KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in the Field of Technology and the Main Field of Study
Technology

First cycle, 15 credits

Young Brothers Wait Concept or Lazy SMP?

A Comparative Study of Parallel Search Techniques in a Chess Engine

ALEX SÄFSTRÖM
SIMON YLLMARK

Stockholm, Sweden, 2024

Young Brothers Wait Concept or Lazy SMP?

A Comparative Study of Parallel Search Techniques in a Chess Engine

ALEX SÄFSTRÖM

SIMON YLLMARK

Degree Programme in Computer Engineering
Date: November 20, 2024

Supervisor: Aws Al-Zarqawee
Examiner: Ki Won Sung
School of Electrical Engineering and Computer Science
Swedish title: Young Brothers Wait Concept eller Lazy SMP?
Swedish subtitle: En Jämförande Studie av Parallella Söktekniker i en Schackmotor

Abstract

Keywords

Chess Engine, Young Brothers Wait Concept, Lazy SMP, Parallel Search, Scalability, Elo Rating

Sammanfattning

Acknowledgements

I would like to thank xxxx for having yyyy. Or in the case of two authors: We would like to thank xxxx for having yyyy.

Stockholm, November 2024
Alex Säfström Simon Yllmark

Innehåll

Figurer

Tabeller

Listings

List of acronyms and abbreviations

Bb	Bitboards
BS	Board State
CB	Chess-Bot
CCRL	Computer Chess Rating Lists
CE	Chess Engine
CNN	Convolutional Neural Networks
ER	Elo Rating
Eval	Evaluation Function
FOSS	Free and Open Source Software
GM	Grand Master
GUI	Graphical User Interface
Lazy SMP	Lazy Symmetric Multiprocessing
MT	Multi-Threading
PSVT	Piece Square Value Table
Scal	Scalability
SEF	Static Evaluation Function
SIMD	Single Instruction Multiple Data
SMP	Symmetric Multiprocessing
Sped	Speedup
UCI	UCI protocol
YBWC	Young Brothers Wait Concept

Chapter 1

Introduction

Despite its age, chess continues to attract more players and is today more popular than ever [1]. Many players use computers to play against their opponents online, practice against the computer, or study and learn from previous games. When using a computer to play chess a chess engine can be used to be played against or to analyse the moves of a game, and is therefore an indispensable tool to these players. Historically, a large portion of the performance gains over time for chess engines has come from the increased performance of the computer hardware they run on.

Due to the heat generated by increasing clock speeds of processors and the difficulty of shrinking the size of transistors, much performance is today gained from having multiple cores in the same processor. It is however not as easy to write a program that utilises multiple cores, and today chess engines use a variety of techniques to achieve this. This study aims to modify an open-source, multi-threaded chess engine so that it can choose between two of the most popular techniques, Young Brothers Wait Concept and Lazy SMP, to compare how the choice between the two affects the performance, playing strength, and Scalability (Scal) of the engine.

1.1 Background

Chess can trace its history back to the game of Chaturanga, an Indian board game from at least 500 B.C. Chaturanga spread worldwide through travelling traders, becoming altered and adapted by locals as time went on. Among others, this led to the creation of Xiangqi in China, Shogi in Japan, and eventually early versions of chess in Europe. By the 15th century, the chess of the time could be considered relatively close to the modern game but would

continue to have its rules standardised throughout Europe well into the 1800s [2].

During the latter half of the 19th century, the most prevalent theory transitioned from a very aggressive, attack-heavy style to one based on positional advantage. By the 1920s this would start to, in turn, be replaced by hypermodernism, in which control of the centre with pieces is favoured over direct occupation with pawns. There had been several world championships, but it was not until 1948 that they would be standardised by FIDE (The International Chess Federation) [3].

It was also around this time that interest in creating machines that could play chess would start to materialise, with Claude E. Shannon in 1950 publishing one of the first papers that described the challenges surrounding creating a machine that would be able to play ”[...]a tolerably good game of chess[...]” [4].

Shannon explains that simple games can have an evaluation function that can calculate if any given position will lead to a win, loss, or draw and that a machine could be programmed to use this evaluation function with the positions from each possible move from the current position to always play perfectly. He gives Nim as an example of a game that has been solved like this and reasons that this cannot be done in practice with chess due to the humongous number of possible moves. Shannon calculates that with the average of around 30 legal moves from any position and the average game length of 40 moves before resignation, a computer would need to calculate 10^{120} positions and that this would take more than 10^{90} years even if one position could be calculated every picosecond (10^{-12} seconds).

He further points out that if one were to use a dictionary of all possible positions, that would require storing around 10^{43} unique positions. To solve this, he concludes that a chess program must use strategy to be able to play, suggesting the use of an approximate evaluation function that at least takes into account the material worth of pieces, their positioning, and their mobility to assign a score to any given position. The program should then generate a tree of all legal moves for a few moves ahead, evaluate the resulting positions at the bottom of the tree, and propagate the scores up the tree to the root by choosing the highest scores for white’s turn and the smallest scores for black’s turn, what would today be called a minimax search algorithm.

Around the same time, Alan Turing became the first person to publish a program that could play an entire game of chess, however, no computer at the

time was strong enough to run it. Turing played a match against Alick Glennie in 1952 by calculating the algorithm's moves by hand but ended up losing after 29 moves [5].

The exponential increase in the processing power of computers as well as algorithmic improvements ensured that the playing strength of chess programs would continue to grow. In 1985, the reigning chess world champion Garry Kasparov was able to play against 32 of the strongest chess engines in the world simultaneously and win against all of them [6, ∼06:00]. Only 12 years later in 1997, would Kasparov be the first world champion to lose against a chess engine, IBM's Deep Blue, by $3\frac{1}{2} - 2\frac{1}{2}$ [7].

A Chess Engine (CE) can be used for other things than just for a Chess-Bot (CB). Many chess players use a CE as a tool for analysing their chess game to improve, which Garry Kasparov was one of the first Grand Master (GM)s to do [8].

There are many CEs today such as Stockfish, Alpha-zero and many more. There are also many minor CEs developed by individuals which they use for creating their CBs. Developing a CE is not a new concept and throughout the years, many techniques have been developed that allow them to reach greater playing heights.

One of the most effective ways to improve the performance of a chess engine is the use of Multi-Threading (MT) to execute its search in parallel. Two of the most common parallelisation techniques are Young Brothers Wait Concept (YBWC) and Lazy Symmetric Multiprocessing (Lazy SMP), but not much work has been publicised about how the performance of the two compare. This thesis aims to create a CE that can operate using both strategies to quantify the strengths and weaknesses of both against each other.

1.2 Problem

This research project addresses the lack of scientific papers comparing YBWC and Lazy SMP head to head, even though many researchers have implemented the algorithms in their CE (with some discussed in chapter 2.

To mitigate the problem, a comparative study of the two parallelisation strategies will be conducted, and a scientific paper will be written. An investigation will be carried out to assess how well the two algorithms perform given the parameters of search speed, playing strength, and Scal. By conducting experiments, data will be collected that can be used to compare the two algorithms in these aspects. Through the comparison of the performances

of the two algorithms, it is anticipated that a conclusion can be drawn regarding whether one algorithm would be a better fit under certain circumstances.

1.2.1 Original problem and definition

The research aims to examine the operational properties of the two algorithms YBWC and Lazy SMP. The research will focus on how the two algorithms affect the CE's Speedup (Sped) and Elo Rating (ER). The research also aims to examine if one algorithm scales better with multiple cores than the other.

1.2.2 Scientific and engineering issues

The main engineering experiences drawn from this project will come from implementing the two parallel search algorithms. Such experiences will include achieving efficient thread management, good load balancing, and managing and reducing synchronisation overhead.

More broadly for the developed program, further experiences will also come from managing the implementation complexity and through ensuring the robustness of the CE with thorough error handling and testing.

Scientifically, the project aims to deepen the knowledge of parallel algorithms. This knowledge will be gained by getting insight into how the two parallelisation strategies explore the sizeable search space required by chess engines to operate, and into how the two strategies scale with increasing core counts.

1.3 Purpose

This project aims to compare Lazy SMP and YBWC to determine which one is better in various aspects. It will serve as a comparative study between the two algorithms. Due to the limited information available on their comparison, this research seeks to fill that gap. The hope is that the findings will be valuable to chess engine developers by providing empirical data on how well the two techniques perform under identical conditions and that it will contribute to ongoing investigations into efficient parallel computing strategies. Additionally, another purpose is to fulfil the goals of this course for the authors to graduate.

1.4 Goals

The main goal of this project is to make a comparative study of Lazy SMP and YBWC, this goal can then further be split into smaller sub-goals. Here is a list of sub-goals that should be completed:

1. Select an open-source CE to work with
2. Implement Lazy SMP and YBWC
3. Perform tests to measure the following parameters:
   (a) Sped
   (b) Scal
   (c) ER
4. Analyse the measured data and draw conclusions from the analysis
5. Verify that the research meets the course objectives

1.5 Research Methodology

For this project, quantitative research has been selected as the methodology. The philosophical assumption guiding this research is positivism, based on the belief that an objective description of the performance of the two parallel search techniques can be achieved through the creation and testing of hypotheses.

To accomplish this, the experimental research method will be utilised, as it allows for a fixed number of variables to be examined to see how changes affect outcomes. In this case, the main variables will be the parallel search technique employed and the number of cores allocated to the engine.

Given the positivistic philosophical assumption and the experimental research approach, a deductive research strategy is the most logical choice, as it involves confirming or refuting hypotheses and using the results to establish causality between the variables and their effects. The primary causality to be investigated is whether one parallel search algorithm offers a performance and/or Scal advantage over the other.

There are however also some qualitative aspects to the project. To begin, YBWC and Lazy SMP are not the only parallel search algorithms used by chess engines, but they were deemed to be the two most popular alternatives as most multi-threaded engines found during preliminary research used one of the two techniques.

Another qualitative aspect is the literature study portion of the project (as can be seen in chapter 3). Here, sources were chosen based on their relevancy to the parallelisation techniques and their findings.

Finally, the open-source chess engine that will be modified for this project was selected using qualitative requirements (which are listed in step 5 of section 4.1).

1.5.1 Philosophical Assumption

It is believed that the choice of positivism as the philosophical assumption suits the research well. This is because it is based on testing hypotheses to either strengthen or weaken them through experimentation. It is worth mentioning that positivism has been recognised as a suitable method for testing the performances of information and communication technologies [9].

Positivism is characterised by the assumption that reality is objective and does not depend on either the observer or the instrument. This assumption is appropriate for research concerning information and communication technologies, where the data is considered valid in itself. In this context, the data is not merely an opinion that can be easily changed, but rather an observation of the truth.

Although realism was considered, it was concluded that conducting tests for the research would be essential, leading to the decision that positivism would be a more suitable approach. Realism mainly centres on understanding the data itself, including the conclusions that can be drawn from it and the insights that can be gained [9]. In contrast, this research focuses more on measuring performances with various variables.

1.5.2 Research Method

The experimental research method was chosen as it was believed to be best suited to the research. It was considered that the adjustment of variables would be central to the planned test experiments with Lazy SMP and YBWC. The most important variables would be the number of cores available and the

algorithm employed.

Experimental studies focus on cause and effect; if one variable is changed while keeping the rest of the system intact, the effect can be observed. This method sets parameters for the investigation and examines how those changes affect the system [9].

