thinking time. Some form of concurrency is required to be able to act on new GTP commands. Threading was chosen as the concurrency model. Other concurrency models, such as polling, were considered but rejected as GTP commands are read from standard input using a blocking call.

The method that will be used for pondering in this project is: a pondering thread will start pondering when no other work must be done, performing a normal MCTS search, and finally stop pondering when a new command is received, such as a move from the opponent. Pondering performs a MCTS search on the current tree. When the opponent makes a move, the valid sub-tree is kept and the standard thinking procedure is followed.

3.3 Multi-Core Parallelisation

Due to the shared memory nature of multi-core systems, it was decide to make use of tree parallelisation. A single tree, which is shared between all the cores, is therefore maintained in shared memory. A number of threads (typically one per core) are executed, and these will all work on the same tree. These threads will be stored in a thread pool and simultaneously started and stopped. This thread pool will help maintain control over

the threads. In order to address over-exploitation and concurrency primitive losses, the virtual loss and lock-free options will be implemented and investigated.

3.4 Cluster Parallelisation

In comparison with multi-core systems, clusters have high latency between processing nodes. Leaf and root parallelisation are therefore the only realistic candidates for cluster parallelisation. Due to project time constraints, both methods could not be implemented and tested. Root parallelisation was chosen in the belief that it would scale better [32, 34].

An implementation aspect that affects the design is the underlying communication system. The two main options that are available for clusters are Transmission Control Protocol (TCP) sockets and Message Passing Interface (MPI).

The MPI standard is the de-facto communication standard for High Performance Computing (HPC) [35]. It offers ease of use and support with tried-and-tested implementations [36]. MPI implementations have support for TCP/IP connections as an underlying system [36]. One advantage of MPI is that, given faster media than TCP/IP, such as shared memory or switched fabric communication, MPI can make use of them without additional code [36]. Another advantage is that MPI is available on most High Performance Computing (HPC) clusters [37], and greatly simplifies creating and running cluster jobs. A noticeable restriction of MPI is that all collective communications (any communications that involve more than two nodes) are synchronous. This implies that all the nodes will have to communicate at very specific times when sharing data.

TCP sockets is a bare-bones TCP/IP communication system. If TCP sockets are used, an entire communication layer must be written; however, very fine details can be optimised to make full use of the available network bandwidth. TCP sockets therefore offer great flexibility. TCP sockets are able to handle asynchronous and synchronous communication. Both TCP sockets and MPI guarantee delivery.

Due to the iterative nature of MCTS, synchronous communication was deemed a lesser issue, as all that would be required to minimise wasted resources is an accurate clock, which MPI itself provides. Use of this accurate clock would mean that all nodes could try and communicate as close to each other in time as possible, minimising time spent waiting.

It was decided to use MPI, as reliability and implementation difficulty were deemed more important than flexibility and the lack of asynchronous collective communications. This implies that synchronous communication will be used for sharing data amongst all the nodes and will affect the remainder of the cluster parallelisation design.

In order to synchronise communications, we will check after each playout to see if the next update should occur. Therefore the longest a node will wait to communicate is the

length of one playout. If the number of playouts completed between updates is more than 100 playouts, the total waiting time will be smaller than 1%, which is acceptable.

Once the specified interval has passed, each processing node will share a part of their tree with the rest of the nodes. Only a part of the tree will be shared in order to limit communication overhead. The portion of the tree to be shared will be based on the top few tree levels and how many playouts have passed through the node; this was chosen based on other cluster parallelisation implementations [32]. Figure 3.2 shows an example MCTS tree, with a portion that would typically be shared indicated. Only nodes in the top 3 tree levels with at least 20%[1] of the total playouts have been selected.

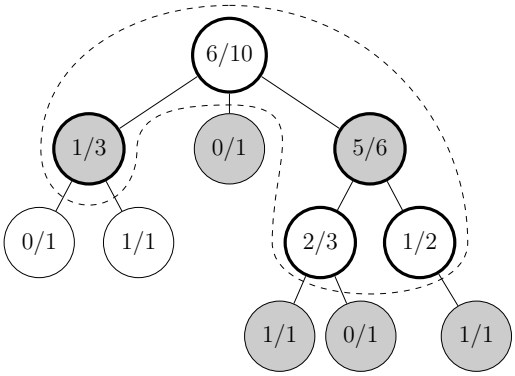


Figure 3.2: Example of cluster sharing. Nodes to be shared are indicated. These nodes are in the top 3 tree levels and have at least 20% of the total playouts.

The shared portions of trees will be combined after sharing. The number of playout wins and losses of each identical position will be combined by adding the wins and losses respectively. In this way, cluster parallelisation of the whole tree with an infinite frequency will be equivalent to multi-core parallelisation.

In order to share a node, a unique identifier for the node, and the number of playout wins and losses, must be shared. This will be enough information to combine the correct nodes from the different trees.

After sharing and updating the tree with the received information, the process will be repeated at a regular interval. In this way cluster parallelisation will synchronise a portion of the tree across the cluster at a specific frequency.

In a "production" cluster system, cluster parallelisation would be combined with multi-core parallelisation to make better use of available resources. However, for this project the two methods were not used together in order to isolate their respective strengths and weaknesses.

[1]The threshold of 20% is artificially high, and only used for example purposes.

Chapter 4

Implementation

4.1 General

This project will make use of an existing MCTS implementation, *Oakfoam* [33] (see Appendix D). Oakfoam was written by the author prior to the start of this project. It makes use of C++ as the main programming language, in order to take advantage of Object Orientated Programming (OOP), while at the same time being very fast. At the start of this project, Oakfoam already had over 10 000 lines of code in its codebase.

Oakfoam makes use of the de-facto interface standard for Go engines, Go Text Protocol (GTP) [38]. In this standard, the engine is the slave, and it can only respond to commands sent from the master. In the client-server representation, the server role is fulfilled by the engine. Commands and responses are sent over standard I/O and all messages are in plaintext. Oakfoam, like other Go engines, has a large number of parameters that can modify its behaviour. Oakfoam allows all of these parameters to be set at runtime through the use of GTP commands. This allows a single binary to be executed with different parameters and act differently, greatly simplifying normal usage and testing.

This project builds on Oakfoam. Therefore, some core software functionality, such as error handling and a build process, was already present and employed by this project's additions. Prior to this project, Oakfoam was single-threaded and did not possess pondering or parallelisation functionality.

4.2 Pondering

In order to add pondering to Oakfoam, threading was required. To maintain portability and reliability, Boost C++ Threads [39] were used. Additionally, Boost was already a requirement of Oakfoam, so no additional dependencies were required.

In the pondering implementation, a new thread is started after each GTP command has finished. This thread ponders until a new GTP command is received, whereupon it stops pondering and the GTP command is processed.

4.3 Multi-Core

Due to the method chosen (tree parallelisation), multi-core parallelisation also required threading. Boost C++ Threads were also used for this. When an MCTS search is performed, a number of threads are started and simultaneously begin an MCTS search. The number of threads is set using a parameter, so it can easily be set to the number of cores on the current machine. Once a stop condition is met, all the threads are stopped and the result is returned as per usual. The virtual loss and lock-free additions are implemented and can be enabled using parameters.

4.3.1 Challenges

A number of challenges were faced when implementing multi-core parallelisation. After the initial implementation was running, the speedup was initially very poor (less than 30% gain from another thread). Profiling using Intel VTune Amplifier [40] revealed that the memory allocation was using a global lock. Fixing this, and a few other small improvements, resulted in the most recent version and scales well on tested hardware (see Section 5.4.1).

During the course of the project, access was acquired to a 256-core shared memory machine at the Centre for High Performance Computing (CHPC) with the purpose of performing multi-core tests. Unfortunately, the multi-core parallelisation failed to scale on this machine. This issue was not resolved within the time available for this project and this machine could therefore not be used for testing.

