Monte-Carlo Tree Search Parallelisation for Computer Go

Francois van Niekerk
E&E Engineering Department
Stellenbosch University
7602 Matieland
South Africa
francoisvn@ml.sun.ac.za

Steve Kroon
Computer Science Division
Stellenbosch University
7602 Matieland
South Africa
kroon@sun.ac.za

Gert-Jan van Rooyen
E&E Engineering Department
Stellenbosch University
7602 Matieland
South Africa
gvrooyen@sun.ac.za

Cornelia P. Inggs
Computer Science Division
Stellenbosch University
7602 Matieland
South Africa
cinggs@cs.sun.ac.za

ABSTRACT

Parallelisation of computationally expensive algorithms, such as Monte-Carlo Tree Search (MCTS), has become increasingly important in order to increase algorithm performance by making use of commonplace parallel hardware.

Oakfoam, an MCTS-based Computer Go player, was extended to support parallel processing on multi-core and cluster systems. This was done using tree parallelisation for multi-core systems and root parallelisation for cluster systems.

Multi-core parallelisation scaled linearly on the tested hardware on 9x9 and 19x19 boards when using the virtual loss modification. Cluster parallelisation showed poor results on 9x9 boards, but scaled well on 19x19 boards, where it achieved a four-node ideal strength increase on eight nodes.

Due to this work, Oakfoam is currently one of only two open-source MCTS-based Computer Go players with cluster parallelisation, and the only one using the Message Passing Interface (MPI) standard.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming, parallel programming*

General Terms

Experimentation, Performance

Keywords

Monte-Carlo Tree Search, Computer Go, parallelisation

1. INTRODUCTION

Due to physical constraints, modern processors are making increasing use of parallel hardware to increase processing power [1]. This has given increasing importance to the parallelisation of computationally expensive algorithms, such as Monte-Carlo Tree Search (MCTS) [2, 3, 4, 5], in order to efficiently use this parallel hardware.

The aim of this work is to implement and evaluate parallelisation of MCTS for Computer Go, for multi-core and cluster systems. This parallelisation is non-trivial as the MCTS tree, which dictates the work done in the various processing nodes, must be responsive to updates received from those nodes.

The work in this paper confirms previous work and includes new experimental results of parallelising the stochastic MCTS algorithm for Computer Go.

2. BACKGROUND AND RELATED WORK

2.1 The Game of Go

Go (otherwise known as Weiqi, Baduk, and Igo) is an ancient game in which two players alternate placing black and white stones on empty intersections of the board, a rectangular grid [6]. Orthogonally adjacent stones of the same colour form chains. After a move, chains left with 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 [6].

Even though the rules of Go are simple, the game has great tactical and strategic depth [7]. This emergent complexity of Go is what makes Go enjoyable for many Go players, but also contributes to it being so difficult for a computer to reach competitive levels of play [8].

To play Go at a non-trivial level, humans often create tree-like structures in their minds [9]. These trees consist of board positions as nodes in the tree, with children of a node being positions that occur after valid moves from the position represented by the node's parent. The act of forming and evaluating such a tree for Go is called *reading* [9]. Due to the additive nature of Go — pieces are added to the current board position, not moved — it is relatively easy for humans to read ahead [9], compared to games like Chess, in which pieces move around. Michael Redmond, a professional Go player, has stated that he can 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 [7].

As with most games, there is no absolute measure of playing strength in Go. Rather, ratings are awarded to Go players based on their performance in previous games. In this work, we measure the relative strength of two players by playing a number of games between them.

2.2 Computer Go

Computer Go refers to the development of computer programs able to play the game of Go. In a number of other games, such as Chess and Othello, computers have surpassed human players in skill [8]. However, Computer Go has not yet achieved the same dominance reached in those games [4, 5]. This is partially due to the complexity of Go — the branching factor of Go is over 100 on a 19x19 board for most of the game, whereas the branching factor of Chess is closer to 20 [8].

2.3 Game Trees

Game trees are used in computer players, for games such as Go, to plan ahead and select moves. They are tree structures, with nodes representing positions and edges representing moves that lead to other positions [10]. Game trees can 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 using techniques like *minimax* or *negamax* [10].

A complete game tree (one that contains all possible game sequences) will reveal perfect play[1] when minimax or negamax is performed on it. However, this requires too much memory and processing for most games (particularly Go) so it is not usually attempted [8]. Instead, techniques such as *alpha-beta pruning* [10] are used to reduce the computational resources required and make the tree search feasible.

Classical approaches to Computer Go tried to replicate the thought process that humans use to grow and evaluate a game tree, using minimax or negamax with alpha-beta pruning and a function evaluating positions [11]. This was not very successful, as the expert knowledge that had to be hand-coded and maintained for an accurate evaluation function became too complex, and thus difficult to extend [11]. In the last decade, MCTS has found popularity in Computer Go by outperforming such classical computer game-playing techniques [3, 4, 5].

2.4 Monte-Carlo Tree Search

Monte-Carlo simulations are stochastic simulations of a model [12]. Through repetition, such simulations can provide statistically significant information and can be useful for problems that do not have a known deterministic solution [12].

In the context of Computer Go, these simulations, often referred to as *playouts*, simulate a game of Go from some initial position until the end of the game, by selecting moves for each player stochastically [13]. Playouts usually make use of heuristics to bias the distribution for move selection from a given position [4, 5]. Once the end of the game is reached, it is straightforward to score the position and determine the winner. If a number of playouts are performed, starting from the same position, then the ratio of wins to losses can form an evaluation of that position. Even though these playouts only make use of the game rules and simple heuristics to select moves, through repetition they are able to provide valuable information regarding the relative quality of moves [3, 13].