The descriptive research method was rejected because the focus is not on asking, ”Why does the CE perform better when a variable is changed?” Instead, the interest lies in verifying or falsifying hypotheses, such as, ”Lazy SMP is more scalable than YBWC.” Investigating a deeper understanding of why the CE performs better when a variable is altered would be a subsequent step in the project.

The descriptive research method investigates what conclusions can be drawn from a set of data and what can be said about it. This method would be more suitable for research projects where more information about the actual data used would be of interest [9].

The applied research method was also rejected because the research does not aim to solve known technological problems for a CE, nor is it a continuation of someone’s research that requires further examination of specific details.

The applied research method is appropriate when a researcher wishes to examine existing research and investigate certain details further [9].

1.5.3 Research Approach

A deductive research approach is deemed most suitable because most of the research will focus on proving or disproving hypotheses. Several hypotheses have been formulated that will be tested through quantitative methods. The deductive research approach primarily involves deducing what is true or false through tests with quantitative data [9].

The abductive research approach was rejected because the research does not focus on understanding the various reasons why the CE performs better or worse when different variables are adjusted. Instead, the interest lies in determining whether it performs better or worse compared to previous attempts. The abductive research approach combines elements of both inductive and deductive research, merging the verification or falsification of theories or hypotheses with the generation of new hypotheses while testing their validity [9].

1.6 Delimitation

1. The actual CE will be designed, while the interface for it will use an existing solution. The CE will adhere to the UCI protocol (UCI) to ensure enough compatibility with front-ends to perform the necessary experiments.

2. This project is not concerned with achieving the utmost performance by deep-diving into every operational aspect of the chess engine and will as such not strive for perfection in every aspect. Decent performance using more basic techniques will be enough to compare the two parallel search algorithms as long as they are given the same prerequisites.

3. The use of different operating systems or hardware configurations will not be considered as a performance aspect in this research.

1.7 Structure of the thesis

Chapter 2 introduces the reader to the basics of chess, and to how a CE is structured.

Chapter 3 is the backbone of this scientific paper where other researchers' work in similar areas are examined and evaluated to give inspiration and improve the research conducted in this research project.

Chapter 4 presents the chosen method for conducting research in this project and discusses the validity of the method along with the reliability of the data measured.

Chapter 5 discusses the work that was performed, the experiments that were implemented in practice, and what equipment was used.

Chapter 6 presents the results from the experiments and analyses the data collected from them.

In chapter 7 the results from the previous chapter are discussed.

Chapter 8 will contain a section for conclusions of the discussion in chapter 7 and the results from chapter 6, a section for limitations or shortcomings in this project, and a section that contains a discussion of what could have been improved in this research for future researchers to expand on the research written in this report.

Chapter 2

Background

Section 2.1 will introduce the reader to important techniques used by a chess engine. Then, section 2.2 will build upon this by introducing how a chess engine with these features can be parallelised with a focus on the two parallelisation techniques the report is concerned about comparing.

2.1 Features of a Chess Engine

A fully-featured chess engine can be quite complex with many moving parts. To make it more digestible, the main features are categorised into four main parts:

- Board Representation
- Move Generation
- Move Search
- Evaluation

Section 2.1.1 will describe how a chessboard can be represented in a chess engine, section 2.1.2 will describe how moves are generated, section 2.1.3 will describe how an engine searches for the optimal move to play, and section 2.1.4 will describe how moves are scored to estimate how good they are.

2.1.1 Board Representation

2.1.1.1 Bitboards

The chessboards and pieces have been chosen to be represented using bitboards. A standard chessboard consists of 64 (8×8) squares, meaning that each square's occupancy can be represented by a 64-bit unsigned integer (a bitboard). It is necessary to differentiate between the different types of pieces and their respective players, meaning that at least 12 (6 pieces × 2 colours) bitboards are required. Three additional bitboards (representing all white pieces, all black pieces, and all pieces) can be derived from the mandatory 12 for convenience by using logical addition (bitwise OR) on the relevant bitboards [10, 11].

Table 2.1 provides a visual representation of a bitboard.

56	57	58	59	60	61	62	63
48	49	50	51	52	53	54	55
40	41	42	43	44	45	46	47
32	33	34	35	36	37	38	39
24	25	26	27	28	29	30	31
16	17	18	19	20	21	22	23
08	09	10	11	12	13	14	15
00	01	02	03	04	05	06	07

(a) Bit index

1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1

(b) All pieces

Table 2.1: (a) shows how bitboards are indexed, with 00 being the least significant bit and 63 being the most significant bit. (b) shows the bitboard representing all pieces in their initial positions.

Bitboards are convenient to use for a few reasons. The first reason is derived from the fact that most processors today are 64-bit, which means that it is very efficient to perform bitwise operations on them since the bitboards themselves are also 64-bits large [10, 11].

The second reason is that bitwise operations make it easy to isolate certain pieces without having to manually go through all the pieces and move the pieces, which will be useful when generating moves. Additionally, the small size of bitboards requires less memory to be allocated for them.

2.1.1.2 Alternatives

Since there is a Bitboards (Bb) for each type of piece it is considered a piece-centric solution to board representation. The alternative to this is board-centric approaches like two-dimensional arrays or Mailboxes. A solution using two-dimensional arrays would simply be eight by eight large integer array, where the integer encodes any piece sitting the square it represents. The problem with 2D arrays is that it takes multiple instructions to calculate the memory address by this type of indexing, which is an unnecessary performance penalty when generating moves [10].

Mailboxes is a common name for board-centric solutions that use a one-dimensional array or a list, the simplest of which works exactly like a 2D array except it is indexed $0 - 63$. Both this solution and 2D arrays are sensitive to wraps when generating moves, which means that an illegal move which would go out of bounds ends up on the opposite side of the board. These solutions must manually check every generated move to see if it contains an erroneous wrap. This problem also exists with bitboards, but can easily be solved, as will be discussed in section 2.1.2. A mailbox solution to this would be to use a larger 120 (10×12) large array that has positions that would be outside the squares of a regular chessboard (two to the left and right, and one above and below). Moves that land in these inaccessible squares can then be ignored after reading the contents of that array position [10].

Another mailbox solution is $0x88$ boards. These are a 128-long array where valid positions are represented by:

$$index \mod 16 < 8$$

and invalid positions are represented by:

$$index \mod 16 \geq 8$$

To simplify, the index can be written in hexadecimal and then the indices whose least significant nibble is less than eight are valid. Table 2.2 shows how such a board is indexed. They are called $0x88$ boards because you can check if a position is valid by using bitwise AND on the position index and the hexadecimal 0x88, and the result will be zero if valid. $0x88$ ends up being faster than the other mailbox techniques because you can determine if a piece has landed on an invalid square in a single bitwise operation. Since half of the

70	71	72	73	74	75	76	77	78	79	7a	7b	7c	7d	7e	7f
60	61	62	63	64	65	66	67	68	69	6a	6b	6c	6d	6e	6f
50	51	52	53	54	55	56	57	58	59	5a	5b	5c	5d	5e	5f
40	41	42	43	44	45	46	47	48	49	4a	4b	4c	4d	4e	4f
30	31	32	33	34	35	36	37	38	39	3a	3b	3c	3d	3e	3f
20	21	22	23	24	25	26	27	28	29	2a	2b	2c	2d	2e	2f
10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f
00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f

Table 2.2: A visual representation of an 0x88 table. The numbers are placed on the chessboard squares they represent. The numbers are the indices of a 128-long array written in hexadecimal. Positions left of the vertical line are valid, and those to the right are invalid.

$0x88$ board is not used, it could be used to store additional information about squares they shadow (ie, additional information about $0x00$ can be stored in $0x08$, information about $0x32$ can be stored in $0x3a$, etc.) [10].

Mailbox solutions typically store the type of piece as a number in the mailbox array. This means that you need to go through the array and compare the contents before you can start generating moves. To mitigate this, you can keep a list each for all white and black pieces where each entry describes the type of piece and its location on the board [10].

The main reasons to use bitboards are that they are more performant than mailboxes without needing additional speedup techniques, they seem simpler to generate moves than mailboxes, and most modern engines found during the pre-study used them.

2.1.2 Move Generation

To generate moves with bitboards, you need to make a distinction between sliding pieces and non-sliding pieces. Non-sliding pieces are those that can move to a fixed number of possible positions; pawns, knights, and kings. Sliding pieces are those that can go indefinitely in a direction until reaching an obstacle; rooks, bishops, and queens. The fact that non-sliding pieces can not ”jump” past other pieces and keep going makes their move generation more complex, and will as such be described after non-sliding pieces.

2.1.2.1 Non-sliding pieces

For non-sliding pieces (pawns, knights, and kings) you can use special bitboards (sometimes called lookup tables) and bit-shifting to create new positions. Lookup tables are 64-bit unsigned integers that are used to isolate or remove the pieces on a rank or file from a bitboard by using bitwise AND on them [11].

Examples of lookup tables can be seen in table 2.3.

0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0

(a) Isolation

1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1

(b) Removal

Table 2.3: (a) shows a lookup table that is used to keep only the pieces on the C-file. (b) shows a lookup table that keeps all pieces except those on the third rank.

First, two isolation lookup tables create a temporary bitboard containing only a single piece. Since the bitboard it came from is known, what type of piece it is and consequently how that piece moves is also known. Because the piece is isolated, it can be moved by simply shifting the entire bitboard by an amount corresponding to where it can go [11]. As an example, a piece will go one step forward with a ”«8” shift, a step right with ”»1”, and two steps backwards and one to the left with ”»17”.

To avoid wrapping (moving horizontally close to the edge and ending up on the other side), removal lookup tables will be used for moves that could lead to a wrap [11]. An example is the knight’s movement, which goes in an ”L” pattern. Two of the moves go right two steps and vertically one step, so when generating either of these moves removal lookup tables are always used for files G and H on the bitboard before shifting. This means that the bitboard will be all zeros if the knight was on the G or H files when generating these moves, but work normally when generating these moves from any other file. Another example is the king who can move one square in any direction. If it is standing on the A file, three moves would result in a wrap (up-left, left, and down-left), so when generating

these moves a removal lookup table is always used for file A before shifting. Removal lookup tables are not needed for ranks, since when a bitboard is shifted so a piece goes beyond the first or eight ranks it will not wrap to the opposite rank, instead resulting in a bitboard of all zeros which is desired.

All the positions that the pieces could theoretically move to are given by this process, but in reality, fewer positions will almost always be available. This is because other pieces on these squares have not been taken into account so far. The same square as another piece of the same colour cannot be occupied by a piece, so these are removed by performing a bitwise AND on the generated positions and the inverse of all pieces of the same colour (own pieces are represented as zero, while everything else is represented as one).

2.1.2.2 Sliding pieces

The easiest way to handle sliding pieces (rook, bishops, and queens) is to do it the same way as non-sliding pieces and calculate one square at a time in each direction until the sliding piece can't go any further [12].

This is quite inefficient though, as more possible moves can be generated from sliding pieces than non-sliding pieces. A pawn can move at most to four different positions (counting diagonal attacks and en passant, which are very situational), knights can at most move to nine different positions (counting castling), and kings can at most move to ten different positions (counting long and short castling) while a bishop could theoretically move up to thirteen positions, a rook up to fourteen positions, and a queen up to twenty-seven positions. In addition, when generating moves like this with sliding pieces you must check if the piece can continue moving after every square while all non-sliding moves are generated at the same time with illegal moves being quickly removed after.