4.4 Cluster

MPI was chosen as the communication system for cluster parallelisation in the design phase and Open MPI was chosen as the MPI implementation. Open MPI [36] was chosen as it is an open-source implementation with support and widespread usage [37]. MPI facilitates the launching of software processes on multiple nodes using the command `mpirun`. This command also initialises preliminary communications between the processes. Once the processes are started, all GTP commands are inputted by the first process before being broadcast to the other processes, in order to maintain the current state across all processes. When an MCTS search is required, all processes simultaneously perform an MCTS search, and periodically share information using the MPI collective communications as

per the design. Algorithm 4.1 shows pseudo code for cluster parallelisation.

```
last_update ← MPI_Time()
while time left do
    MCTS_Playout()
    if MPI_Time() > ( last_update + update_period ) then
        ClusterUpdate()
        last_update ← MPI_Time()
    end
end
```

Algorithm 4.1: Cluster parallelisation solution. This will run on each processing node. See Algorithm 4.2 for `ClusterUpdate()`.

```
local_nodes ← GetTreePortion()
local_messages ← MakeMessages(local_nodes)
cluster_messages ← MPI_Share (local_messages)
for each message in cluster_messages do
    UpdateTreeWithMessage(message)
end
```

Algorithm 4.2: Algorithm for `ClusterUpdate()`

The portion of the tree that is shared is divided up into messages, with a single message per tree node. In order to minimise communication overhead, these messages are small. In the design (see Section) it was determined that a hash and the number of playout wins and losses must be sent for each node. As hashes are not reversible, the parent hash must also be sent, in case the node is not yet present in another tree. Therefore, each message will consist of the following four parts:

1. Node hash
2. Parent hash
3. New playouts since last update
4. New wins since last update

64-bit Zobrist hashes [41] will be used as the hash.[1] Zobrist hashing is a technique that can generate an arbitrary length hash for a board position with a good distribution [41]. It is assumed that the 64-bit hashes will be unique and that any collisions will not affect performance.

[1] Zobrist hashing functionality was present in Oakfoam prior to this project.

In order to find the correct tree node given a hash, a hash table storing a pointer to the node in the tree is used. The hash table was chosen, as a fast lookup is possible given a good hash function. If the node hash is not in the table, the parent hash is used to find the parent node and then expand the parent to find the node in the tree.

Once the messages are shared amongst the cluster, they must be combined with the local MCTS tree. They are combined by adding the new playouts and wins to the node's respective values. In this way, each local tree will strive to be the sum of all the cluster trees had there been no parallelisation.

4.4.1 Challenges

A number of challenges were faced in implementing cluster parallelisation. The first main issue occurred when different nodes on the cluster had different MPI versions. The reported error message was obscure and not relevant to the problem, but after careful debugging the cause was found and fixed.

Another large issue caused seemingly random freezes of the cluster implementation. No feedback was given as to why it was occurring, and it was difficult to reproduce on other systems. Eventually it was determined that there was an error with the version of Open MPI in use (1.3.2). Fortunately, the problem only occurs when shared memory is used, therefore shared memory usage was explicitly disabled for this project.

Chapter 5

Testing and Results

5.1 Overview

Due to the stochastic nature of MCTS, it is very difficult to verify the correctness of an MCTS implementation. Instead, we elect to verify an MCTS implementation with empirical results; testing is therefore a fundamental part of MCTS research.

Usually, changes to the MCTS algorithm are evaluated by playing against a version of the same program without the change (self-play) and against another reference program. In this project it was decided to perform measurements using only self-play as it was expected that this would be able to offer comprehensive testing, while eliminating the extra complexity of another program. The Gomill tool package [42] was extensively used for testing.

Tests were performed on board sizes 9x9 and 19x19, as these are popular board sizes [5] and should give a good indication of how board size plays a role. However, a 19x19 board is close to 4.5 times as large as a 9x9 board, therefore playouts take more than four times longer on 19x19 and more moves are required to fill a portion of the 19x19 board. This means that tests on 19x19 will take much longer than those on 9x9.

The time that was required to perform tests was a limiting factor in this project. There are 100 or more moves in a typical 19x19 Go game, and with a time limit of 10s/move,[1] tests on 19x19 can take longer than 15min per game. A single game has a very high variance, therefore a number of games must be played to get an accurate result. A run of 100 games (the minimum number for a test in this report) can easily take longer than a day on 19x19. Testing was therefore selective and could not cover a very wide range of parameters.

There was no to very little variance on the speedup results, therefore no error bars are shown — only the fastest result of a few samples is shown.

[1] A time limit of 10s per move is rather fast, but realistic. Longer time limits would take even longer to test and were therefore not used.

Access to the CHPC's cluster was acquired, but system downtime and implementation difficulties (see Section 4.4.1) rendered this cluster practically useless to this project. All tests were performed on the Stellenbosch University's *Rhasatsha* cluster and another standalone machine very similar to a Rhasatsha cluster node.

5.1.1 Conventions

In all graphs that show error bars, these bars show the 95% confidence interval[2] of each result, unless otherwise noted. In some of the graphs, particularly those with error bars, the results are slightly staggered to aid readability — e.g.: all results in Figure 5.10 near the grid line for "4 Cores/Periods" are in fact measured for exactly 4 cores or periods.

The term "normalised" in the context of playout speed refers to the act of dividing results by the playout speed for a single core. In other words, a normalised playout speed of two implies a speed twice that of the single core measurement.

5.2 Pondering

Pondering is difficult to properly measure, as it highly depends on how long the opponent player thinks for. It was decided to measure pondering in games where the opponent thinks for an identical amount of time. We expect that the player with pondering enabled will be stronger.

If the thinking time is the same for both players, the number of playouts done during pondering will be approximately the same as during normal thinking time. Therefore, after the opponent move is made and the correct sub-tree is selected, a portion of the pondering playouts will remain. The strength gain from pondering should therefore not depend on the thinking time available if it is the same for both players.

5.2.1 Evaluation

Purpose

Determine if pondering provides a strength increase and quantify it.

Method

Two players played against each other, with pondering being enabled for the one, and both having a fixed time per move. This was repeated with different times, on 9x9 and 19x19 board sizes.

[2]The confidence interval used is of the normal approximation of a binomial distribution.

Results

The results are shown in Tables 5.1 and 5.2. The 95% confidence intervals are shown.

Time/Move	Games	Winrate [%]	Elo Diff.
2s	100	57 ± 9.70	50
10s	100	68 ± 9.14	130

Table 5.1: Performance with Pondering on 9x9

Time/Move	Games	Winrate [%]	Elo Diff.
2s	100	57 ± 9.70	50
10s	100	56 ± 9.73	40

Table 5.2: Performance with Pondering on 19x19

Analysis of Results

The results indicate that pondering functions within expectations and is in line with previous results in the field that show an improvement of 60 Elo [28]. Further tests can investigate the strength increase when time allocation is asymmetric.

5.3 Scaling

Although not a direct focus of this project, parallelisation depends on the principle that MCTS scales — if more playouts are performed, the playing strength increases. This section's aim is to verify this for our MCTS implementation and quantify the improvement of strength as the number of playouts increases. The scaling results will then represent the best possible results for parallelisation in following tests.

5.3.1 Evaluation

Purpose

Verify that MCTS improves in strength with an increase in thinking time and quantify this improvement.

Method

A fixed period of time was used for each series of tests. The reference player was given a single period of thinking time per move, while the opponent was given a number of periods per move. A number of games were then played between these players and the

results recorded. This process was repeated for range of periods given to the one player, on board sizes 9x9 and 19x19.

Results

The results are shown in Figures 5.1 and 5.2.