To improve the performance of Monte-Carlo simulations in adversarial scenarios, they were combined with game tree search to form Monte-Carlo Tree Search (MCTS) [3, 5]. Besides its early application to Computer Go, MCTS has been applied to various other domains, including General Game Playing (GGP) [3, 5, 14].

MCTS begins by creating a tree with the current game position as the root node. Each node in the tree will store the number of wins and losses for playouts beginning from a descendant of that node [3]. Figure 1 shows an example MCTS tree. All leaf nodes and nodes that can add another valid child[2] form the *frontier* of the MCTS tree.

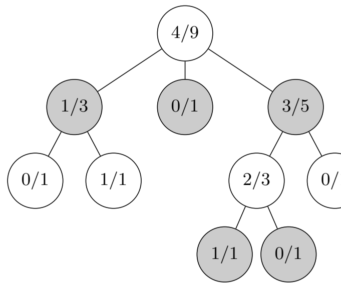

Figure 1: Example MCTS tree. Nodes show the number of playout wins over the total number of playouts from descendants of that node (from the perspective of one player). Shaded nodes indicate the opponent will play next from this position.

The MCTS algorithm consists of a number of iterations of four steps: selection, expansion, simulation and backpropagation [15].

Selection is the process of descending the tree to a frontier node (see Figure 2). Each node on the descent path is selected according to a selection policy. This policy has to balance *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. Upper Confidence Bounds (UCB) was the first selection policy used [3], but recently other selection policies with better empirical performance are typically used [5].

Once the selection process has stopped descending, *expansion* is performed by adding a child to the frontier node reached (see Figure 3). In this way, the tree is constantly expanding, looking further into the possible future. Expansion increases the accuracy and relevance of the tree by making it a more realistic representation of possible outcomes.

Simulation is the process of performing a playout starting from the new child node added to the frontier in the expansion step (see Figure 4).

[1]Perfect play is to make the best move possible each turn.

[2]Not all valid moves from a position are added to the tree immediately, or even at the same time.

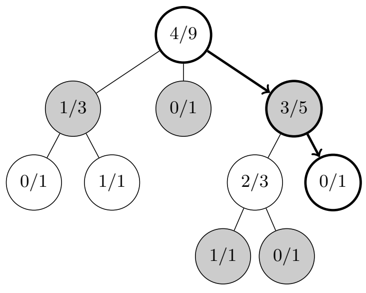

Figure 2: MCTS selection, showing the process of descending the tree.

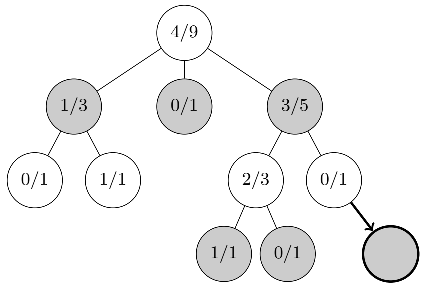

Figure 3: MCTS expansion, showing the process of adding a node to the frontier of the tree.

In the last step, *backpropagation*, the simulation result from the new frontier node is propagated back up the tree until the root is reached (see Figure 5).

When a stop condition is met, such as a specific number of playouts having occurred, or a specific time having passed, the MCTS search can be stopped and the best move selected according to some criteria. A child of the root node is selected, representing a valid move to a new position from the current position. It has been shown that, using better performing selection policies, selecting the child with the most playouts is a more robust method than the child with the highest win-loss ratio [5].

The process of descending through a node and later adding a playout result to that node can be viewed as sampling from a Bernoulli random distribution. However, as the tree expands over time, the parameter of this distribution changes and the sequence of playout results therefore forms a non-stationary process. This non-stationarity plays an important role in parallelisation.

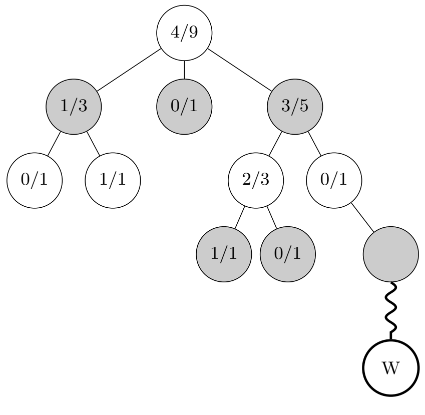

Figure 4: MCTS simulation, showing a playout performed with a win (W) result.

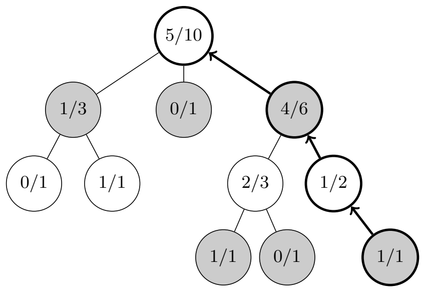

Figure 5: MCTS backpropagation, showing how the simulation result of Figure 4 propagates up the tree.

It has been shown that given an increase in the number of playouts, MCTS increases in strength [16]. This increase in playouts can be achieved through an increase in thinking time or an increase in the rate of playouts. An increase in thinking time is usually not an option, but an increase in the rate of playouts can be accomplished through optimisation or parallelisation.