Instead of the simple solution described above, Magic Bitboards will be used. Rooks and bishops are allowed to move indefinitely in the orthogonal and diagonal directions respectively, until an edge of the chessboard or a blocking piece is reached. Since only pieces that are in the way can stop these pieces, 64 bitboards (one for each square) are created for rooks and bishops, isolating the squares of the pieces' attack patterns that can contain blocking pieces. A bitboard containing only the pieces that block the rook or bishop is obtained by performing a bitwise AND on the bitboard of all pieces. By removing the pieces that do not affect move generation, a

reasonable precomputation of the bitboard containing possible moves for each combination of blocking pieces for every square can be achieved.

Magic Bitboards is a technique in which these solution bitboards are stored in a hashtable, using the bitboards that contain only the blocking pieces as indices to locate them [12]. Perfect hashing (hashing without collisions) is employed by Magic Bitboards, using a (magic) number that is ANDed with the bitboard of blocking pieces to create a unique index. This process is only necessary for rooks and bishops, as queens move like a combined rook and bishop. When looking up moves for queens, the process is thus performed twice: once to find rook moves from their position and once to find bishop moves.

An alternative to the use of Magic Bitboards is the performance of a PEXT lookup. This method is similar to Magic Bitboards in that a hash table is utilised to find all possible moves, but the Parallel bits EXTract CPU instruction is employed to generate the index. This approach is slightly faster than Magic Bitboards [12], but it was decided not to use it since it is not natively supported by all of the authors' computers.

2.1.2.3 Legality and Special Moves

The previous two subsections have outlined how the attack and movements are created in a pseudo-legal manner. This means that the moves and attacks are directed towards places that the pieces can reach, but do not consider whether the movement places the king in check. Any move that puts the king in check or maintains the king in check is not permitted, so any such moves must be removed.

The simplest solution for this is to retain all pseudo-legal moves and to detect and remove or invalidate them when generating moves for the opponent in the next ply. This approach has the advantage of not requiring the legality of every move to be calculated, as many will be pruned away by the search function, however, legality is still established for all un-pruned moves [13].

Pawns are special because they move and attack using different patterns. Therefore, any movement that lands on an occupied square regardless of colour must be removed and only attacks that land on squares occupied by the opposite colour are kept.

They can additionally move two squares forward on their first turn, so this additional move must also generated for white pawns on the second rank and

for black pawns on the seventh rank.

Finally, they have a special attack – the ”en passant” – which allows them to take a pawn by going diagonally behind it if the enemy pawn moved two squares during the previous turn.

The easiest way to implement this is to have an en passant bitboard that marks the square that is passed when a pawn moves two squares and to include this bitboard along the opponents pieces when checking if pawn attacks are valid.

When a pawn reaches the final rank of the board (eight for white, one for black) it will be given a promotion. This can be solved with bitboards by replacing the single move that placed the pawn on the final rank with one of four new positions – one with a knight on that square, one with a rook, one with a bishop, and one with a queen.

Another special move is castling – where a king is moved two squares towards one of its rooks and the rook is moved to the square on the other side of the king. It can only be performed if the king and rook have not moved previously, there are no pieces between them, and none of the king, rook, or the squares between them are under attack. It is considered a non-sliding move despite involving a rook because there are only two outcomes (long castling with the A-file rook and short castling with the H-file rook). For pawns, it is only necessary to check their positions to see if they are allowed to move two squares because they cannot move backwards. Kings and rooks, however, can do this so a flag must be set that tells if they have moved yet or not. Isolating bitboards can be used to see if there are any pieces between the king and the rooks and stop when the bitboard of those isolated squares is not zero.

2.1.3 Move Search

When generating all moves from a position, the engine will go to the generated positions and start generating moves from this position. Doing this creates a tree of positions that will need to be searched for the move with the highest likelihood of leading to a victory. This section will describe methods used when searching through this game tree.

2.1.3.1 Negamax algorithm

Negamax is a simplified implementation of the minimax algorithm. The main idea behind minimax is that the terminal nodes of a game tree

are evaluated and assigned scores. These scores are then propagated up the tree and the branch with the best score will be chosen as the best move. The scores are propagated up the tree by selecting the smallest score (minimising) and choosing the largest score (maximising) every other depth level. Here the white player will choose the largest score and the black player the smallest score. This works because the players want to play the score that is best for them and therefore worst for the opponent.

Minimax requires two almost identical separate functions for minimising and maximising and can therefore introduce bugs from accidentally using the wrong function or from needing to know if the player is black or white.

Negamax simplifies this by only using one function and always maximising the engine regardless of colour.

It works because a maximised score for black will be the same as the maximised score for white if the pieces are in the same positions but with their colours swapped.

This means that the maximised scores from the next depth level can be negated and it will be equivalent to if they were minimised [14].

Figure 2.1 shows a small game tree where black minimises the score and white maximises the score.

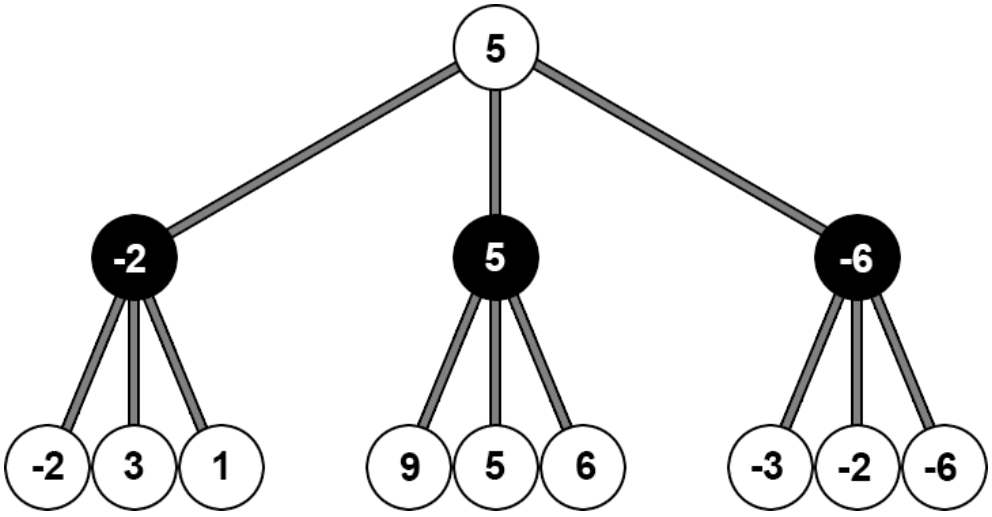


Figure 2.1: Visualisation of how minimax and negamax select the best move, created in Adobe Photoshop.

Pseudo-code describing the negamax logic can be seen in figure 2.2.

```
int negamax(int depth){
    if(depth == 0)
        return evaluate();

    int max_score = INT_MIN;
    generate_moves();
    while(move = get_next_move()){
        make_move(move);
        score = -negamax(depth -1);
        unmake_move(move);
        if(score > max_score)
            max_score = score;
    }
    return max_score;
}
```

Figure 2.2: Negamax Algorithm Pseudo-code

2.1.3.2 Alpha-beta pruning

Alpha-beta pruning is a minimax/negamax improvement that greatly reduces the number of moves needing to be evaluated by removing those that are guaranteed not to be played. It works by keeping track of two values, alpha which stores the minimum score the maximising player can get, and beta which stores the maximum score the minimising player can get. If the maximising player encounters a score greater than the beta it will immediately stop searching since the minimising player will never choose that move. Conversely, if the minimizing player finds a score lower than the alpha, the search will stop because the maximizing player will not select it [15].

Figure 2.3 shows the same tree as in figure 2.1 but now with alpha-beta pruning. The alpha cutoff when evaluating the second branch is -2 and there is no cutoff in this branch because no child node has a lower value than this. The alpha cutoff is then updated to 5 when the third branch is evaluated, and because the first node is less than this the remaining child nodes are cut off.

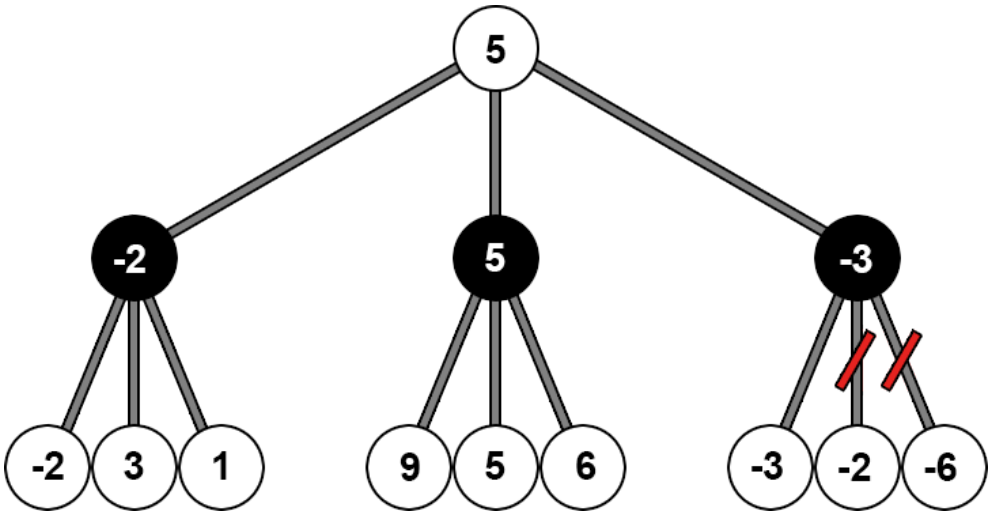


Figure 2.3: Visualisation of how alpha-beta pruning removes nodes from consideration in a minimax/negamax tree, created in Adobe Photoshop.

Alpha-beta pruning works best when the best moves are searched first. Michael Levin showed that the number of terminal nodes (T) generated to a certain depth (n) can be estimated with a fixed number of branches (b) from each node to be $T = b^n$ and that it can be reduced with alpha-beta pruning to at best $T = b^{\lceil n/2 \rceil} + b^{\lfloor n/2 \rfloor} - 1$ with perfect ordering [15].

Table 2.4 uses Levin's theorem to show how much the number of terminal nodes required to evaluate can be reduced assuming 30 branches from each node.

2.1.3.3 Transposition tables

When generating every possible move the same positions will inevitably repeat sooner or later. It would seem wasteful to recalculate how good the same position is repeatedly, so evaluated positions should be stored to quickly look up how good a position is when reencountered. The solution is to use a transpositions table – a hashtable that uses the positions as the key and stores the position score, the depth it was searched at, and the best following move [16].

With the score, generating and evaluating the children of the position can be skipped.

With the depth, it can be determined if a reevaluation of the position is necessary due to the presence of a sub-tree that is too small.

With the best move, the children can be ordered when reevaluating the

depth (n)	b^n	$b^{\lceil n/2 \rceil} + b^{\lfloor n/2 \rfloor} - 1$
0	1	1
1	30	30
2	900	59
3	27000	929
4	810000	1799
5	24300000	27899
6	729000000	53999
7	21870000000	836999
8	656100000000	1619999
9	19683000000000	25109999
10	590490000000000	48599999

Table 2.4: Number of terminal nodes to evaluate with and without alpha-beta pruning with perfect ordering and 30 branches per node (b) up to a depth of 10.

position from having insufficient depth, ensuring that the previously best move is evaluated first, which often has a good chance of being the best move again, thereby pruning the rest of the sub-tree.