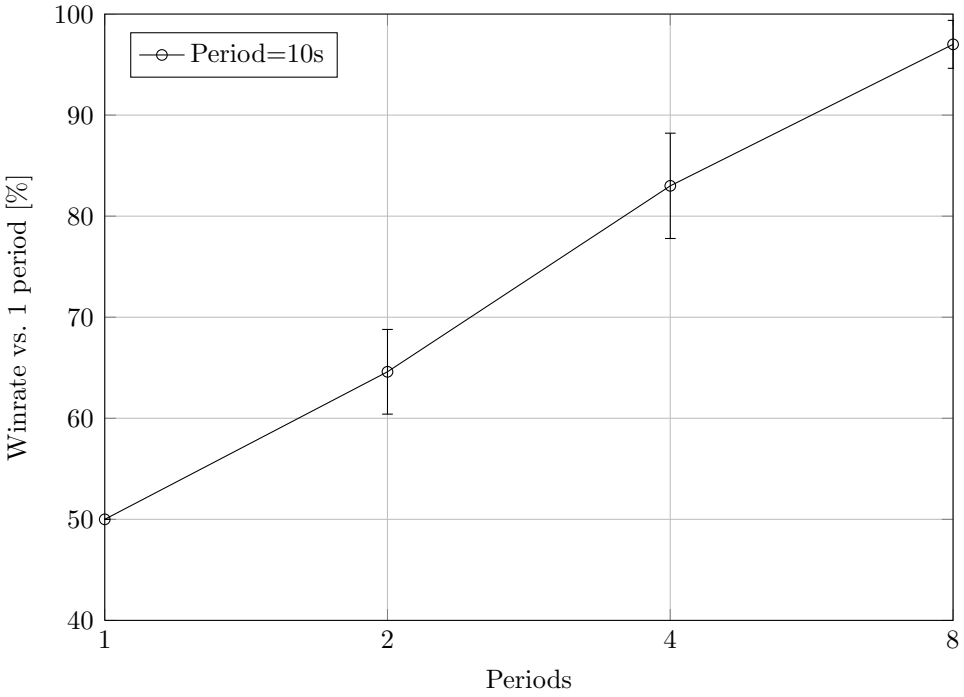


Figure 5.1: Scaling of MCTS on 9x9

Analysis of Results

The results confirm that an increase in thinking time results in an increase in strength for MCTS on both 9x9 and 19x19. These results are the ideal results for parallelisation and will be used for comparison purposes in the tests following.

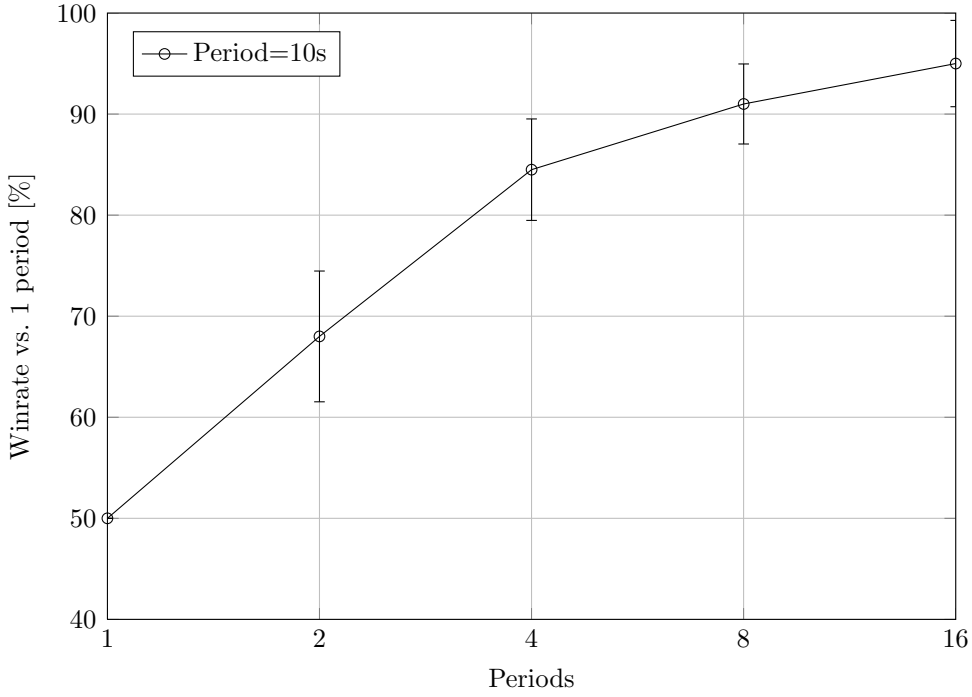


Figure 5.2: Scaling of MCTS on 19x19

5.4 Multi-Core

Multi-core parallelisation will be evaluated in two ways. Firstly, we will determine if the rate of playouts increases with an increase in parallel hardware. This will determine the upper limit on the strength increase. Secondly, we will test for the parallel effect to determine if the rate increase leads to an increase in strength.

Multi-core testing was performed on the Rhasatsha cluster. The cluster's processing nodes are mostly machines with eight cores, and two machines with more cores. However, the job allocation uses a greedy algorithm and the machines with more cores could therefore not be acquired for use. Multi-core tests could therefore only go up to eight cores.

In all of this section's tests, no cores were over-subscribed — the number of active threads on each machine was less than or equal to the number of cores on that machine.

5.4.1 Speedup

Purpose

Successful parallelisation increases the rate of playouts with an increase in processing nodes. Verify and quantify this increase in speed for multi-core parallelisation.

Method

The program was started on a specific number of cores. The program was then sent a command to generate a move and the rate of playouts was then recorded. This was repeated with the virtual loss and lock-free options. After all tests, the results were normalised around the single core measurements.

Results

The results are shown in Figures 5.3 and 5.4.

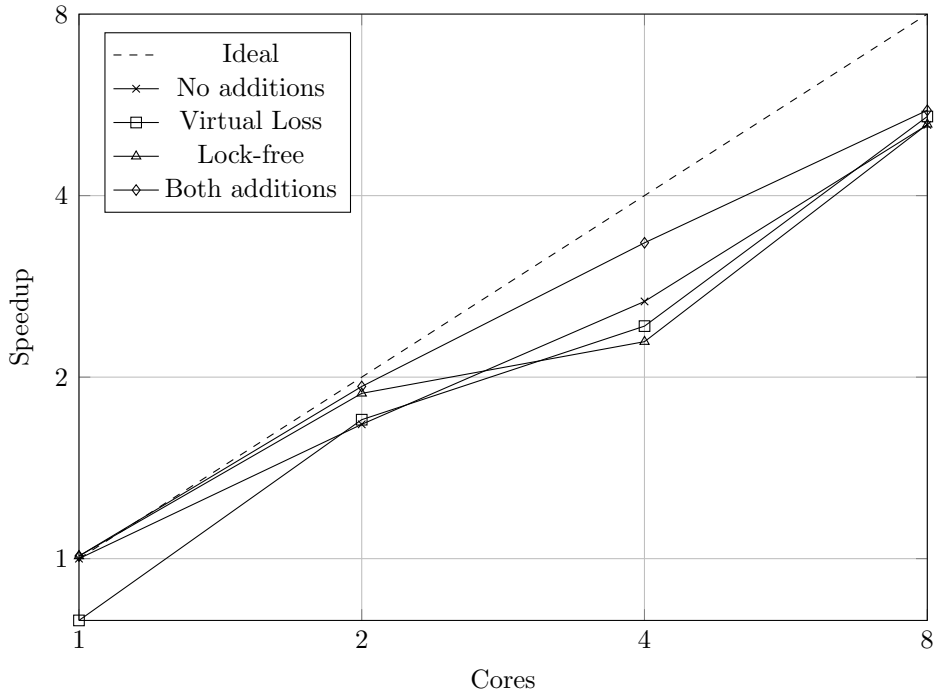


Figure 5.3: Speedup of Multi-Core Parallelisation on 9x9

Analysis of Results

On both 9x9 and 19x19, multi-core parallelisation showed an increase in speed proportional to the increase in processing nodes. Multi-core parallelisation is therefore successful in increasing the rate of playouts.

However, the results show no discernible difference between the various multi-core versions in the range of hardware tested on and can therefore no remark can be made regarding the virtual loss and lock-free additions.

Further tests should investigate the speedup achieved on hardware with more cores. It is proposed that with more cores, the lock-free option will become more important.