2.5 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 simultaneously making use of a number of processing nodes. These nodes can be central processing unit (CPU) cores on a symmetric multiprocessing (SMP) (multi-core) machine or spread out over a number of machines in a cluster. We say a method scales to a certain number of nodes when it is stronger running on that many nodes than a version running on one less node.

MCTS can be parallelised using various methods, the major ones being tree parallelisation, leaf parallelisation, and root parallelisation [5, 17]. One issue that is common to all of these methods, in varying degrees, is the parallel effect.

2.5.1 Parallel Effect

It is important to note that parallelising a well-tuned MCTS implementation will usually lead to a loss in playing strength when the total number of playouts is fixed: because the process of node selection is non-stationary, a parallel implementation will select nodes and perform playouts from them differently to a sequential implementation. For example: in a parallel implementation a number of processing nodes might perform playouts in parallel starting from the same node, only to find that the node is not favourable once all the playouts have completed; in a sequential implementation the first playout might have shown that the node is not favourable and subsequent playouts would then have explored other nodes of the tree. The loss in strength resulting from parallelising a fixed number of playouts is referred to as the parallel effect in this paper.

2.5.2 Tree Parallelisation

In tree parallelisation, a shared MCTS tree is used by all processing nodes (see Figure 6) [5, 17, 18]. This implies that the tree is constantly available to 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. However, memory access latency is not a problem for multi-core systems, and tree parallelisation is generally used on these systems [5].

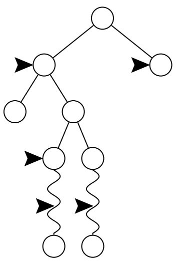

Figure 6: Tree parallelisation, showing a number of processing nodes working on a shared MCTS tree simultaneously.

To prevent data loss with tree parallelisation, concurrency primitives such as mutexes are generally used. These mutexes control access to the data stored in the tree 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 processing nodes are used, this can lead to time being wasted waiting to acquire mutexes. The *lock-free* modification dictates that tree node mutex locks are not acquired [19]. This means that multiple processing nodes can access a tree node's data simultaneously. This lack of mutexes can lead to some slight inaccuracies and inconsistencies in playout statistics, but the processing power made available by not waiting for mutex locks may outweigh the losses resulting from these issues [19].

A problem that can arise with tree parallelisation is over-exploitation, where processing nodes duplicate work being done at the same time on other processing nodes. This is a manifestation of the parallel effect. The *virtual loss* modification attempts to mitigate this problem. It involves adding a loss to each node in the tree visited during the selection descent, and then removing it when propagating the result back up the tree [17]. This deters other processing nodes from descending down the same path in the tree and potentially performing unnecessary work if there is another path which is of similar quality. This modification thus encourages exploration.

Enzenberger and Müller showed that they could improve the scaling of tree parallelisation from 3 to 8 nodes with the lock-free modification [19]. Segal has shown tree parallelisation with the lock-free and virtual loss modifications to scale to 64 nodes while it was limited to 8 nodes without virtual loss [16].

2.5.3 *Leaf Parallelisation*

Leaf parallelisation employs a single master node and multiple slave nodes (see Figure 7) [5, 17]. The master node maintains the MCTS tree and requests that slave nodes 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. In leaf parallelisation, the four steps of an MCTS iteration are thus divided between the master and slave: the slave performs the simulation step, while the master performs the other three steps.

This method can be successful on clusters, but the single node maintaining the tree can potentially become a bottleneck [20]. Kato and Takeuchi have shown leaf parallelisation to scale to one master node and 15 slave nodes [20].

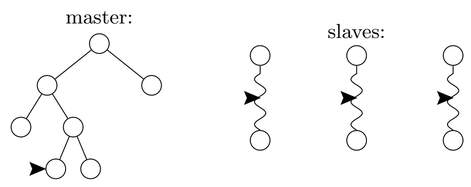

Figure 7: Leaf parallelisation, showing the master and slave nodes, with the slaves performing the playouts for the master's MCTS tree.

2.5.4 *Root Parallelisation*

In root parallelisation, each node maintains its own MCTS tree with periodic sharing of information about these trees between the nodes (see Figure 8) [5, 17]. When information is shared, only a portion of the tree is shared in order to minimise the communication overhead of sharing. A possible sharing strategy is to share the nodes in the top 3 levels that have at least 5% of the total playouts through them, at a frequency of 3 Hz [21]. In this method, each of the processing nodes performs all four steps of the MCTS iterations on its tree. The sharing frequency must balance the communication overhead with keeping the MCTS trees as relevant as possible. If root parallelisation could update at infinite frequency, share the whole tree, and update in zero time, then it would be equivalent to tree parallelisation. Bourki et al. have shown root parallelisation to scale to 40 nodes [21].

Figure 8: Root parallelisation, showing a MCTS tree per processing node.

3. DESIGN AND IMPLEMENTATION

3.1 General

This work builds on the existing MCTS-based Computer Go player, *Oakfoam* [22]. Oakfoam is implemented in C++ and is available under the BSD open-source licence. Oakfoam has a number of strength-improving modifications to the vanilla MCTS algorithm, including an improved selection policy and playout heuristics. Version 0.1.0 of Oakfoam was used for this work. Prior to this work, Oakfoam did not have parallelisation functionality. More extensive discussion of this work is available in [23]. Refer to Section 8 for more information on Oakfoam.