2.1.4 Evaluation

The Evaluation Function (Eval) takes a Board State (BS) and evaluates who is leading, white or black. The Eval for the project needs to be adequate and efficient because the main purpose of this project is to measure parallelisation. To make the evaluation function adequate some key aspects must be considered [17, 18, 19].

2.1.4.1 Material

Assigning values to different chess pieces will help give the computer a sense of what chess pieces are good to trade and what they are worth. This is an essential aspect of chess because it can then embody that all pieces' values are relative. During the game, the positioning or amount of pieces on the board may change and that can affect the values of the individual chess pieces [20].

This guideline can be implemented by first giving the individual pieces starting values which may change during the game. The Piece Square Value Table (PSVT) will contribute to assigning values to the pieces and a float value will be created to indicate which game phase the game is in. There are Three

game phases Opening, Middlegame and Endgame. During the course of the game, the game phase will move from Opening to Endgame from left to right and this process can not be reversed.

2.1.4.2 Pawn Structure and centre control

In the game of chess, it is beneficial to control the centre of the chessboard with pawns. It is also beneficial to have a couple or less connected pawn structure. The chess pieces should not be spread out on the board without purpose [20].

How then to implement this guideline is a different matter. A program will be built to check that all the pawns are connected and the use of PSVT will be used to assignee good, neutral and bad squares for the pawns [21].

2.1.4.3 King safety

To evaluate king safety is essential for a good Eval but not simple to implement. One reason why it is difficult is that, through a chess game, it is hard to say that now is king is safe, king safety is rather a more relative aspect [20].

Some aspects one could use to strengthen the argument for king safety are:

1. Pawn Shield
    (a) Are there pawns in front of the king
2. Pawn Storm
    (a) Are there enemy pawns attacking the king's pawns
3. Attack Units
    (a) How many enemy pieces are attacking the king or the king's defending pieces?

2.1.4.4 Piece-Square Tables

PSVT is a table that contains values for a given square for a given piece. It is a way of evaluating if a piece sits on a good, neutral or bad square. It is also a way of telling the CE if you move that piece to that square then you have an advantage over your opponent. This will serve as a technology to accelerate the development of minor pieces which is essential for playing good chess [21].

The PSVT can be implemented as a byte array of size 64 containing values for given indexes.

2.1.4.5 Machine learning

Building an Eval based on Convolutional Neural Networks (CNN) technology was considered but ultimately dismissed because it would be too complex to implement given the time frame. A Static Evaluation Function (SEF) was chosen because the implementation details seemed simpler. If an Eval were to be made using CNN technology, a couple of considerations would need to be addressed. For starters, uncertainty exists regarding the source of the training data for the Eval; questions arise about how the training data would be transformed into a suitable format for the Eval, how much time would be required to train the Eval, how the Eval would be tested, and how the impact of the Eval on the overall performance of the CE could be measured [22, 23, 24, 25].

The CE Stockfish's ER went up when it switched from SEF to a form of neural networks called "Efficiently Updatable Neural Network" [13]. This suggests that the strongest evaluation functions today are based on neural networks.

2.2 Parallelisation

The core of this project handles parallelisation. Parallelisation is when a computer executes multiple tasks in parallel on separate processors that would otherwise be executed sequentially by a single processor. The ideal scenario for the CE would be Single Instruction Multiple Data (SIMD) scenarios where the data being processed does not have any dependencies of each other. Then multiple threads could be executed simultaneously which would increase the efficiency of the CE [26].

The two parallel search techniques handled in this study are the YBWC and Lazy SMP. The tree is being generated and evaluated as it is searched through, and the challenge comes from how multiple threads should work together to improve the performance without interfering with each other. Section 2.2.1 describes how YBWC works, and section 2.2.2 how Lazy SMP works.

2.2.1 Young Brothers Wait Concept

The main idea behind YBWC comes directly from its name – other threads must not handle the younger brothers of the nodes in the search tree until their eldest brother has been evaluated. This means that when the search is started, the main thread will generate all moves from the root position, make the first

of those moves, generate all moves for it, make the first of those moves, and repeat until maximum depth has been reached. The main thread is now at the leftmost terminal node, evaluates it, and unmakes that move. The first eldest brother has now been fully handled, so the other threads can handle the rest of the child nodes from that position. When all the children of this node have been evaluated, the best score from them will be assigned to this node and it will be finished, meaning that the younger brothers of it can be handled. This process is then repeated until the entire tree has been evaluated [27].

There are however more things of note than this. To begin, a thread gets to work on a sub-tree by asking a different thread if it has work it can give, and the asked thread either responds that it has nothing to share or with the lowest (closest to max depth) sub-tree it has available. The use of alpha-beta pruning would also mean that the amount of work a thread must do on a sub-tree is reduced or cut off entirely by the master thread that gave it that work, so the threads must be able to communicate about this for time not to be wasted on useless computations [27, 28].

2.2.2 Lazy SMP

Lazy SMP is a relatively new parallelisation algorithm first described in 2013 by Daniel Homan on the Computer Chess Club Forum, inspired by a previous forum discussion about ”lazy” ways to implement Symmetric Multiprocessing (SMP), and it is stated that it is quite simple to implement [29]. This algorithm is set apart from other parallelisation algorithms because the communication between the threads is minimal and reduced to only the transposition table (described in section 2.1.3.3).

The idea behind this algorithm is to simply start the different threads simultaneously, offset from each other with a different root move ordering and/or different depths. The different threads will update the transposition table as they conduct their search, so when a thread encounters a position that has already been evaluated by another thread it will simply retrieve the score from the transposition table. The threads would likely try to evaluate the same positions at the same time before they were finished if all threads start with the same move ordering, so their initial positions are offset to reduce this risk. For the use of a shared transposition table to be efficient, it needs to be made lock-less to avoid the synchronisation overhead [26].

A problem that occurs with this parallelisation strategy is for the threads to know if they are evaluating a cutoff node or which of its children are cutoff nodes. Different kinds of technologies can be implemented to mitigate the reoccurring problem but it remains a flaw in the algorithm. [26]

Chapter 3

Literature Study

This chapter will describe some important reports that were read in preparation for this project. Section 3.1 will describe the results of some reports that studied YBWC, section 3.2 will detail Lazy SMP reports, and section 3.3 will depict papers on ER.

3.1 Young Brothers Wait Concept

3.1.1 Parallel Chess Searching and Bitboards

In this science paper "Parallel Chess Searching and Bitboards", the author David Ravn Rasmussen was experimenting with a chess engine to get a better understanding of how well YBWC works with multiple processors. The author found that YBWC is not very scalable, the Sped gained when adding many processors was not ideal.

One thing that the author mentions is that the memory bandwidth in the experiments was not ideal. One could argue if the memory bandwidth was more ideal then the amount of data transferred during operation would have had a lesser impact on the measured Sped.

The results from their experiments were:

"Maximum speedup was 9.2 on 22 processors in the main experiment, 8.3 on 16 processors in the preliminary experiment and 9.4 on 12 processors in the truly dedicated 12 processors experiment".

The author argues that the main reason why YBWC is not very scalable is because there are too many nodes containing a cutoff node that are searched in

parallel. The author also states that the size of the search tree grows linearly, with 10 processors the tree size doubles as compared to 1 and triple with 20 processors.

The author further notes that the Sped gained when adding more processors is sub-linear. That means that the rate at which the tree grows is greater than the gained Sped. However, the least amount of time used is at 17 processors. The author then thought that the reason for this is that there are too many nodes containing a cutoff node that are searched in parallel [30].

3.1.2 Two-level Task Scheduling for Parallel Game Tree Search Based on Necessity

In their conference paper "Two-level Task Scheduling for Parallel Game Tree Search Based on Necessity," Akira Ura et al. aimed to improve speedup Sped by minimising unnecessary task execution. They achieved this by classifying tasks into two priority classes: a lower class for tasks likely to be pruned by alpha-beta pruning, and a higher class for tasks less likely to be pruned.

While this approach resulted in improved speedup, the authors suggest that expanding the classification to three levels could yield even better results. However, they also caution that introducing a third class might increase inter-process communication, potentially diminishing the speedup gains [31].

3.1.3 GridChess: Combining Optimistic Pondering with the Young Brothers Wait Concept

In his paper "GridChess: Combining Optimistic Pondering with the Young Brothers Wait Concept", Kai Himstedt describes his implementation of a distributed YBWC search as well as optimistic pondering and how these affect the engine's Elo rating [32].

The engine is based on Toga by Tomas Gaksch, with the YBWC implementation called Cluster Toga and the whole system using optimistic pondering being GridChess. Among other things, Himstedt measured how Cluster Toga running on different numbers of cores and clusters performed (without optimistic pondering) against Gaksch's Toga running on a single core after 70 games each with 20-minute timers and 5-second increments.

The results the author presented show that Cluster Toga running on a single cluster with a single core won against Gaksch's Toga 48.57% of the time, with Cluster Toga's Elo difference being -10 (95% confidence interval: [-67, +47]).

With two cores on a single cluster, it won 55.00% of the time and has an Elo difference of +36 (interval: [-20, +92]). With four cores on a single cluster, it won 66.43% of the time with the Elo difference of +120 (interval: [+63, +120]).

This trend continues when using multiple clusters up until four clusters with four cores each, but further increasing to eight clusters with four cores each has the same win percentage of 72.86% and Elo difference of +172 (but with a slightly tighter interval: [+119, +232] vs [+115, +238]).

The author admits that the width of the confidence intervals makes it difficult to determine the actual Elo difference between the engines, likely due to the relatively low number of matches played, but points out that Cluster Toga consistently has a positive Elo difference with at least four cores and that this difference increases with more cores.

The author also explains that the number of games played and the number of opponents was limited by time constraints and the amount of access allowed to the computer cluster Cluster Toga ran on. He further explains that Gaksch's Toga was run on a separate computer that although uses a different CPU should have the same single-core performance as the ones Cluster Toga ran on, that the YBWC implementation in Cluster Toga has some slight inherent overheads, and that Gaksch's Toga was compiled with higher optimisation, but does not explicitly state that these are reasons to why Cluster Toga performed worse than Gaksch's Toga when single-threaded.

While this report does not present a time-based Sped from running on multiple cores (or clusters), it does show that the Sped achieved leads to better play. Assuming that the performance from running on multiple clusters is comparable to the performance from running on the same amount of cores locally, it would seem that scales well up to 16 cores with 32 cores resulting in a minimal difference.

3.1.4 A Fully Distributed Chess Program

In their paper "A Fully Distributed Chess Program", Rainer Feldmann et al. discuss how they parallelised a chess engine (ZUGZWANG) with YBWC and alpha-beta improvements [28]. The alpha-beta improvements they implemented include transposition tables (global, local, and distributed), local killer lists, local history tables, and iterative deepening.

In their solution, each processor will divide the work they are given

into subproblems, and the processors not currently working will randomly ask another processor for a subproblem to work on. When a processor receives a subproblem, it in turn can also be subdivided and sent to processors asking for work. When a subproblem has been solved, the solution is sent back to the processor that initially gave away the subproblem.