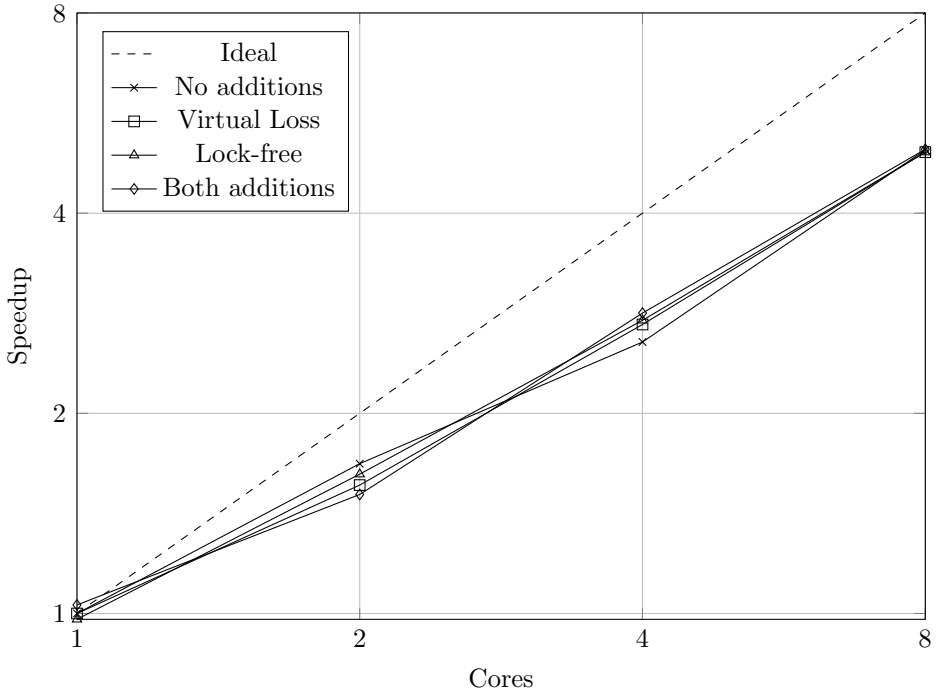


Figure 5.4: Speedup of Multi-Core Parallelisation on 19x19

5.4.2 Parallel Effect

Purpose

The parallel effect is the strength lost when MCTS is parallelised. Measure this effect for multi-core parallelisation.

Method

The reference player was given a single core, while the opponent was given a number of cores. A number of games were then played between these players with a fixed number of playouts per move. This process was repeated for range of cores given to the one player, on board sizes 9x9 and 19x19, and with virtual loss and lock-free additions.

Results

The results are shown in Figures 5.5 and 5.6. Note that both figures only show a limited portion of the y-axis to amplify differences.

Analysis of Results

The results show that, on the tested hardware, multi-core parallelisation shows no parallel effect on 9x9. On 19x19, the multi-core parallelisation shows no parallel effect with the virtual loss addition, while without the addition, strength is severely affected by the parallel effect.

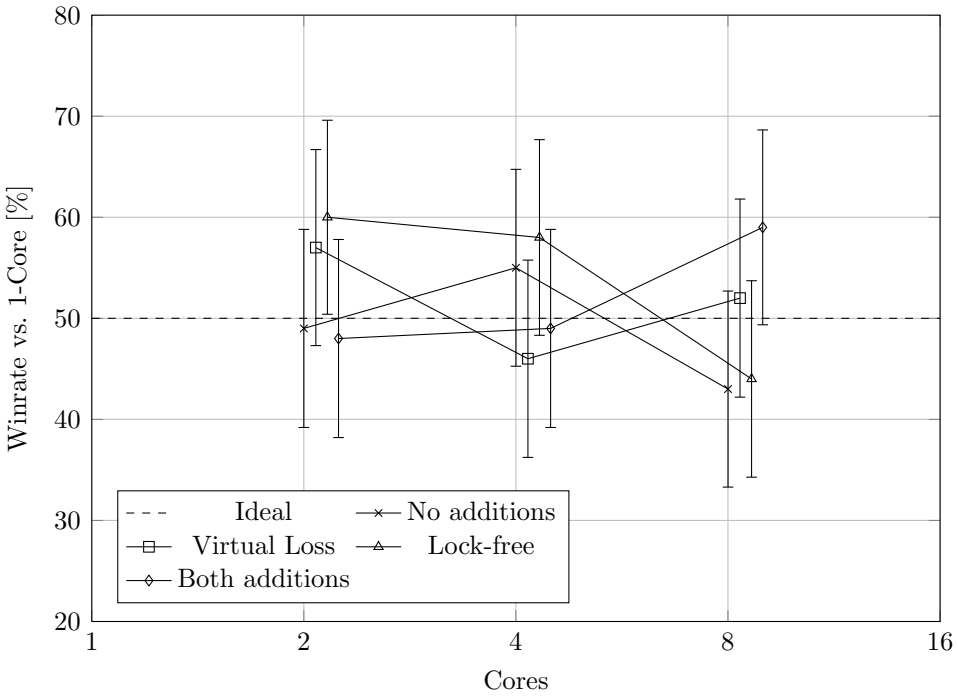


Figure 5.5: Parallel Effect of Multi-Core Parallelisation on 9x9

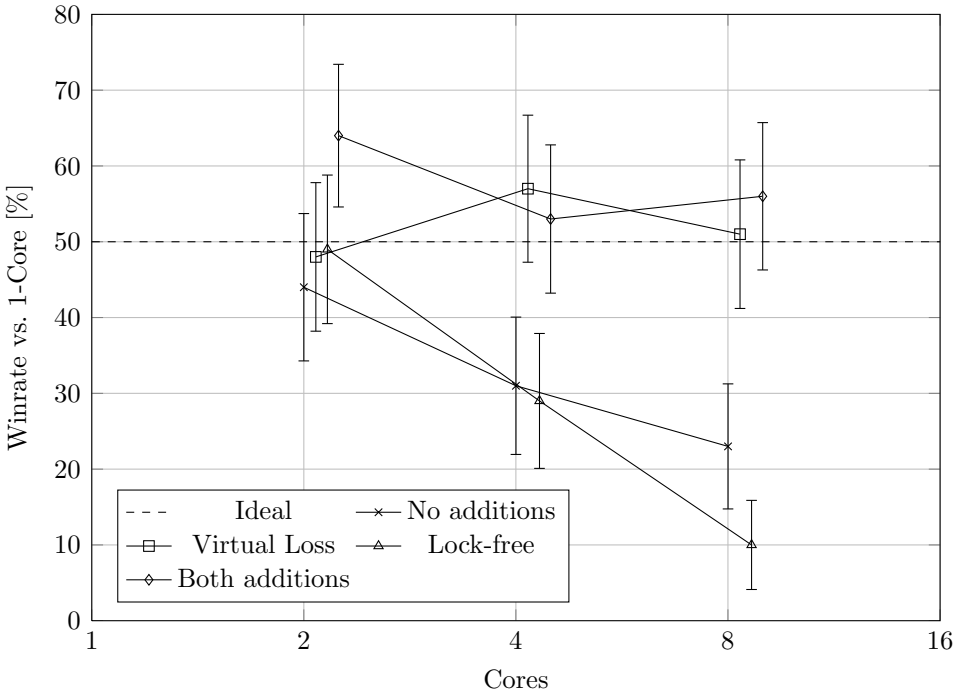


Figure 5.6: Parallel Effect of Multi-Core Parallelisation on 19x19

The lack of the parallel effect (with virtual loss) and linear speedup implies that the multi-core parallelisation scales very well on the available hardware. As such, it was decided not to perform a further strength comparison, as this would require extensive time for additional unnecessary testing.

5.5 Cluster

Cluster parallelisation will be evaluated in three ways. Firstly, we will determine if the rate of playouts increases with an increase in parallel hardware. This will determine the upper limit on the strength increase. Secondly, we will test for the parallel effect, to determine if the rate increase leads to an increase in strength. Finally, we will quantify the strength increase gained from an increase in parallel hardware.

All cluster tests were performed on the Rhasatsha cluster, located at Stellenbosch University. All tests made use of the job scheduling software on the cluster.

The interval between tree-sharing updates is a parameter for some of the tests and is referred to as p.

5.5.1 Speedup

Purpose

Successful parallelisation increases the rate of playouts with an increase in processing nodes. Verify and quantify this increase in speed for cluster parallelisation.

Method

The program was started on a various numbers of cluster cores. The program was then sent a command to generate a move and the rate of playouts was then recorded. After all tests, the results were normalised around the single core measurements.

Results

The results are shown in Figures 5.7 and 5.8.