3.2 Multi-core Parallelisation

Due to the shared memory nature of multi-core systems, it was decided to make use of tree parallelisation, as described in Section 2.5.2. In this case, the processing nodes are the CPU cores of the multi-core system. A single MCTS tree, which is shared between all the processing nodes, is therefore maintained in shared memory. A number of threads (typically one per CPU core) are executed, and all work on the same tree. In order to address over-exploitation and performance losses from waiting for concurrency primitives, the virtual loss and lock-free modifications were implemented and these can be enabled at runtime.

Multi-core parallelisation requires threading, and Boost C++ Threads [24] were used for this. When an MCTS search is performed, a number of threads, stored in a thread pool, are started and simultaneously begin an MCTS search. The number of threads is set using a parameter, and can easily be set to the number of CPU cores on the current machine. Once a stop condition is met, all the threads are stopped and the final result is returned.

3.3 Cluster Parallelisation

For clusters, the processing nodes are CPU cores distributed over a collection of machines. In comparison with multi-core systems, clusters have high latency between processing nodes as the nodes are usually not on the same physical machine. Leaf and root parallelisation are therefore the only realistic candidates for cluster parallelisation. Root parallelisation, as described in Section 2.5.4, was chosen due to it scaling better than leaf parallelisation in previous work [21, 25].

For root parallelisation, each processing node will periodically share a part of their tree with the rest of the nodes. The communication system used is very important for this sharing. The two main options that are available for cluster communication are using the Message Passing Interface (MPI) and building a layer atop Transmission Control Protocol (TCP).

The MPI standard is the de-facto communication standard for High Performance Computing (HPC) [26]. It is relatively easy to use and has tried-and-tested implementations with support [27]. MPI implementations have support for TCP/IP connections as an underlying communication mechanism [27]. One advantage of MPI is that, given faster communication mechanisms between processing nodes than TCP/IP (such as shared memory or switched fabric communication), MPI can make use of them without additional code [27]. Another advantage is that MPI is available on most High Performance Computing (HPC) clusters [28], and greatly simplifies creating and running cluster jobs. A notable restriction of the current MPI standard is that all collective communications (any communications that involve more than two nodes) must be synchronous. Since pairwise communication is not feasible for many nodes, this implies that all the nodes will have to communicate at very specific times when sharing data.

For finer-grained control of communication between processing nodes, a custom communication layer can be implemented atop TCP. Although this approach is more complex, it allows one to make full use of the available network as well as handle asynchronous and synchronous communication.

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 to communicate as close to each other in time as possible, reducing time spent waiting.

It was thus decided to use MPI, as the reduced complexity was deemed more important than the flexibility lost and the lack of asynchronous collective communications. It was decided to use Open MPI [27], an open-source implementation with support and widespread usage [28], as the MPI implementation.

In order to synchronise communications, a check is performed after each playout to see whether the next update should occur, or another playout should be started. Therefore, the longest a processing node should wait to communicate is the length of one playout. An optimised MCTS implementation can typically perform at least 1000 playouts per second, so if the number of playouts completed between updates is more than 100 playouts (which would correspond with a sharing rate of about 10Hz), the total waiting time should thus be smaller than 1%.

In order to limit communication overhead, only a portion of the tree is shared. The portion of the tree shared is based on the top few tree levels and how many playouts have passed through the node; this approach was chosen based on other cluster root parallelisation implementations [21]. The exact number of levels and playouts that signify the shared portion are adjustable parameters. Figure 9 shows an example MCTS tree, with a portion that would typically be shared indicated. In the example, only nodes in the top three tree levels with at least 20% of the total playouts have been selected.[3]

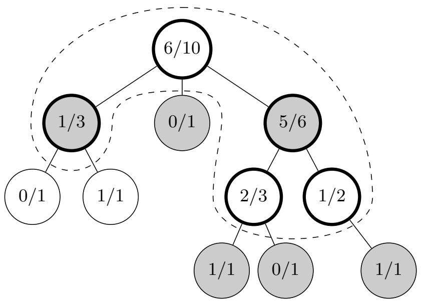

Figure 9: Example of cluster sharing. Nodes to be shared are indicated. These nodes are in the top three tree levels and have at least 20% of the total playouts.

The shared tree portions will be combined after sharing. The playout results of each equivalent position will be combined by adding the playout results since the last time shared.

In order to share a node, a unique identifier for the position represented by the node must be shared, as well as the playout results for that position since the last time it was

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

shared. This will be enough information to combine playout results for equivalent nodes in the different trees. A 64-bit Zobrist hash [29] is used as this unique identifier. Zobrist hashing is a technique that can generate an arbitrary length hash code for a board position with a good distribution [29]. It is assumed that the distribution of these hashes will be such that any collisions will not substantially affect performance.

4. EXPERIMENTS AND RESULTS

4.1 Overview

Due to the stochastic nature of MCTS, it is difficult to verify the correctness of an MCTS implementation. It is therefore common to verify an MCTS implementation with empirical results — testing is a fundamental part of MCTS research [5].

Usually, changes to an MCTS implementation are evaluated by playing against a version of the same program without the change (self-play) and against other reference programs. In this work, 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 additional complexity of introducing another program. The Gomill tool suite [30] was used for testing.