The processors communicate with each other by sending a few kinds of messages to update the alpha-beta search windows. When the search window for a node changes, it will inform the processors with subproblems for this node of this with the "NEWBOUND" message. If the change of the search window leads to nodes being cut off immediately, a "CUTOFF" message will be sent to the affected processors instead so they stop immediately.

The authors use the 24 positions from the Bratko-Kompec test to measure the performance of ZUGZWANG [33]. Feldmann et al. present that the Sped of having 64 processors search to a depth of seven plies (a ply is one side's turn) from these positions is on average 34.77. The authors explain that this average is made higher by one of the positions that on its own had a Sped of 63.57. To account for this outlier position, they also measured this position to a depth of six plies which resulted in a Sped of 41.41 with the new average for all being 27.99.

It should be noted that Feldmann et al. did more measurements than just those discussed here, but a copy of this source that included the entire results chapter could not be found. Nonetheless, the results mentioned in the abstract and mainly discussed in the conclusion chapter were included in their entirety.

3.1.5 Summary

List of YBWC sources		
Author	**Pros**	**Cons**
David Rasmussen	YBWC Sped measurements on different cores, give code on the chess engine	Not ideal experiments and Could have tested other parallelisation strategies
Akira Ura	Implementation of YBWC was unique and explains how the order of executed tasks can affect results due to pruning	Done further testing with 3 priority classes and comparative testing with other strategies based on YBWC
Himstedt	Good Elo and win rate increase from multiple cores	Sped not measured
Feldmann et al.	Good Sped for a high number of cores	A low number of tested positions makes the average results vulnerable to outliers

3.2 Lazy SMP

3.2.1 Piece By Piece: Building a Strong Chess Engine

In their paper "Piece By Piece: Building a Strong Chess Engine", Sander Vrzina studied how different techniques can be used to develop a competent chess engine, ending up with an engine playing at around 2400 Elo [13].

One of the techniques implemented was Lazy SMP for parallel search. Vrzina found that Lazy SMP was simple to implement, and measured a Sped of 40% when running on four cores. Unfortunately, Vrzina experienced some instability in Lazy SMP that led to the engine making unexpected blunders which they attribute to an unidentified error in either the alpha-beta pruning logic or in the transposition table.

Vrzina had initially tried implementing a different parallel search technique but was not satisfied with the Sped and so chose to use Lazy SMP. They expect an implementation without the error in their engine would eliminate the blunder moves and that larger Speds are possible. The author also ranks which techniques they thought would be most important to focus on if they were to continue work on the engine, and listed Lazy SMP as the top priority due to the potential Sped it offers.

3.2.2 A Complete Chess Engine Parallelized Using Lazy SMP

In his master thesis "A Complete Chess Engine Parallelized Using Lazy SMP", Emil Fredrik Østensen describes the techniques he used when creating a chess engine, focusing on using Lazy SMP to parallelise it [26]. Østensen describes how he avoids synchronisation overhead by using a lockless hashtable as the transposition table by including a checksum calculated by inserting thread in the entries.

Østensen managed to create an engine with an estimated Elo of 2238 and measured a Sped at depth 9 of 1.14 with two threads and 2.01 with four threads, the average number of nodes searched for the same number of positions being 78% more with two threads and 101% more with four threads, and the average number of nodes searched per second being 202% as many with two threads and 405% as many with four threads.

Østensen believes that the best way to improve upon Lazy SMP would be to reduce its randomness, and suggests two ways to potentially accomplish this. The first way would be to occasionally check the transposition table for moves in its history to avoid redundant searches. The second way would be to make the algorithm determine if it is currently in a CUT node or an ALL node before deciding the child node to explore first.

3.2.3 Evaluating Heuristic and Algorithmic Improvements for Alpha-Beta Search in a Chess Engine

In his bachelor's thesis "Evaluating Heuristic and Algorithmic Improvements for Alpha-Beta Search in a Chess Engine", Henrik Brange describes how he created a chess engine utilising MVV-LVA move ordering, a transposition table, iterative deepening, and Lazy SMP parallel search with measurements of how these techniques affect the execution time of the engine [34].

The Lazy SMP implementation was tested on four cores against a single-threaded version of the same engine, and when searching to depth 6 for 600 board positions the average execution time of Lazy SMP was 33.4% lower than the single-threaded version (a Sped of 1.51). Brange also measured the memory usage during iterative deepening and found that Lazy SMP required slightly over 20% more memory for the first two iterations, but only about 1% more for the third iteration.

Furthermore, the number of hits to the transposition table was measured for each depth, with Lazy SMP being at worst having 2.8 as many hits and at best 25 times as many. As the engine was made in Java, the author also recorded how often and how much time was spent by the JVM garbage collector and found that the Lazy SMP version spent 24.5% of its execution time on average on garbage collection as opposed to just 7.8% by the single-threaded equivalent.

The author lists three factors that could have affected the paper's validity. The first one is that the board positions chosen, although taken from real games, may not be representative of real play since most of them depict middle-game, but defends this choice because the middle game is the most complex and where the search tree's branching factor is the greatest.

The second is the dynamic optimisation made by the JVM which can make the performance (especially in early execution) vary, but this was counteracted

by running the search 60 times without the transposition table before starting recording time.

The final factor is the fact that all implemented techniques are connected and affect each other's execution, which means that implementing different techniques can potentially have a great impact on the efficiency of others.

Finally, the author concludes that care should be taken when implementing Lazy SMP in Java to minimise the garbage created to reduce the amount of time taken up by garbage collection.

3.2.4 Branch and Bound Algorithm for Parallel Many-Core Architecture

In their paper "Branch and Bound Algorithm for Parallel Many-Core Architecture", Kazuki Hazama and Hiroyuki Ebara discuss their solution to the travelling salesman problem using Lazy SMP [35].

The authors developed a Lazy SMP implementation in C++ using OpenMP and conducted experiments on five sets of 700 to 2000 cities on 8, 16, 32, 64, and 128 cores. The results show that after five runs the average time to search for the shortest path decreased with each increase in core count until 32 or 64 cores depending on the set of cities.

The authors explain that a reason for this is that the shared hash table they used in the solution only allows one thread to write at a time, which leads to the overhead from waiting for write permissions eventually becoming larger than the time saved from the increased core count. The largest average time decrease they found was for the "pr1002" data set when going from eight to sixteen cores, where eight cores on average finished in 4735.27 seconds and sixteen cores in 68.46 seconds – a Sped of 69.17.

It should be noted that the hash tables used as transposition tables in chess are typically lockless, allowing multiple simultaneous writes, as was previously discussed by Østensen (section 3.2.2).

3.2.5 Summary

List of Lazy SMP sources		
Author	**Pros**	**Cons**
Vrzina	Easy to implement, decent Sped	Blunders from instability, dependent on transposition table implementation
Østensen	Good Sped, minimal communication	synchronisation overhead Large search overhead
Brange	Good Sped, high transposition table hit rate	Higher memory usage, much time spent on garbage collection
Hazama and Ebara	Scales quite high	Overhead from exclusive writes to hash table

3.3 Elo

3.3.1 Chess Results Analysis Using Elo Measure with Machine Learning

In this study, Fadi Thabtah et al. tried to examine how good Elo Rating is for measuring a player's strength. They begin by describing Elo as a measured player strength based on their past performance, if you win a lot of matches

then your Elo increases.

The researchers found that Elo Ratings are surprisingly accurate for estimating how likely it is that a player would win or lose, a player with a higher Elo rating is more likely to win against a player with a lesser. They found that the Elo Rating is not very accurate for measuring a player's strength but rather which player is more likely to win [36].

3.3.2 Chess Game Result Prediction System

In this science paper, Zheyuan Fan et al. try to estimate how good the Elo Ratings of chess players are at predicting the outcomes of matches. The prediction is more accurate if the Elo Rating differs more from player to player, if the difference is small then the outcome will be more unpredictable.

Though the source has not compared its data to other data from other sources to check the accuracy of the results, considering empirical evidence the data extracted in the source should be valid. Bear in mind that the source compares estimated outcomes with actual outcomes to estimate prediction accuracy.

"Chess itself is a rather unpredictable game, especially if two players are close in rating performance and there are tons of games where lower-rated players upset higher-rated players. Therefore, a success rate of 55.64 % given the three possible game outcomes is not a bad result." [37]

3.3.3 Summary

List of Elo sources		
Author	**Pros**	**Cons**
Fadi Thabtah	Elo Rating explained Elo accuracy explained	More nitty-gritty details
Zhenyuan Fen	Elo unpredictability when small differences between players	Results need more comparison, give examples

3.4 Engines

This section contains a list of CEs that are Free and Open Source Software (FOSS). The CEs will be used in this project for measuring the ER for the handpicked CE. The only requirements for the opponents were that they are free to use, that they have a pre-compiled binary available for Linux, and that they have an ER in a similar range to that of the handpicked engine to avoid one dominating the other. All ER measurements presented in this list were taken from 40/15 matches (time control beginning at fifteen minutes where an additional fifteen minutes is awarded after every forty moves) performed by the Computer Chess Rating Lists (CCRL), one of the most trusted computer chess bodies.

1. Crafty
    - Crafty is a CE with an ER of 2953, according to the CCRL. Crafty was made primarily by Robert Hyatt, which began in the 1990s, and the final version was released in 2016. This is the handpicked

CE for this research project which will be optimised and used in experiments. This CE is handpicked because it is written in C which is a programming language where execution can be described in more detail than other programming languages. There will be two versions of Crafty, the original version that implements YBWC and the other version that implements Lazy SMP. Both versions will initially have the same ER of 2953 because ER is an estimation of a player's strength and both engines will compete against other engines with known ER to refine the ER [38].

2. Clovis III

   - Clovis III is a CE with an ER of 2864, according to the CCRL. Clovis is being developed by Jonathan McDermid, with version III released on March 1st 2023. It is a single-threaded UCI engine written in C++ using many techniques, including bitboards, negamax, transposition tables, and piece square tables.

3. Atlas 3.91

   - Atlas 3.91 is a CE with an ER of 2916, according to the CCRL. The developer of the engine, Andrés Manzanares Campillo, describes Atlas as a hobby project which is free to use according to CCRL. The latest version of the engine is Atlas 4.35 but the older version have an ER closer to Crafty. Atlas 3.91 include UCI protocol. The developer included contact information on the website. The first version of Atlas was released in April 2004, and version 3.91 was released in January 2018.

4. Amoeba 3.4

   - Amoeba is a CE with an ER of 3043, according to the CCRL. It was developed by Richard Delorme, with the latest version of 3.4 released on December 16th 2021. Amoeba is a multi-threaded UCI engine written in the D programming language.

5. Chess22k 1.14

   - Chess22k is a CE with an ER of 3117, according to the CCRL. It was developed by Sander Maassen vd Brink, with the latest version of 1.14 released on April 26th 2020. Chess22k is a multithreaded UCI engine written in Java.

Chapter 4

Method

4.1 Research Process

Step 1: Preliminary Research

1. The research begins with searching for research papers on chess engines from databases, for example, the KTH Library's database search engine and Google Scholar to get a baseline knowledge of the general architecture of chess engines, and how the different parts affect it.
2. Zotero was used to organise and collect sources. This makes things easier later on to create a reference table for the sources used for this thesis report.