Analysis of Results

The results show very good speedup for cluster parallelisation on 9x9. On 19x19, the speedup is close to linear, but with a low gradient. Additional testing is required to determine the exact cause of this low gradient, however, it is postulated that this is due to the length of a playout on 19x19 and the fact that sharing is synchronised. Further testing can investigate this.

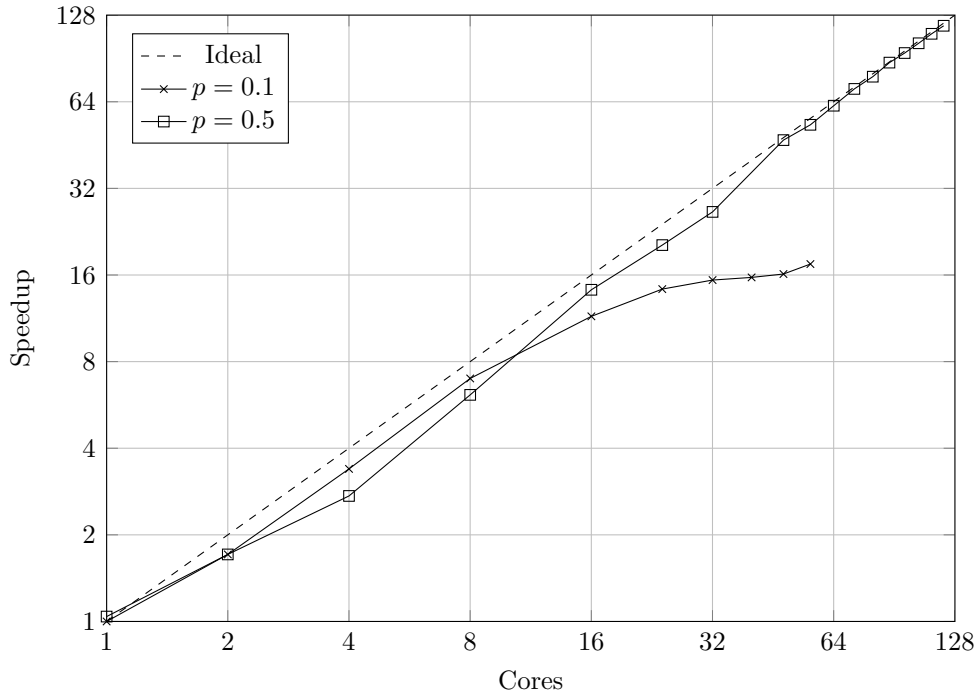


Figure 5.7: Speedup of Cluster Parallelisation on 9x9

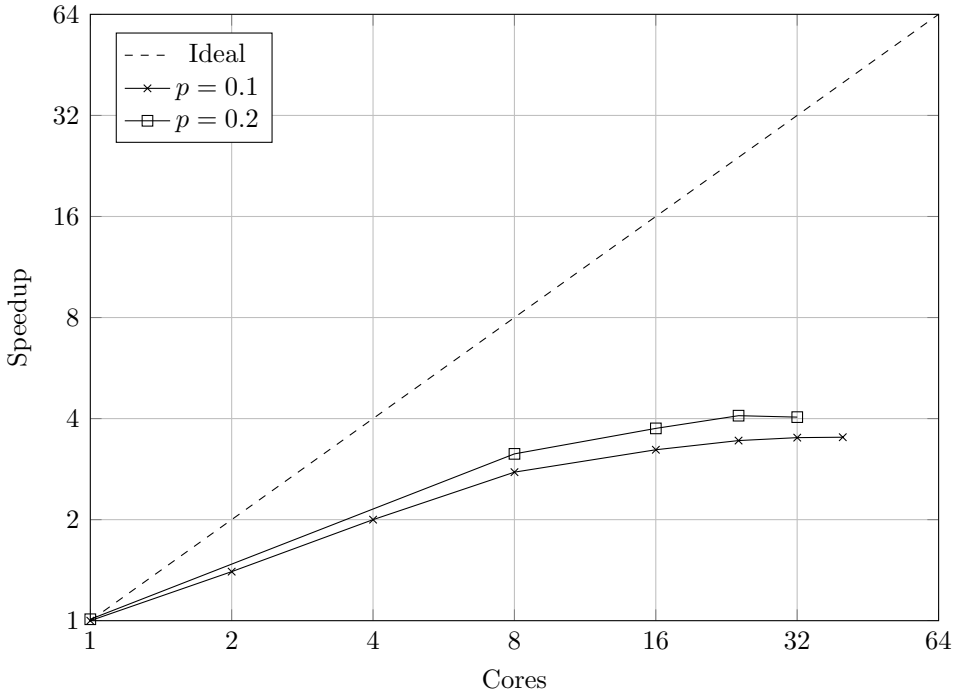


Figure 5.8: Speedup of Cluster Parallelisation on 19x19

5.5.2 Parallel Effect

Purpose

The parallel effect is the strength lost when MCTS is parallelised. Measure this effect for cluster parallelisation.

Method

The reference player was given a single core, while the opponent was given a number of cores. A number of games were then played between these players with a fixed number of playouts per move. This process was repeated for range of cores given to the one player, on board sizes 9x9 and 19x19, and for different sharing intervals.

Results

The results are shown in Figure 5.9. Note that the figure only shows a limited portion of the y-axis to amplify differences.

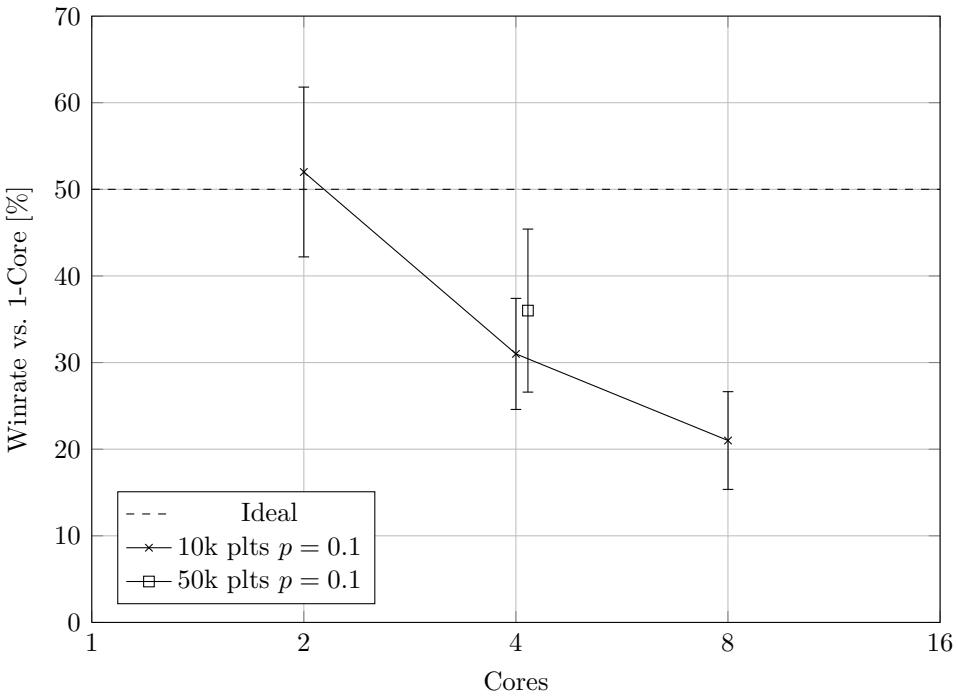


Figure 5.9: Parallel Effect of Cluster Parallelisation on 9x9

Analysis of Results

The results show that the parallel effect is prominent for cluster parallelisation on 9x9. It was therefore decided to not perform further parallel effect tests and rather perform a strength comparison on 9x9 and 19x19, shown in the next section.

5.5.3 Strength Comparison

Purpose

Measure the strength increase when multiple cluster cores are used for a fixed time search.

Method

The reference player was given a single core, while the opponent was given a number of cores. A number of games were then played between these players with a fixed thinking time per move. This process was repeated for range of cores given to the one player, on board sizes 9x9 and 19x19, and for different sharing intervals. These results are compared to the scaling results from Section 5.3.