Tests were performed on 9x9 and 19x19 boards, as these are popular board sizes [6]. Using two board sizes should give a good indication of the effect of board size on the results. Note that, since a 19x19 board is about 4.5 times larger than a 9x9 board, playouts take more than four times longer on 19x19. Therefore 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 work. There are 100 or more moves in a typical 19x19 Go game, and with a time limit of 10 seconds per move,[4] tests on 19x19 can take longer than 15 minutes per game. In order to get an accurate result, a number of games must be played. A run of 100 games (the minimum number for a test in this work) can easily take longer than a day on 19x19. Testing was therefore selective and could not cover a very wide range of parameters.

In order to evaluate the parallelisation implementations, a sequence of tests was performed. We started by measuring the speedup — the rate of playouts with a specific number of processing nodes over the rate of playouts on a single processing node. We did this by starting Oakfoam on a certain number of processing nodes, sending a command to generate a move, and then recording the rate of playouts. If no increase in speedup is observed beyond a certain number of nodes, we know that the implementation being tested does not scale past that many nodes.

We continued by measuring the parallel effect (see Section 2.5.1). For this test, two versions of Oakfoam using a fixed number of playouts per move played a series of games against each other and the winrate[5] of the tested version was measured. The reference version was run on a single node, while the tested version was run on various numbers of processing nodes. A 50% winrate is ideal, indicating that there is no parallel effect and that the tested version scales according to the speedup results.

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

[5]The winrate is the number of wins divided by the total number of games in the series.

When near-ideal results were not found, a further test was performed to measure any possible strength increase. Similar to the previous test, a series of games between two versions of Oakfoam was played. However, in this strength comparison test, both versions are given a fixed thinking time per move. This test requires a baseline for comparison. This baseline is generated by emulating an ideal parallel player — the tested version is given a thinking time per move equal to that of the reference version multiplied by the number of nodes the ideal player is emulating executing on.

The speedup results showed very little variance, with only the occasional slower outlier, most likely due to system processes in the background. No error bars are shown for these results, only the fastest result of 4 samples is shown.

In all graphs that show error bars, these bars show the 95% confidence interval[6] of each result. In some of the graphs, particularly those with error bars, the results are slightly staggered to aid readability. For example, all results in Figure 17 near the grid line for "4 Nodes/Periods" are in fact measured for exactly 4 nodes or periods.

All tests were performed on the Stellenbosch University's *Rhasatsha* cluster [31] and another standalone machine. The cluster nodes have eight-core Intel Xeon processors ranging from 2.67 GHz to 2.83 GHz and the standalone machine has a four-core Intel Xeon E5520 2.27 GHz processor.

4.2 Multi-core Parallelisation

Multi-core parallelisation was evaluated by measuring its speedup and parallel effect. All multi-core testing was performed on the eight-core nodes of the Rhasatsha cluster. While we had hoped to test multi-core scaling beyond eight cores, our tree parallelisation implementation was unfortunately unable to scale to even two threads on the many-core SPARC-architecture machine we had planned to use for this purpose. This issue seems to be due to a difference between x86 and SPARC architectures, which we hope to address in our implementation in future work.

4.2.1 Speedup

The speedup of the multi-core tree parallelisation implementation (with various combinations of the virtual loss and lock-free modifications) on 9x9 and 19x19 was measured. The results are shown in Figures 10 and 11.

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

There is no discernible difference between the various multi-core versions in the range of hardware tested on and therefore no conclusion can be made regarding the utility of the virtual loss and lock-free modifications on the playout rate.

4.2.2 Parallel Effect

The multi-core tree parallelisation implementation was tested with different modifications on 9x9 and 19x19 to measure the parallel effect. All of these tests used 10 000 playouts per move. The results are shown in Figures 12 and 13.

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

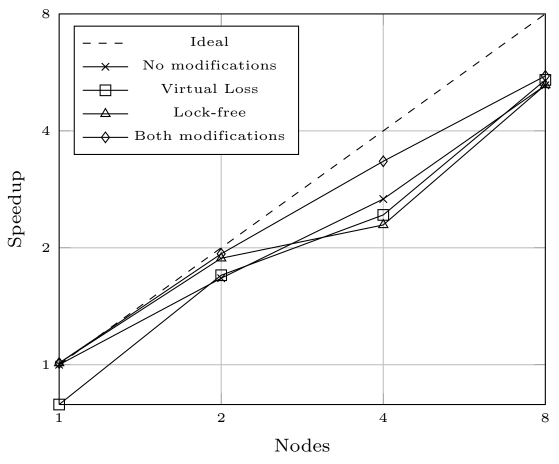

Figure 10: Speedup of multi-core parallelisation on 9x9 with various modifications.

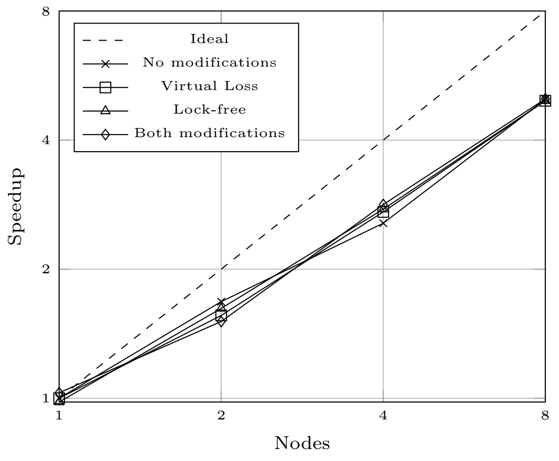

Figure 11: Speedup of multi-core parallelisation on 19x19 with various modifications.