Step 2: Formulate a research question

1. Before the preliminary research had started, the intention was to create a CE and document ways in which its performance could be improved through known techniques. Through the preliminary research, it was discovered that similar work had been conducted many times, leading to the conclusion that improvements in this area would be challenging with the current research. The focus was shifted to a comparative study of two parallelisation strategies, YBWC and Lazy SMP. It was believed that transforming the research into a comparative study would contribute more to the field of Chess research, given that not many scientific papers compare the two algorithms.

2. Then, the question of which parameters should be used to measure the performance of the algorithms for comparison arose. Interest was directed towards whether one parallel search algorithm would be faster, whether one would scale better to a higher core count, and whether one would result in stronger game-play.

Step 3: Literature Review

1. When sufficient knowledge had been gained from the preliminary research, a literature review was initiated on an area that was found to be interesting and under-explored: YBWC compared to Lazy SMP.
2. A comparison of both parallel search techniques in terms of efficiency and game performance was desired, so a literature review on ER was also initiated to gain a better understanding of how to quantify win probability.

Step 4: Research planning

1. Three aspects are to be measured: Sped, ER, and Scal.
2. Sped shall be based on time-to-depth measurements from different core counts. Time-to-depth is a measurement utilised in many papers encountered that records the time taken for the engine to search a specified number of moves in advance from a given starting position, or in other words, the time required to reach a depth level in the search tree. These measurements will be taken from a large number of middle-game positions to gain a broad sample size while avoiding the early book moves that are typically handled by the Graphical User Interface (GUI).
3. ER will be estimated by allowing the CB to play a substantial number of matches against other CB with a known ER using an automated matchmaking program. An ER for the CE will then be estimated based on statistics from its performance in previous matches and an example of an ER formula. The formulae for calculating Elo (4.1 and 4.2) [39, 38] require an initial rating to function. The initial ratings are taken from the CCRL, which the formulae 4.1 and 4.2 will then transform into a more refined rating.

$$ExpectedScore_A = 1/(1 + 10^{(Elo_B - Elo_A)/400}) \tag{4.1}$$

$$NewElo_A = Elo_A + K(Score_A - ExpectedScore_A) \tag{4.2}$$

4. Two types of Scal will be calculated; how Sped scales and how ER scales against increased core counts. Neither calculation requires bespoke measurements, instead using the measured data from the previous two items.
5. It has been concluded that the aspects that need to be documented for this research are the outcomes of the CB battles against other CB, the ER of those opponents, and the time-to-depth recordings. These shall all be recorded for each combination of parallel search algorithm and core count.

Step 5: Implement the parallel search algorithms

1. The first step in this process is to select an open source CE to implement the algorithms in. A few criteria were put in place before one was selected.
   - Must be FOSS
   - Written in a language understood by the authors.
   - Preferably already having YBWC implemented, as Lazy SMP is broadly considered easier to implement.
   - Must already support transposition tables, preferably lock-less.
   - Should ideally use traditional evaluation (as opposed to neural networks), as the authors are more familiar with it.
   - Should ideally be written with readability in mind over performance, to ease comprehension of its inner workings and enable modification.
2. With this in mind, Crafty by Robert Hyatt was chosen, it was the only CE found that managed to meet all requirements.
3. After the CE has been selected, a version implementing Lazy SMP will be developed. In addition, tests will be developed to measure execution time.

Step 6: Execute experiments

1. What data to be measured is described in section 4.3.
2. How the data shall be measured is described in section 4.4.

Step 7: Analyse results

Step 8: Discuss results

4.2 Research Paradigm

As discussed in section 1.5, a decision has been made to perform quantitative, positivistic research using an experimental research method and a deductive approach.

An experimental research strategy will be employed because the project aims to verify or falsify hypotheses. Experimental research is about changing dependent or independent variables to understand better how they would affect the outcome. This method helps the user to get a better understanding of the correlation between variables and the outcome.

The Ex Post Facto research strategy was considered but ultimately dismissed because it is best used when the researchers for some reason cannot manipulate the variables affecting the outcome, usually due to ethical concerns or when the requirements for reproducing some research are out of reach. It has been determined that neither of these poses a hindrance and that the data that can be gathered using the experimental strategy will be more aligned with the intended measurements.

As the experimental research strategy has been chosen exclusively, the use of experiments to collect data is the only natural choice. Experiments have been selected as the data collection method, as this aligns well with the experimental research strategy. While other options were available according to source [9], experiments appear to be the most valid choice, whereas the alternatives seemed to fall outside the scope of the research.

The data analysis will consist of both computational mathematics and statistics. Statistics will be used to summarise the collected data, identify patterns and anomalies, and determine probability distributions. Computational mathematics will be employed to analyse the efficiency of the algorithms, including time and space complexity, as well as for Elo calculation.

For quality assurance, the reliability, replicability, and validity of the method will be assessed. It is important to understand the extent to which the collected data deviates from run to run, the degree to which the method can be reused to obtain the same or similar results, and the level of trust that can be placed in the correctness of the measurements. These aspects are vital, as they directly affect the scientific credibility and impact of the findings. Furthermore, ethical considerations must be taken into account when

collecting, analysing, and reporting the data, as well as regarding the use of open-source solutions as part of the engine.

4.3 Data Collection

There are two kinds of data of interest in this project: the measured Sped and the ER. A separate measurement for Scal is not required, as it is derived from Sped.

4.3.1 Speedup

Sped will be measured as a relation between the time it took to execute a specific task with one core compared to executing the same task with multiple cores. The experiments will measure Sped from 50 different middle-game chess positions up to depth 20, where the positions are handpicked GM matches. The timer will start when the search begins and stop when the search has finished at the final depth. The mathematical formula 4.3 will be used to measure the Sped:

$$Sped_N = Time_1/Time_N \tag{4.3}$$

Where $Time_1$ is the time it took for one processor to finish the work, and $Time_N$ is the time it took N number of processors to finish the same work.

4.3.2 Elo Rating

ER can be measured as mathematical computation based on the CB previous performances against other CBs with known ER. Previous measurements by the CCRL will be used to get an initial ER for the CE. The ER will be refined by letting the CB play against other CBs with cutechess-cli and the new ER will be calculated with formulas 4.1 and 4.2. The more the CB win against other CBs the higher ER it gets [36, 37].

4.3.3 Scalability

Although there is no single formula for calculating Scal, it can be described by analysing a few other factors. With the speedup measurements for all core counts, regression analysis can be used to create a function that roughly describes what speedup can be expected for a given core count.

Furthermore, the efficiency of an increased core count can be calculated with 4.4. In this equation, N denotes the number of cores used and $Speedup$ is the result of equation 4.3. By calculating the efficiency of each core count from the experiments, a possible trend could be derived.

$$Efficiency_N = Speedup_N/N \tag{4.4}$$

The measurements here are taken from an identical problem size for all core counts, which means that strong scaling holds and Amdahl's law can be used to calculate the theoretical limit on speedup from the serial and parallel parts of the program. From this, since the actual speedup is known, an estimated size of the parallel portion of the program can be derived, as can be seen in equation 4.5.

$$Sped_N = 1/(s + (p/N)) \tag{4.5}$$

Where Sped is from formula 4.3, N is the number of cores used, s the part of the program that is sequential and p is the part of the program that can be executed in parallel. The condition must hold that $s + p = 1$ where from this condition p can be determined.

4.3.4 Ethics to method

A moral question worth considering is whether the results are being manipulated or if the findings do not contribute to chess research. It can be argued that these concerns do not present significant ethical dilemmas and are not worth taking into account.

4.3.5 Summary

The kinds of tests that will be executed in this research can be viewed in table 4.1 along with how many times those tests will be run at minimum for each configuration. The chess engine configurations are shown in table 4.2.

Test type	**Number of runs**
Time-to-depth	50 positions
Engine competition	40 (4 opponents $\times$ 10 matches)

Table 4.1: Table showing the types of tests and how many times they will be run.

Algorithm	Number of cores
YBWC	1
YBWC	2
YBWC	4
YBWC	8
Lazy SMP	1
Lazy SMP	2
Lazy SMP	4
Lazy SMP	8

Table 4.2: Table showing each configuration of parallel search algorithm and number of cores to be used for the tests.

4.4 Experimental design and Planned Measurements

This research will investigate what parallelisation strategy, Lazy SMP or YBWC, produces the best results. For this, three parameters will be investigated: Sped, Scal and ER. To measure how these parameters change in different conditions, the variables of the parallel search algorithm (YBWC or Lazy SMP) used and the number of cores the engine is run on will be used.

To be able to compare the two parallelisation algorithms fairly, they shall both be implemented in the same chess engine. Doing this will remove the variables that would arise from comparing two entirely different engines. For this project, the open-source chess engine Crafty was chosen as the base to build off of.

Crafty is an engine originally developed by Robert Hyatt in 1994 and was one of the first engines to use bitmaps for board representation. Crafty was one of the most influential chess engines of the 1990s and 2000s and was continuously developed until its last release, version 25.2 on October 29th 2016. Crafty was chosen for this project due to its simple design, well-commented code, and robust YBWC implementation that would allow us to focus on writing a forked version that instead implements Lazy SMP with relative ease.

Crafty uses the XBoard protocol for communication, which necessitates a frontend that supports both XBoard and UCI for us to be able to conduct tests against most modern engines. For this cutechess-cli was chosen, a very popular command-line frontend that supports both protocols, with many advanced

options available.

To measure Sped and Scal, an experiment will be conducted where the engine has to search some number of moves ahead from a given middle game position. The experiment uses a large number of initial positions, and each combination of parallel search algorithm and core count will be run from every position. The experiment shall record the time it took to complete the search to the given depth, and the number of nodes being processed per second by the engine. From this point, Sped and Scal can be derived, and the methodology for doing so is described in section 4.3.

Before the experiments begin there is a hypothesis that Lazy SMP is more scalable than compared to YBWC due to how the algorithms operate, the hypothesis will be verified or falsified through experiments. It is believed that by measuring Sped and Scal, it will be possible to estimate or conclude whether one algorithm is more, equally, or less scalable than the other. Additionally, the ER of both algorithms will be measured to estimate the extent of the ER boost gained by increasing the number of cores; this could also be described as the measurement of scalability for Sped and ER gained per core.

ER will be measured by letting the CB play against a given number of opponents of known ER a given number of times. Initially, the ER of both algorithms YBWC and Lazy SMP using one core will be calculated, with the number of cores being doubled and a new ER being calculated for up to eight cores. With a known ER for each combination of parallel search algorithm and core count, it will be possible to derive the amount of ER gained by each core increase for both algorithms.

It is worth mentioning that the same SEF will be used for both algorithms, meaning that when both algorithms examine a BS, they will assign it the same rating. This consistency enhances the accuracy of the comparison, as both algorithms will, in most cases, draw the same conclusions and evaluate a BS simultaneously. Furthermore, if testing is conducted on the same hardware and both algorithms face the same opponents, the comparative study between the two algorithms becomes more relevant, as they utilise the same or similar resources.

4.4.1 Test environment

Environment: KTH computer rooms or a personal computer at home source code, list of opponents, and scripts used to automate testing will be in an appendix.

4.4.2 Hardware and Software to be used

Hardware: either one of the authors' personal computers or a KTH school computer.