Results

The results are shown in Figures 5.10 and 5.11.

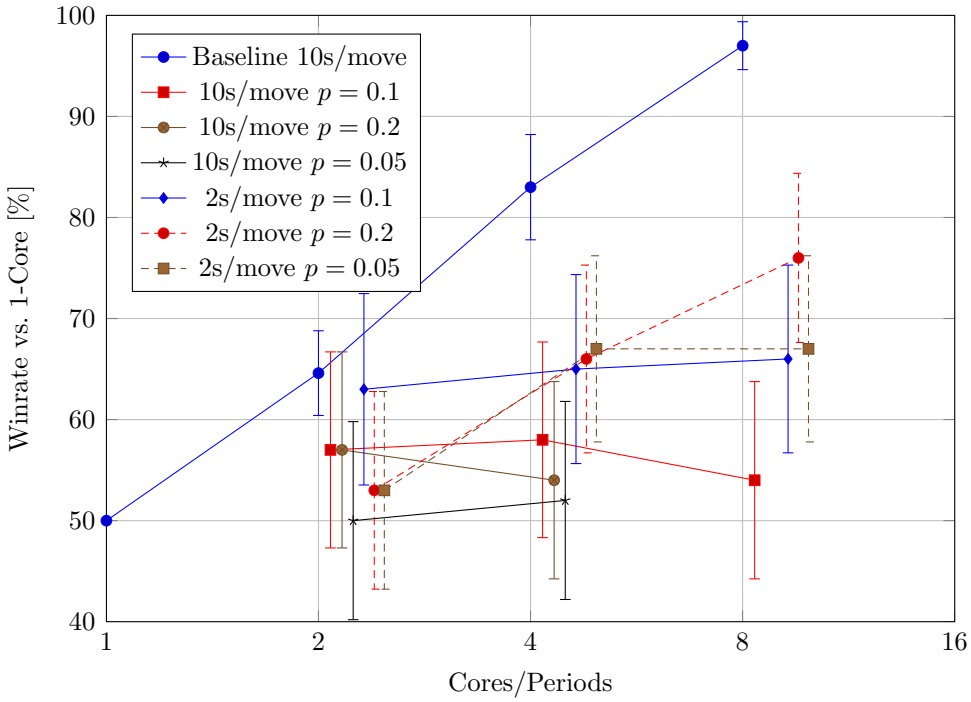


Figure 5.10: Strength of Cluster Parallelisation on 9x9

Analysis of Results

The results on 9x9 show that the cluster parallelisation is not very successful and is not able to scale well on the cluster. On 19x19, the results show that cluster parallelisation is able to scale up to eight cores, where it achieves a strength equivalent to four cores for ideal parallelisation. Also, on 19x19 running on too many cores can be detrimental to strength, as shown by the results for 32 cores being worse than those on 16 cores.

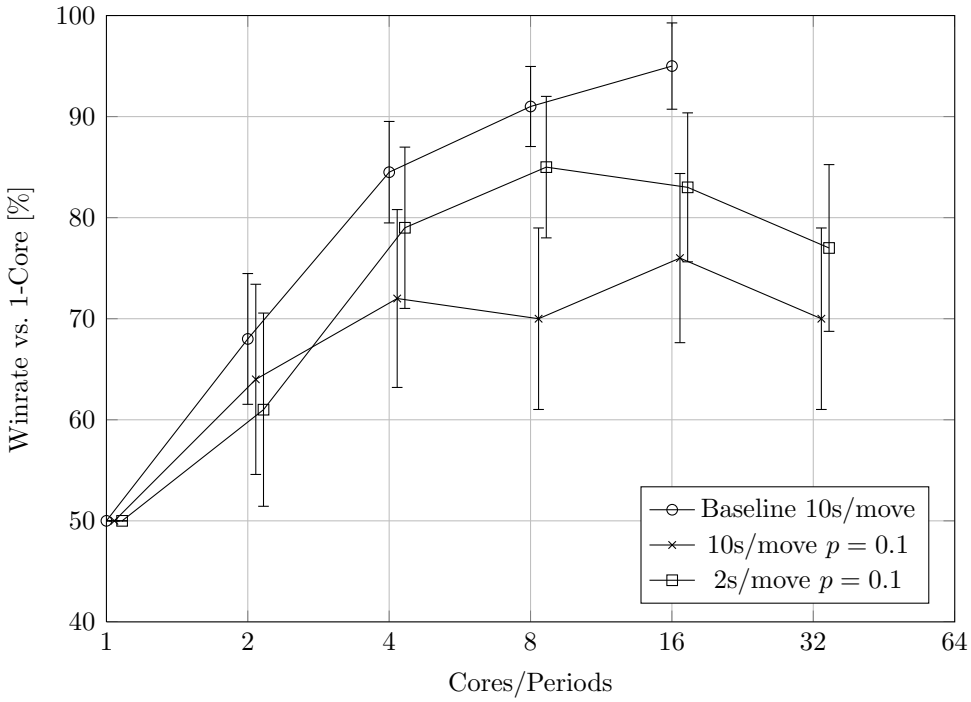


Figure 5.11: Strength of Cluster Parallelisation on 19x19

Chapter 6

Conclusion

6.1 Overview

MCTS is the focus of current Computer Go research. Currently, processors are making increasing use of parallel hardware, making parallelisation more important. This project researched and implemented solutions for pondering and parallelising of MCTS in order to take advantage of today and tomorrow's parallel hardware.

An existing MCTS implementation was successfully improved through the design and implementation of pondering and parallelisation on multi-core and cluster systems. The design was correctly implemented; however, the design failed to address some unforeseen issues that came up during testing. If time allowed, an iterative process, where the design and implementation are updated based on testing, could eventually address all these issues.

The pondering implementation performed according to expectations, with strength gains in the order of 60 Elo.

Multi-core parallelisation was tested up to eight cores and, with the virtual loss addition, showed no measurable negative parallel effect on both 9x9 and 19x19. This, coupled with a high linear speedup, show that the multi-core parallelisation achieves close to ideal scaling on the tested hardware.

Cluster parallelisation was tested up to eight and 32 cores on 9x9 and 19x19 respectively. It showed minimal to no strength improvement on 9x9. However, on 19x19 it showed a scaling up to eight cores where it was equivalent in strength to four ideal cores. This testing shows that there is much room for improvement on both 9x9 and 19x19 for cluster parallelisation.

This project has demonstrated that MCTS can successfully take advantage of multi-core and cluster systems with varying degrees of success. Further work can hopefully address the shortcomings that were identified.

6.2 Contributions

This project makes the following contributions:

- Oakfoam is the only open source MCTS Computer Go engine licensed under the copyleft BSD license, and this project made a substantial contribution to Oakfoam. A contribution to the Computer Go community was therefore made.
- Oakfoam has been improved to become one of the few MCTS implementations with cluster parallelisation.
- Oakfoam is one of only two MCTS implementations to currently have open source cluster parallelisation.[1]
- Oakfoam is now the only open source MCTS implementation to make use of MPI.

6.3 Further Work

Further work can test the multi-core parallelisation on more than eight cores to determine if the multi-core implementation is sufficient for many-core hardware not available for this project.

Further work on cluster parallelisation can also consider leaf parallelisation, which was not implemented in this report, as well as hybrids of two or more parallelisation methods.

Further work can also consider a wider range of cluster parallelisation parameter values with the hope of finding a more optimal set of parameters. Additionally, further work can investigate the possibility to parallelising MCTS for ad-hoc very-high-latency networks.

6.4 Closing

MCTS is the dominant algorithm for Computer Go. Parallelisation provides a way for MCTS to take advantage of the large processing power of parallel hardware and dramatically increase the strength of Computer Go programs. This project has implemented pondering and parallelisation techniques to enable an MCTS implementation to take advantage of parallel hardware. There is still room for improvement, and further work can advance on this project's work.

[1] The Pachi engine also has cluster parallelisation.

Bibliography

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from Berkeley," tech. rep., University of California at Berkeley, 2006.

[2] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *ECML-06*, 2006.

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

[4] "Rules of Go." Sensei's Library, `http://senseis.xmp.net/?RulesOfGo`.