On the tested hardware, multi-core parallelisation shows no noticeable parallel effect on 9x9. On 19x19, the multi-core parallelisation shows no parallel effect with the virtual loss modification, while without the modification, strength is severely handicapped by the parallel effect.

The lack of the parallel effect (with virtual loss) and linear speedup implies that our multi-core parallelisation scales very well on the available hardware and any strength increase can be measured by the corresponding increase in the rate of playouts. Therefore it was not considered necessary to perform a strength comparison for multi-core parallelisation.

4.3 Cluster Parallelisation

Cluster parallelisation was evaluated by performing the speedup, parallel effect and strength comparison tests on our root parallelisation implementation.

The interval between tree-sharing updates is a parameter for some of the tests and is referred to as p (in seconds). The portion of the tree shared is the same in all the tests and consists of the nodes with more than 5% of the total tree playouts that are in the top three tree levels.

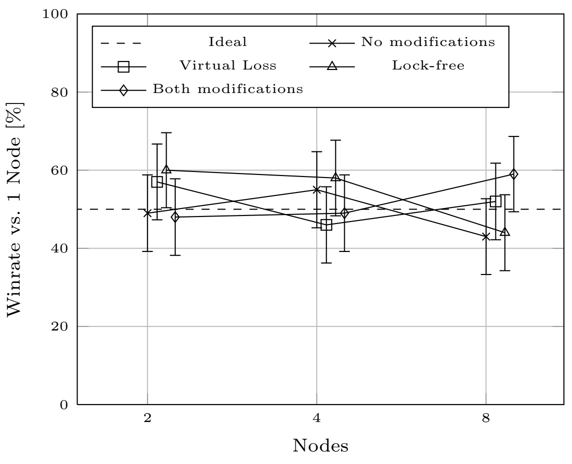

Figure 12: Parallel effect of multi-core parallelisation on 9x9 with various modifications and 10 000 playouts per move.

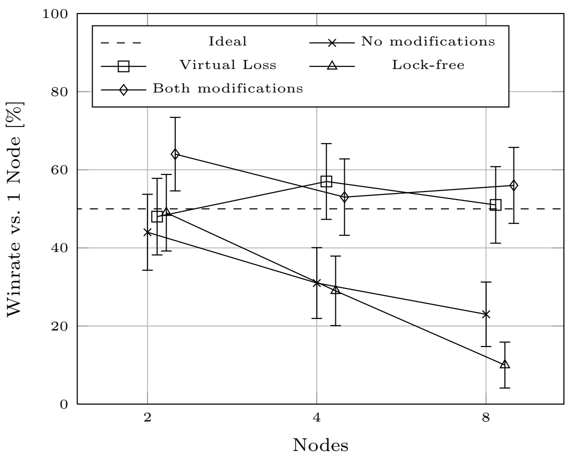

Figure 13: Parallel effect of multi-core parallelisation on 19x19 with various modifications and 10 000 playouts per move.

4.3.1 Speedup

The speedup of the cluster root parallelisation implementation on 9x9 and 19x19 was measured. The results are shown in Figures 14 and 15.

The results show almost ideal speedup for cluster parallelisation on 9x9. On 19x19, the speedup is close to linear, but with a lower gradient. We believe that this is due to the increased length of a playout on 19x19 and the fact that sharing is synchronised. This means that more time is spent waiting to synchronise on 19x19 if the sharing period is the same as on 9x9. We can also see that after a number of nodes, the speedup plateaus. We believe that this is due to the communication overhead starting to dominate.

4.3.2 Parallel Effect

The cluster root parallelisation implementation was tested on 9x9 and 19x19 to measure the parallel effect. All of these tests were conducted with a sharing interval of $p = 0.1$. The results are shown in Figure 16.

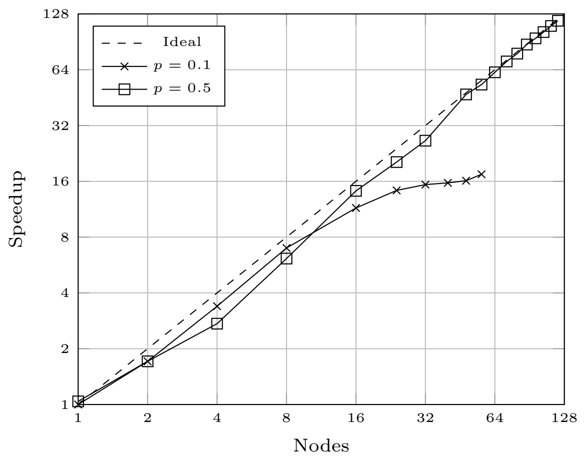

Figure 14: Speedup of cluster parallelisation on 9x9 with various sharing intervals.

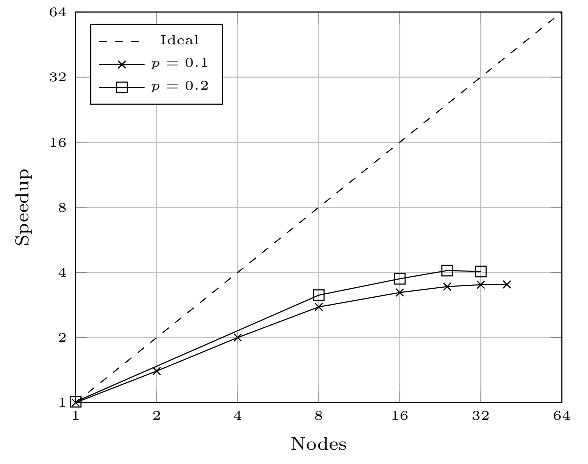

Figure 15: Speedup of cluster parallelisation on 19x19 with various sharing intervals.