Software:

1. The source code of Crafty
2. JetBrains CLion (IDE)
3. cutechess-cli (automated matchmaking for local engines)
4. Overleaf (report writing)
5. Grammarly (grammar control for the report)
6. Microsoft Excel and Apple Numbers (spreadsheet data analysis)
7. Zotero (reference management)
8. Astah Professional (UML creator)

4.5 Assessing reliability and validity of the data collected

4.5.1 Validity of method

A quantitative method is presented in this report. It addresses how the choice of using YBWC or Lazy SMP affects the performance of a CE by measuring Sped, Scal, and ER for different core counts. The authors have determined these areas to be the most important concerning parallelisation of a CE, but other less central areas such as resource usage or performance on different operating systems could be of interest to some. Still, the chosen parameters cover the essential differences between the parallel search algorithms.

The chosen method should align well with existing metrics used to measure the performance of chess engines. Throughout the pre-study, most calculations of Sped and Scal were done using data from time-to-depth measurements. Many have chosen to calculate ER as suggested by this method, from conducting matches against other CEs, but others have chosen to make looser estimates such as online tests. From a quantitative perspective, the chosen method should give a reasonable ER measurement, but this depends on how many matches can be played in the allotted time for the project.

4.5.2 Reliability of method

The method's reliability will be asserted by running small-scaled versions of the experiments to see how much variance there is in the results. The results will also be compared against previous works, such as those listed in chapter 3, to see if they are in line with the characteristics of their results.

4.5.3 Data validity

The performance of YBWC and Lazy SMP will rely on both how they are implemented and the overall performance of the rest of the engine. Ideally, the entire engine would be perfectly implemented with all available improvements present, but this is unrealistic for many reasons – the most obvious of ones being the continued innovation in the strongest chess engines on the market, the technical skill of the authors of this report, and the time limit imposed on this project.

The open-source chess engine Crafty will be the basis for the project, which should greatly reduce the amount of work required before the tests can begin. Furthermore, this should allow more time to be spent on assuring the correct implementation of the parallel search algorithms.

The engine developed for this project will as such not be able to compete against the likes of AlphaZero or Stockfish, but that is not its purpose. The two parallel search techniques will be part of an otherwise identical engine in the tests. There may exist advanced improvements that would benefit one technique over the other and would therefore affect the data collected if implemented. The engine developed here will not focus on such advanced techniques, and will instead act more like a baseline comparison between YBWC and Lazy SMP.

The data that will be measured are relevant to the phenomena being

investigated. Calculating Sped and Scal requires time measurements, and are here recorded by letting the engine search a fixed number of moves ahead from a large number of middle game positions. Each combination of parallel search technique and the number of cores will search to the same depth and use the same positions to ensure the data generated by them are comparable with each other. Middle game positions are used because opening moves are typically handled by the GUI using a library, and all endgames with up to seven pieces have been solved with Syzygy tables [40]. It is therefore middlegame positions that best reflect the performance of an engine. Using a large number of positions further provides a wider sample size of data to analyse.

Researchers from other projects have used a similar approach to measure the data Sped [13, 26, 35].

It is difficult to correctly assess the playing strength of a player, let alone a CE, but calculating the ER is today the most recognised and common way to give a score to the probability of a player's likelihood to win a match. This calculation requires win/loss/draw results from many matches to be accurate, but the data itself has a low risk of being inaccurate assuming the engines are allocated resources fairly.

4.5.4 Reliability of data

Other researchers that have conducted similar kinds of research have trusted their data to be objective but have still taken it with a grain of salt [26, 28, 30, 32]. The phenomenon that when executing the same task with the same equipment would yield the same results could happen but is not a given fact. To measure the reliability of the data measurements of normal deviation between execution to estimate how much deviation can be expected. The data can be assumed to be objective but the accuracy of the data can be taken with a grain of salt.

4.6 Planned Data Analysis

This research is quantitative, where the data analysis strategy will consist of both computational mathematics and statistics. The collected data will be visualised with figures with analysis and conclusions following shortly after. This analysis method goes hand in hand with other researchers' approaches [31, 34, 35].

4.6.1 Data Analysis Technique

Regression analysis will be applied to the collected data. This will develop mathematical functions that estimate the changes in execution speed and playing strength for parallel search techniques using different core counts. From the collected data a mathematical function can be estimated to estimate how much Sped and ER is gained for the algorithms YBWC and Lazy SMP for every core added [41]. From previous measurements between independent and dependent variables, future outcomes can be estimated.

The analysis can be used for calculating variances in statistical data [42, ch. 14]. The method will be used when calculating the differences in execution time between identical executions to estimate an average deviation in the collected data.

4.6.2 Software Tools

For creating figures and writing the report the online tool Overleaf will be used for this project. A spreadsheet tool will be used to organise the collected data, perform regression analysis, and identify the distribution of data.

4.7 Evaluation framework

During the pre-study session of this research, other researchers' work have been examined and presented in chapter 2 of this report. The results of their research have been assessed but their methods have not been assessed as thoroughly. Even though the method part has not been examined in greater detail the other researchers' method of approach is similar to the method chosen in this project.

The research method can be summarised as a five-stage method:

1. Perform tests
2. Evaluate results
3. Plan new tests
4. Iterate steps 1-3
5. Result and analysis

The research method in itself is flexible and not detailed but should produce results. The validity of the results can be discussed because the method itself is not rigorous and detailed. There is in place an overarching plan of how the test should be measured and for what purpose. There is a risk for deviation from set research measurement goals in step 2 but if diligently evaluated that could be avoided or if an error occurs a step back in the process and re-evaluate the results.

4.8 System documentation

Laptop A with Blabla and Laptop B with Blabla and UCI Protocol interface blabla.

Chapter 5

What you did

5.1 Hardware/Software design …/Model/Simulation model & parameters/…

5.1.1 Alex computer spec:

5.1.2 Documentation of script:

A command script was written for the tests in order to automatise the tests and to increase validity of tests by letting a computer do it. The command script will contain commands about what we want crafty to measure, in this case it is Sped. The text file will contain the commands that Crafty will execute as well as references to chess games that Crafty will use as material for the tests.

5.1.3 The two Crafty implementations(link to drive?):

5.2 Implementation …/Modeling/Simulation/…

5.2.1 Analysis

Implementing Lazy SMP began with downloading version 25.2 of Crafty from the official repository and going through its source code to become familiar with the design of the engine before making changes to it. The basic flow of the program can be described as follows:

1. Create the game tree.

2. Read the command line and configuration file for options that must be set before the engine is initialised.

    2.1 `Option()` (in `option.c`) identifies all valid options and calls the appropriate functions to configure the engine appropriately.

3. Initialise the chessboard along with all data structures required for the engine to function using `Initialize()` (in `init.c`)

4. Reread command line arguments and the configuration file for options that must be configured after initialisation.

    4.1 `Option()` identifies all valid options and calls the appropriate functions to configure the engine appropriately.

5. Enter the main loop of the program.

    5.1 Read text entered by the user.

        5.1.1 If `Option()` recognises an option from the text, that option will be configured.

        5.1.2 If no option is recognised, `InputMove()` (in `input.c`) is used to check the text for moves and extract them (if found).

    5.2 When a move has been input, `MakeMoveRoot()` (in `make.c`) is used to make that move. After the move has been made, the current player's turn is flipped (white to black, or vice versa).

    5.3 Check if the made move resulted in a draw by three-fold repetition, the 50-move rule, or by insufficient material before conducting a search.

    5.4 Call the `Iterate()` function (in `iterate.c`) to start the iterative deepening search.

        5.4.1 Initialise the variables necessary for the search. Most are set to a default value, but some will use the results of the previous search when available.

        5.4.2 If the `smpmt` option was set to a value greater than 1, it will create the specified number of threads using `pthread_create()`.

            5.4.2.1 When a thread is created it goes to the `ThreadInit()` function (in `thread.c`). All helper threads are initialised here, and when all are done they call the

`ThreadWait()` function where they wait until a searching thread wants help.

5.4.3 If the current search is not the first one, store the principle variation from the previous search in the hash table to make sure it is accessible.

5.4.4 The `Search()` function in `search.c` is used to search one iteration.

5.4.4.1 First determine if the match should end in a draw due to repetition or the 50-move rule before going further.

5.4.4.2 Use the transposition table to see if the current position has already been encountered, and if so return the stored value.

5.4.4.3 If a Syzygy table exists, there are at most six pieces left, and the last move wasn't a capture, use the Syzygy table to find a response without searching.

5.4.4.4 If possible try to do a null-move search to get a cut-off without having to perform a full search.

5.4.4.5 If the current position is part of the principle variation but was not found in the transposition table, recursively call the `Search()` function again with a search depth that is two less.

5.4.4.6 When none of the previous checks are fruitful, perform a full search with the `SearchMoveList()` function with the `mode` parameter set to `serial`.

5.4.4.7 `SearchMoveList()` will initiate variables depending on if `mode` is set to `serial` or `parallel`.

5.4.4.8 Go through the move list

5.4.4.9 Make the move from the list using the `MakeMove()` function in `make.c`.

5.4.4.10 Determine if the move puts the opponent in check, as that will slightly change how the search is performed.

5.4.4.11 Use Futility Pruning and Late Move Pruning to reduce the size of the search tree.

5.4.4.12 Use Late Move Reductions to make the search depth smaller if it appears late in the move list, and is therefore considered likely to be poor.

5.4.4.13 Use the `SearchMove()` function to perform a Principle Variation Search using a smaller beta value to get a null window. The function recursively traverses the branches from the given position by calling the `Search()` function. If the final depth level has been reached (it is at a terminal node), the `Quiesce()` function (in quiesce.c) is called instead to perform a Quiescence Search where static evaluation is used to get a stand/pat score and it tries to generate captures and checks to improve upon it. `SearchMove()` will perform a second search with the original beta value if the value produced is between the alpha and beta values (and therefore not a null window).

5.4.4.14 Unmake the move and check its result. If it is a Fail-High it cancels the search and tells any sibling threads to stop searching. If it is In-Window it will update the alpha and beta values and continue. If it is Illegal it will just continue to the next move in the list.

5.4.4.15 Since the oldest brother node has been searched in this branch it will check if the current node is an appropriate splitting point with the `ThreadSplit()` function (in thread.c). If so, it will use the `Split()` function to choose an available thread to copy data from the parent thread to and tell it to start searching.

5.4.4.16 When all moves have been gone through, the best move will be returned to the parent thread if it is searching in parallel. If it is instead serial, the best move is returned, or MATE/DRAW if no legal move was found.

5.4.5 Check if the result from the iteration is either a Fail-Low or Fail-High, in which case Alpha or Beta respectively is updated for the next iteration.

5.4.6 When all iterations have finished the results of the search are printed according to the user settings.

5.4.7 If the `smp_nice` flag is set and `ponder` is not (for example during computer matches), all helper threads are killed. The resulting value of the search is then returned to the main method.

5.5 Check if the opponent is offering a draw and respond depending on the search results.

5.6 Check if Crafty has been checkmated or is in a stalemate, and if so end the game and respond accordingly.

5.7 Check if Crafty wants to offer a draw or resign depending on the search results.

5.8 Print the move chosen from the search.

5.9 Change to the opponent side to see if the chosen move leads to a draw. If it does, end the game and inform the opponent.

5.10 Save the ponder move of the principle variation so it can be used in the next search.

5.2.2 Implementing Lazy SMP

The programming began when sufficient knowledge of Crafty's inner workings had been acquired. The following subsections will describe the process for implementing Lazy SMP in Crafty.