[5] "Different sized boards." Sensei's Library, `http://senseis.xmp.net/?DifferentSizedBoards`.

[6] R. A. Hearn, *Games, Puzzles, and Computation*. PhD thesis, Massachusetts Institute of Technology, 2006.

[7] "Reading." Sensei's Library, `http://senseis.xmp.net/?Reading`.

[8] C. Garlock, "Michael Redmond on studying, improving your game and how the pros train." `http://www.usgo.org/news/2010/06/michael-redmond-on-studying-improving-your-game-and-how-the-pros-train/`, 2010.

[9] A. Rimmel, O. Teytaud, C.-S. Lee, S.-J. Yen, M.-H. Wang, and S.-R. Tsai, "Current frontiers in Computer Go," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, p. 229, 2010.

[10] R. B. Segal, "On the scalability of parallel UCT," *Lecture Notes in Computer Science*, vol. 6515/2011, pp. 36–47, 2011.

[11] G. M. Chaslot, M. H. Winands, and J. H. van den Herik, "Parallel Monte-Carlo Tree Search," in *Proceedings of the 6th International Conference on Computer and Games*, Springer, 2008.

[12] B. Bouzy and T. Cazenave, "Computer Go: an AI oriented survey," 2001.

[13] "Elo rating." Sensei's Library, `http://senseis.xmp.net/?EloRating`.

[14] A. E. Elo, *The Rating Of Chess Players, Past & Present*. Arco, 1978. ISBN 0-668-04721-6.

[15] "Rank." Sensei's Library, `http://senseis.xmp.net/?Rank`.

[16] "Handicap." Sensei's Library, `http://senseis.xmp.net/?Handicap`.

[17] M. Shor, "Game tree." Dictionary of Game Theory Terms, `http://www.gametheory.net/dictionary/GameTree.html`.

[18] C. University, "Minimax search and alpha-beta pruning." `http://www.cs.cornell.edu/courses/cs312/2002sp/lectures/rec21.htm`.

[19] C. Frayn, "Computer chess programming theory." `http://www.frayn.net/beowulf/theory.html#negamax`.

[20] J. Woller, "The basics of Monte Carlo simulations." `http://www.chem.unl.edu/zeng/joy/mclab/mcintro.html`, 1996.

[21] B. Brügmann, "Monte Carlo Go," 1993.

[22] P. Kaminski, "Las Vegas Go," 2002.

[23] R. Carr, "Simulated annealing." Wolfram Web Resource, `http://mathworld.wolfram.com/SimulatedAnnealing.html`.

[24] B. Bouzy and B. Helmstetter, "Monte Carlo Go developments," in *Advances in Computer Games conference (ACG-10), Graz 2003*, pp. 159–174, Kluwer, 2003.

[25] J. Mehat and T. Cazenave, "Combining UCT and nested monte carlo search for single-player general game playing," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, p. 271, 2010.

[26] G. Chaslot, *Monte-Carlo Tree Search*. PhD thesis, Maastricht University, 2010.

[27] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo Tree Search," in *Proceedings of the 5th International Conference on Computer and Games* (J. H. van den Herik, P. Ciancarini, and H. H. L. M. Donkers, eds.), vol. 4630/2007 of *Lecture Notes in Computer Science*, (Turin, Italy), pp. 72–83, Springer, June 2006.

[28] S.-C. Huang, R. Coulom, and S.-S. Lin, "Time management for monte-carlo tree search applied to the game of go," *International Conference on Applications of Artificial Intelligence (TAAI)*, 2011.

[29] C. Nentwich, "CUDA and GPU performance." `http://www.mail-archive.com/computer-go@computer-go.org/msg12474.html`, 2009.

[30] M. Enzenberger and M. Müller, "A lock-free multithreaded Monte-Carlo Tree Search algorithm," in *Advances in Computer Games 12*, 2009.

[31] H. Kato and I. Takeuchi, "Parallel Monte-Carlo Tree Search with simulation servers," in *13th Game Programming Workshop (GPW-08)*, November 2008.

[32] A. Bourki, G. Chaslot, M. Coulm, V. Danjean, H. Doghmen, T. Hérault, J.-B. Hoock, A. Rimmel, F. Teytaud, O. Teytaud, P. Vayssière, and Z. Yu, "Scalability and parallelization of Monte-Carlo Tree Search," in *The International Conference on Computers and Games 2010*, (Kanazawa Japon), 2010.

[33] F. van Niekerk, "Oakfoam." `http://bitbucket.org/francoisvn/oakfoam`.

[34] R. S. Kamil Rocki, "Massively parallel Monte Carlo Tree Search," 2010.

[35] J. Kepner, "Parallel programming with MatlabMPI." `http://www.ll.mit.edu/mission/isr/matlabmpi/matlabmpi.html`.

[36] "Open MPI: Open source high performance computing." `http://www.open-mpi.org/`.

[37] "Open MPI: FAQ." `http://www.open-mpi.org/faq/?category=general#why`.

[38] G. Farnebäck, "Specification of the Go Text Protocol, version 2, draft 2," 2002. GTP.

[39] "Boost C++ libraries." `http://www.boost.org/`.

[40] "Intel VTune Amplifier XE." `http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/`.

[41] A. L. Zobrist, "A new hashing method with application for game playing," tech. rep., 1970.

[42] M. Woodcraft, "Gomill tool suite." `http://mjw.woodcraft.me.uk/gomill/`.

[43] "GCC, the GNU compiler collection." `http://gcc.gnu.org/`.

[44] "Autoconf." `http://www.gnu.org/s/autoconf/`.

[45] "Mercurial." `http://mercurial.selenic.com/`.

[46] "Gogui." `http://gogui.sourceforge.net/`.

[47] "Valgrind." `http://valgrind.org/`.

Appendix A

Project Planning Schedule

Detailed project planning was performed in late July. At this point, a literature review had already taken place, and pondering and multi-core parallelisation had been implemented. The Gantt chart generated for the planning is shown in Figure A.1.

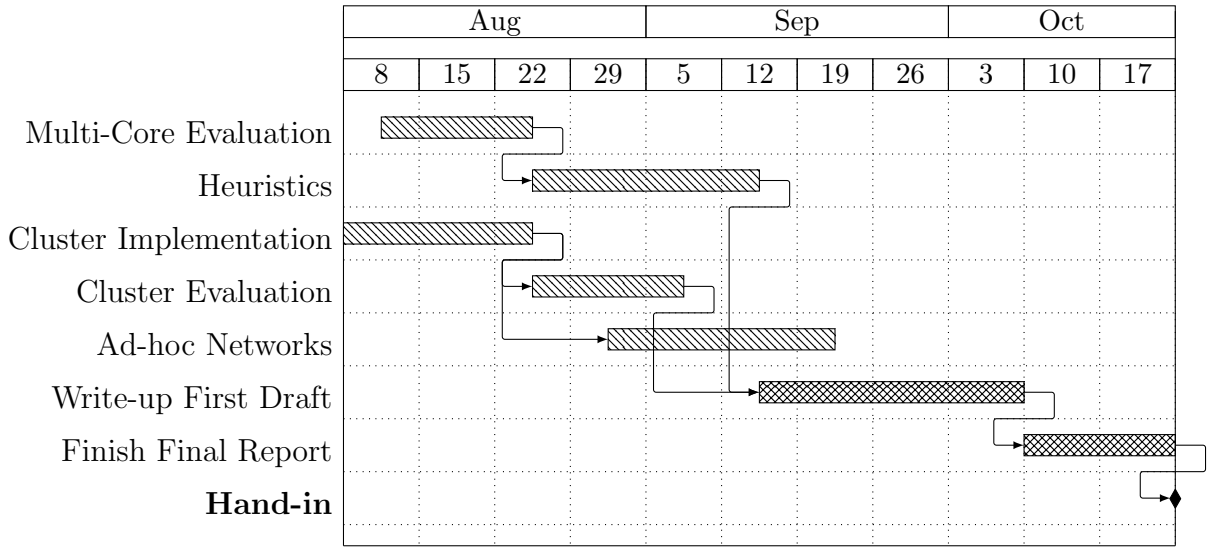


Figure A.1: Project Plan

Appendix B

Project Specifications

The following specifications were defined for this project:

- Research the current state of Computer Go, MCTS and its parallelisation.
- Develop designs for:
  - Pondering
  - Multi-core parallelisation
  - Cluster parallelisation
- Implement above designs by extending an existing MCTS implementation.
- Evaluate the performance of above implementations.