The parallel effect is prominent for cluster parallelisation on 9x9, unlike for multi-core parallelisation. Since we expect 19x19 to be more susceptible to the parallel effect, it was decided not to perform further parallel effect tests and instead proceed with a strength comparison on 9x9 and 19x19.

4.3.3 Strength Comparison

The strength of the cluster root parallelisation implementation on 9x9 and 19x19 was measured and compared to an ideal baseline. These tests were performed using different sharing intervals. The results are shown in Figures 17, 18 and 19.

The results on 9x9 show that our root parallelisation implementation is not very successful and is not able to scale well on the cluster. On 19x19, the implementation is able to scale up to eight nodes, where it achieves a strength equivalent to four nodes assuming ideal parallelisation. We also note that running on too many nodes on 19x19 may be detrimental to strength, as indicated by the results for 32 nodes being worse than those on 16 nodes, although not by a statistically significant amount.

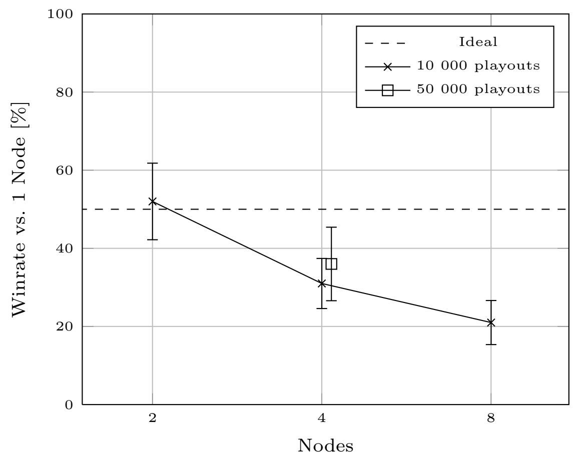

Figure 16: Parallel effect of cluster parallelisation on 9x9 with a fixed number of playouts per move.

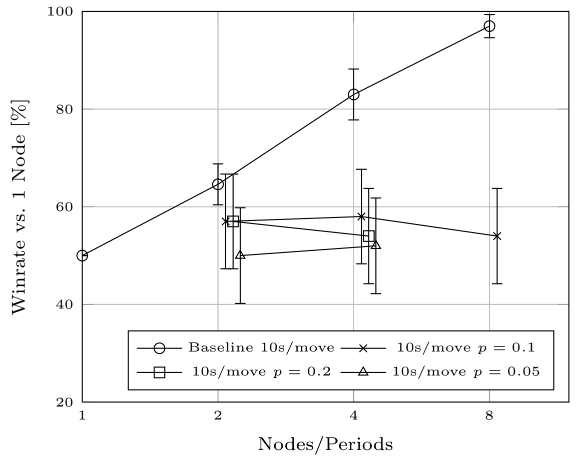

Figure 17: Strength of cluster parallelisation on 9x9 with longer time settings and various sharing intervals.

5. CONCLUSIONS

MCTS is the focus of current Computer Go research. Currently, processors are making increasing use of parallel hardware, making parallelisation more important. This work implemented and tested solutions for parallelising of MCTS.

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 near-ideal scaling on the tested hardware, similar to results in other work [16, 19]. Future work on multi-core parallelisation can continue testing on a greater number of cores.

Cluster parallelisation was tested on up to 8 and 32 nodes on 9x9 and 19x19 respectively. Minimal to no strength improvement was observed on 9x9, similar to results previously reported [21]. However, on 19x19 we observed scaling up to eight nodes, where performance was equivalent in strength to the ideal for four nodes. This is worse than, but comparable to, results in other work [21]. This testing shows that

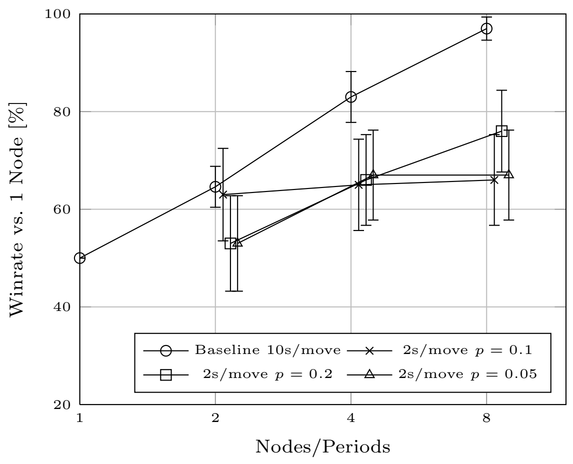

Figure 18: Strength of cluster parallelisation on 9x9 with shorter time settings and various sharing intervals.

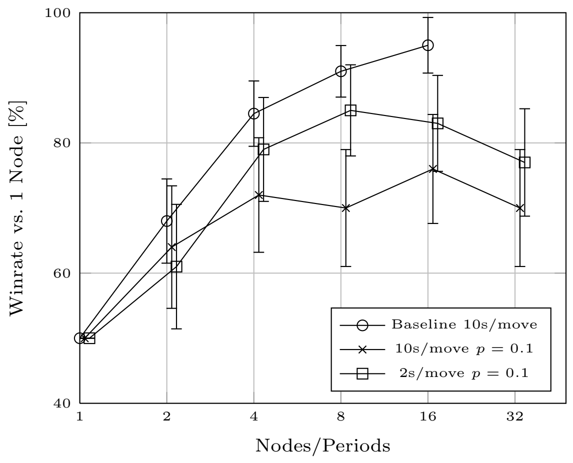

Figure 19: Strength of cluster parallelisation on 19x19 with various time settings.

there is much room for improvement on both 9x9 and 19x19 for cluster parallelisation in terms of ideal scaling. Future work on cluster parallelisation can consider different sharing criteria or other parallelisation methods.

Due to this work, Oakfoam is currently one of only two[7] open-source MCTS-based Computer Go players with cluster parallelisation, and the only one using MPI.

6. ACKNOWLEDGEMENTS

This work was partially supported by the National Research Foundation of South Africa.

7. REFERENCES

[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] B. Brügmann, “Monte Carlo Go,” tech. rep., Max-Planck-Institute of Physics, 1993.

[3] L. Kocsis and C. Szepesvári, “Bandit based Monte-Carlo Planning,” *Machine Learning: ECML 2006*, pp. 282–293, 2006.

[4] A. Rimmel, O. Teytaud, C.-S. Lee, S.-J. Yen, M.-H. Wang, and S.-R. Tsai, “Current Frontiers in Computer Go,” *IEEE Symposium on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 229–238, 2010.

[5] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, “A Survey of Monte Carlo Tree Search Methods,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–49, 2012.

[6] K. Baker, *The Way to Go.* American Go Foundation, 1986.

[7] 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/`, accessed on 2011-10-23, 2010.

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

[9] “Reading.” Sensei’s Library, `http://senseis.xmp.net/?Reading`, accessed on 2011-10-23.

[10] N. J. Nilsson, *Principles of Artificial Intelligence.* Tioga Publishing Company, 1980.

[11] B. Bouzy and T. Cazenave, “Computer Go: An AI oriented survey,” *Artificial Intelligence*, vol. 132, pp. 39–103, Oct. 2001.

[12] N. Metropolis and S. Ulam, “The Monte Carlo Method,” *Journal of the American Statistical Association*, vol. 44, no. 247, pp. 335–341, 1949.

[13] B. Bouzy and B. Helmstetter, “Monte-Carlo Go Developments,” in *Advances in Computer Games*, 2003.

[14] J. Méhat 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, pp. 271–277, 2010.

[15] G. M. J.-B. Chaslot, *Monte-Carlo Tree Search.* PhD thesis, Maastricht University, 2010.

[16] R. Segal, “On the Scalability of Parallel UCT,” in *Computers and Games*, pp. 36–47, Springer, 2011.

[17] G. M. J.-B. Chaslot, M. H. M. Winands, and H. van den Herik, “Parallel Monte-Carlo Tree Search,” *Computers and Games*, pp. 60–71, 2008.

[18] S. Gelly and Y. Wang, “Exploration Exploitation in Go: UCT for Monte-Carlo Go,” in *NIPS Conference On-line trading of Exploration and Exploitation Workshop*, 2006.

[19] M. Enzenberger and M. Müller, “A Lock-free Multithreaded Monte-Carlo Tree Search Algorithm,” *Advances in Computer Games*, pp. 14–20, 2010.

[7]The open-source Computer Go player Pachi has cluster parallelisation using TCP.

[20] H. Kato and I. Takeuchi, "Parallel Monte-Carlo Tree Search with Simulation Servers," *13th Game Programming Workshop (GPW-08)*, 2008.

[21] A. Bourki, G. M. J.-B. Chaslot, M. Coulm, V. Danjean, H. Doghmen, J.-B. Hoock, T. Hérault, A. Rimmel, F. Teytaud, O. Teytaud, P. Vayssiere, and Z. Yu, "Scalability and Parallelization of Monte-Carlo Tree Search," in *Computers and Games*, pp. 48–58, Springer, 2010.

[22] "Oakfoam." `http://bitbucket.org/francoisvn/oakfoam`.

[23] F. Van Niekerk, *MCTS Parallelisation.* Engineering final year project, Stellenbosch University, 2011.

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

[25] K. Rocki and R. Suda, "Massively Parallel Monte Carlo Tree Search," in *VECPAR 2010, 9th International Meeting High Performance Computing for Computational Science*, 2010.

[26] J. Kepner, "Parallel programming with MatlabMPI." `http://www.ll.mit.edu/mission/isr/matlabmpi/matlabmpi.html`, accessed on 2011-10-23.

[27] "Open MPI: Open source high performance computing." `http://www.open-mpi.org/`, accessed on 2011-10-23.

[28] "Open MPI: FAQ." `http://www.open-mpi.org/faq`, accessed on 2011-10-23.

[29] A. Zobrist, "A new hashing method with application for game playing," *ICGA Journal*, vol. 13, no. 2, pp. 69–73, 1970.

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

[31] "Rhasatsha cluster." `http://www.sun.ac.za/hpc`.

8. APPENDIX: SOURCE CODE

All source code that was used in this work is part of Oakfoam, an open-source MCTS-based Computer Go player. Version 0.1.0 was used for the work in this paper and is tagged in the code repository. All default parameters were used unless specified otherwise. Oakfoam is available for download at: `http://bitbucket.org/francoisvn/oakfoam`