Appendix C

Outcomes Compliance

This project fulfils a number of ECSA outcomes as listed below, with associated sections of the report identified:

ECSA Outcome	Compliance
ECSA Outcome #1 Identification of problems, suggestion and implementation of solutions	The problem of MCTS parallelisation was studied and identified in Chapter 2, solutions were identified in Chapter 3 and the implementation was presented in Chapter 4.
ECSA Outcome #2 Application of a knowledge of mathematics and basic engineering science in implementing solutions	Pondering, multi-core parallelisation and cluster parallelisation solutions were designed in Chapter 3 and relied heavily on Computer Science knowledge. Statistical testing of these solutions was performed in Chapter 5.
ECSA Outcome #3 Implementation of solutions through the design of components, subsystems and systems	The proposed solutions were designed to be self-contained and form a part of a larger pre-existing system in Chapter 3.
ECSA Outcome #4 Gathering information, analysing it critically, and then drawing sensible conclusions	A background study was performed and the findings were presented in Chapter 2, and used to make sound design and implementation decisions in Chapters 3 and 4. In Chapter 5, comprehensive testing was performed and analysed. Finally, conclusions on the project were presented in Chapter 6.
ECSA Outcome #5 Effective use of aids such as measurement equipment and software in order to verify and analyse designs	A number of 3rd party tools were used in this project and listed in Appendix F. Chapter 5 demonstrated the use of some of these tools to verify and analyse the implemented solutions.
ECSA Outcome #6 Effective reporting of the project, in both written and oral form	This report presented the project's work in written form and an oral presentation will present said work in an oral form.
ECSA Outcome #9 Demonstration of the ability of independent learning	An extensive background study of the field of Computer Go and MCTS was performed and presented in Chapter 2.

Appendix D

Source Code

The source code for Oakfoam is maintained using the Mercurial version control system and released under the open source BSD license. Complete source code is available from the URL shown below:

`http://bitbucket.org/francoisvn/oakfoam`

D.1 Build Instructions for Ubuntu

Instructions for building Oakfoam on Ubuntu are included here for convenience. They should be easy to adapt to other *nix systems.

Note: '`$`' is not part of the below commands, but rather signifies a user prompt.

1. Install the necessary packages:

```
$ sudo apt-get install g++ libboost-all-dev
```

2. Get a copy of the Oakfoam source code in one of two ways:

```
$ wget http://bitbucket.org/francoisvn/oakfoam/get/tip.tar.gz
$ tar -xzf tip.tar.gz
$ cd francoisvn-oakfoam-tip
```

   or (with Mercurial):

```
$ hg clone http://bitbucket.org/francoisvn/oakfoam
$ cd oakfoam
```

3. Build Oakfoam:

```
$ ./configure
$ make
```

4. The Oakfoam binary should now be created and named `oakfoam`.

In order to compile in the MPI functionality, also install the `openmpi-dev` and `openmpi-bin` packages and run "`./configure --with-mpi`" prior to compiling.

Appendix E

Go Rules

There are a number of different rule sets for Go. The *Tromp-Taylor* Go Rules are widely accepted amongst the Computer Go community due to their clear and mathematical formulation. They are reproduced here verbatim from `http://homepages.cwi.nl/~tromp/go.html`. The rules are in **bold**, while comments follow.

1. **Go is played on a 19x19 square grid of points, by two players called Black and White.** The grid of points is usually marked by a set of 19x19 lines on a wooden board. Each player has an arbitrarily large set of stones of his own color. By prior agreement a rectangle of different dimensions may be used.

2. **Each point on the grid may be colored black, white or empty.** Using boards, coloring a point (intersection) black or white means placing a stone of that color on the point. Coloring a point empty, i.e. emptying a point, means removing the stone from it.

3. **A point P, not colored C, is said to reach C, if there is a path of (vertically or horizontally) adjacent points of P's color from P to a point of color C.** Connected stones of the same color, sometimes called *strings*, all reach the same colors. Reaching empty means having empty points adjacent to the string, called *liberties*.

4. **Clearing a color is the process of emptying all points of that color that don't reach empty.** Strings without liberties cannot exist on the board at the end of a turn.

5. **Starting with an empty grid, the players alternate turns, starting with Black.** For handicap games, the weaker player, taking black, may be given an *n stone handicap*; these are n consecutive moves played before the first white move.

6. **A turn is either a pass; or a move that doesn't repeat an earlier grid coloring.** This is the *positional superko* (PSK) rule, while the *situational superko* (SSK)

rule forbids repeating the same grid coloring with the same player to move. Only in exceedingly rare cases does the difference matter, sufficient reason for the simpler PSK rule to be prefered.

7. **A move consists of coloring an empty point one's own color; then clearing the opponent color, and then clearing one's own color.** For any specific move, at most one of the clearing processes can have effect; the first is called capture, the second suicide. Allowing suicide means that a play on an empty point can be illegal only due to superko.

8. **The game ends after two consecutive passes.** As a practical shortcut, the following amendment allows *dead stone removal*: After only 2 consecutive passes, the players may end the game by agreeing on which points to empty. After 4 consecutive passes, the game ends as is.

9. **A player's score is the number of points of her color, plus the number of empty points that reach only her color.** This is called *area scoring*. An almost equivalent result is reached by *territory scoring* where in addition to empty surrounded space we count opponent stones captured instead of own stones not captured.

10. **The player with the higher score at the end of the game is the winner. Equal scores result in a tie.** By prior agreement, for games between equals, a fixed amount can be added to white's final score. This is called *komi*, and can be chosen a non-integer such as 5.5 to avoid ties.

Appendix F

Tools Used

This project made use of a number of 3rd party tools:

GCC
: GNU Compiler Collection, including the C++ compiler used [43].

GNU autotools
: `configure` script, including Makefile generation [44].

Mercurial
: Distributed Version Control System (DVCS) for source code [45].

GoGui
: GTP GUI frontend for playing against and debugging Go engines [46].

Gomill
: Tool suite for generating Go matchups and recording game results [42].

Boost
: C++ library for threading and other functionality [39].

Open MPI
: MPI implementation for cluster parallelisation [36].

Valgrind
: Memory checker and single threading profiler [47].

Intel VTune Amplifier
: Advanced profiler for multi-threaded programs [40].

Appendix G

Elo Ratings

Elo Ratings are a system used to quantify the strength of a player in a game such as Go. Given the Elo scores of players A and B as R_A and R_B respectively, the probability of A beating B is given as [14]:

$$E_A = \frac{1}{1 + 10^{(R_B - R_A)/400}} \tag{G.1}$$

Table G.1 shows the winning probability of the stronger player given a difference in Elo score.

Elo Diff.	Prob. [%]	Elo Diff.	Prob. [%]	Elo Diff.	Prob. [%]
10	51.44	210	77.01	410	91.37
20	52.88	220	78.01	420	91.82
30	54.31	230	78.98	430	92.24
40	55.73	240	79.92	440	92.64
50	57.15	250	80.83	450	93.02
60	58.55	260	81.71	460	93.39
70	59.94	270	82.55	470	93.74
80	61.31	280	83.37	480	94.06
90	62.67	290	84.15	490	94.38
100	64.01	300	84.90	500	94.68
110	65.32	310	85.63	510	94.96
120	66.61	320	86.32	520	95.23
130	67.88	330	86.98	530	95.48
140	69.12	340	87.62	540	95.72
150	70.34	350	88.23	550	95.95
160	71.53	360	88.82	560	96.17
170	72.68	370	89.38	570	96.38
180	73.81	380	89.91	580	96.57
190	74.91	390	90.42	590	96.76
200	75.97	400	90.91	600	96.93

Table G.1: Probabilities for some Elo differences

Table G.1 Source: `http://cgos.boardspace.net/`