5.2.2.1 Serialisation

The first change made to the source code was to make the search functions single-threaded. This is because the threads should perform their search independently from one another and communicate using the transposition table.

The serialisation was done by modifying `search.c` to remove all checks for `mode` being set to `parallel` and all checks for `CPUS` (a variable defined when compiling that tells Crafty how many threads it is allowed to use) being greater than one. No further change was necessary for serialisation as the remaining code will execute the same way as when the original program is run single-threaded.

5.2.2.2 First attempt

Unfortunately, the first attempt at writing a Lazy SMP implementation was unsuccessful. For the sake of completeness, however, the process will still be described.

The first attempt to implement Lazy SMP was based on using a customised waiting function in `thread.c` that all threads, including the main thread, start their search from simultaneously. In `thread.c`, this new waiting function was called instead of `ThreadWait()` after having been initialised

in `ThreadInit()`. The function was also called by the main thread in `iterate.c` in place of the `Search()` call.

The wait function begins with entering an infinite `while` loop. The first operation in the loop is to use `pthread_barrier_wait()` to ensure the main thread is ready before data is copied from it to the helper threads. Since much of the development was done on a MacBook and because macOS does not include built-in `pthread_barrier` support, a custom implementation made by Aleksey Demakov called DarwinPthreadBarrier was used instead when compiled on Mac computers.

A new function based on `CopyFromParent()` was used to give the helper threads data needed to begin their searches. This function copies all the data that `CopyFromParent()` does, except for statistical counters as that would overwrite the counters of the helper threads. This function is only called when the ID of the thread is not zero because the main thread would never need to copy data from itself to itself.

A second `pthread_barrier_wait()` call is made so that the main thread does not start a search that could change its data before it is copied by the helpers. After the barrier, all threads start searching in parallel using the `Search()` function.

When a result has been returned to the main thread, it will return that value to `iterate.c` where the rest of the program continues as normal.

This version of the program compiled correctly, but would crash due to segmentation faults when being run with multiple threads. The cause of this is unknown but is in hindsight thought to be because some of the data copied to the helper threads not having been properly initialised before being copied.

Regardless of the reason why this attempt failed, the program was rolled back to the changes made by the serialisation and the Lazy SMP implementations of other engines were investigated more closely before making a second attempt.

5.2.2.3 Second attempt

The second and final attempt at implementing Lazy SMP takes inspiration from the Cheng chess engine. More specifically it is based on the pseudo-code presented by Cheng's author, Martin Sedlak, on the Computer Chess Club forums when describing Cheng's Lazy SMP implementation. The three pseudo-code functions detailed in Sedlak's forum post are presented in figure 5.1 [43].

Similar to the first attempt, a custom waiting loop is responsible for the majority of the implementation. This function is only called by the helper threads after they have been initialised. The function, called `LazyWait()`, first calls the `CalculateDepthOffset()` function to get an offset to how deep the threads will search. This function uses the ID of the calling thread to give an offset that is 0 for $1/2$, 1 for $1/4$, 2 for $1/8$, and so on. This distribution is based on that used by Østensen in his master thesis "A Complete Chess Engine Parallelized Using Lazy SMP" [26]. The offset is in place to coerce the threads into searching different paths of the search tree to better utilise the transposition table.

After having acquired the offset, `LazyWait()` enters an infinite `while` loop. The thread is then stuck in another loop where it continuously checks if the search has been terminated while it is stopped. Both cases are changed by the main thread, when terminated the thread will return and when no longer stopped it will start searching using `Search()` to a depth equal to the current iteration depth plus the offset. The thread will set itself to stopped after completing searching, at which point the infinite loop repeats.

Immediately before entering the iterative deepening loop in `iterate.c`, a new function called `CopyToHelpers()` is called by the main thread. This function is based on the existing `CopyFromParent()` function, but has been modified to take the `TREE` structure of the main thread and copy all the relevant data to all helper threads.

Also in `iterate.c`, the main thread will start and stop all helper threads immediately before and after beginning its own search respectively. This is only done if more than one thread is in use and if the current iteration is not the first to make sure that that there are root moves available to be searched. The helper threads are started with the `StartHelpers()` function that gives all helper threads the alpha, beta, and side to move. They are stopped with the `ThreadStopAll()` function, that works by setting the `stop` flag of all threads to 1.

There are a few additional changes done for this implementation. To begin with, the `TREE` structures of the helper threads are initiated by `ThreadInit()` in `thread.c`, instead of being copied from the the main thread when being assigned a branch to search.

Another change was to multiply the size of the `root_moves` array by the number of threads, so that each thread gets their own copy of all root moves. This was done because the array is sometimes rearranged during the search,

and if multiple threads did this at the same time the data could be damaged.

A similar change was to make `failhi_delta` and `faillo_delta` arrays, so that each thread has its own value. If they all shared the same values, a fail low or fail high could update the alpha or beta to create a larger window than necessary.

5.2.3 Writing The Tests

Two python scripts were written in order to automate the testing process. The first script (which can be seen in appendix XX) is used to perform the time-to-depth tests. It should be placed in the same directory as the Crafty executable along with the 30 different GameXX.txt files containing the moves that set up the position to search from.

The script takes three parameters: the first GameXX.txt file to search, the last GameXX.txt file to search from, and the number of threads that Crafty should be run with.

It then executes crafty, tells it to use a 256 MB transposition table, a 64 MB pawn hash table, to not ponder, to search to the depth 25, to use at most 120 seconds to search, and how many threads to use before starting the search. When the script detects that the search has finished it saves the time it took to a log file and terminates Crafty. This process is repeated 50 times for each GameXX.txt file within the range GameXX.txt files specified by the user.

The second script is used to automate matches between two versions of crafty and the opponents. It requires that both versions of crafty have been configured in cutechess under the names ”crafty-ybwc” and ”crafty-lazy” and that all opponents have been configured with the names ”Amoeba”, ”Atlas”, ”Chess22k”, and ”Clovis” before starting. The script must be placed in the same directory as cutechess-cli. To use different engines than those used in this thesis, replace the names in the opponents array with those corresponding to the engines configured in cutechess.

The script takes one commandline parameter: the number of threads that Crafty shall use.

5.2.4 Executing The Tests

The tests were run on one of the authors' personal computers. The relevant specifications are listed below:

- CPU: AMD Ryzen 5 3600 (6 cores, 3.6 GHz, 32 MB L3 cache)

- RAM: Corsair Vengeance LPX 16 GB (2×8 GB dual-channel, 3200 MHz, DDR4)
- Operating System: Ubuntu 24.04.1 LTS (running on Windows Subsystem for Linux, Windows 11 Pro 23H2)

Before running the tests, all non-essential background applications were closed to free up system resources. The Windows Task Manager and Core Temp (version 1.18.1) were used to monitor resource usage and CPU temperature respectively. Testing only began when Task Manager reported 0% CPU usage across all cores and Core Temp reported temperatures under 40°C. A bug in Windows sometimes caused the ”CTF Loader” process (a Windows service responsible for handling some alternate user input methods[44]) to use around 25% of the available CPU utilisation and temperatures of over 50°C, but this was resolved by force-closing and relaunching it (by running ctfmon.exe).

The time-to-depth tests were run on 1, 2, 4, and 8 threads, while the Elo tests were run on 1, 2, and 4 threads. Nothing more was needed for the time-to-depth tests, but the Elo tests required more work to calculate the Elo ratings. Two additional python scripts were as such written.

The first script goes through the logs created when playing the matches and calculates the number of won matches, the number of lost matches, the number of drawn matches, and the win rate of Crafty against each opponent for all core counts and writes it to a text file.

The second script also goes through the logs, but instead calculates an Elo rating for Crafty based on the results from all matches for every core count. To calculate the Elo, an initial rating for Crafty, ratings for the opponents, and a K-factor are all required per formulae 4.1 and 4.2. The initial rating for Crafty and the opponents' ratings were taken from the CCRL, while the K-factor is a command-line parameter.

```
IterativeDeepening:
    synchronize smp threads (copy age, board, history,
        repetition list, multipv => helpers)
    depth 1 with full width window on 1 thread
    loop (depth=2 .. max)
        AspirationLoop:
            (as usual)
            start helper threads( depth, alpha, beta )
            root( depth, alpha, beta)
            stop helper threads
            (rest as usual)
        end aspiration loop
    end id loop

starting helper threads:
    clear smp abort flag
    for each helper thread:
        copy rootmoves and minimum qs depth => helper
        signal helper to start root search at current
            depth (add 1 for each even helper (assuming
            0-based indexing) with aspiration alpha,
            beta bounds and wait until helper starts
            searching

aborting helper threads:
    set abort flag for each helper and wait for each to
        stop searching
```

Figure 5.1: Cheng Lazy SMP Pseudo-code

Chapter 6

Results and Analysis

This chapter will present the results and discussions surrounding them.

6.1 Major results

The major results are divided into two categories; those obtained from the time-to-depth tests (in section 6.1.1) and those from the Elo-rating tests (in section 6.1.2).

6.1.1 Time-to-Depth Test Results

inledande text To determine the speedup gained by Lazy SMP and YBWC when increasing the number of cores, a time-to-depth test was conducted. In this test, both algorithms were tasked with calculating the best move from a set of chess positions up to a certain depth, repeated x number of times, to then calculate the median with varying numbers of cores at their disposal. The results of the tests are presented in graph **??**.

grafen

summering/ansats

inledande text An important factor to consider when comparing YBWC and Lazy SMP is the standard deviation in the time taken to solve a chess problem. Does one parallelization strategy consistently solve a specific problem in a shorter time span than the other? If one algorithm solves most problems more quickly, it would be considered more reliable. The results of these tests are shown in graph **??** below.

grafen

summering/ansats

inledande text An important factor to consider when comparing YBWC and Lazy SMP is the standard deviation in the time taken to solve a chess problem. Does one parallelization strategy consistently solve a specific problem in a shorter time span than the other? If one algorithm solves most problems more quickly, it would be considered more reliable. The results of these tests are shown in graph **??** below.

källdata graf

6.1.2 Elo-rating Test Results

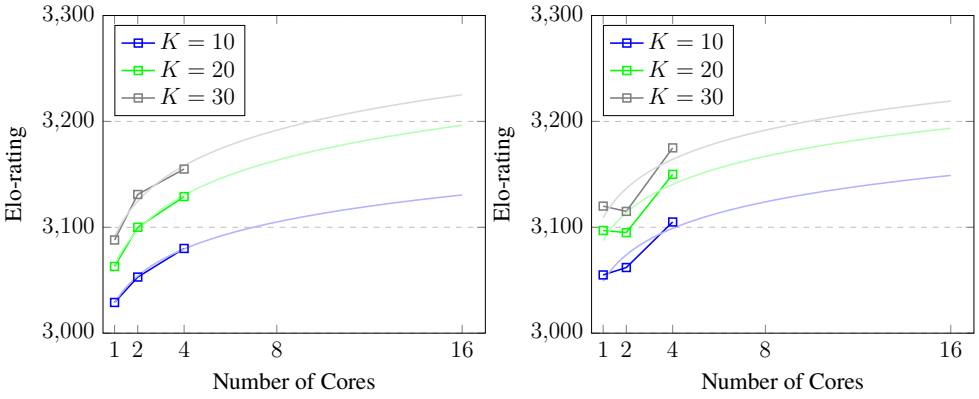
This section will present the results of the Elo-rating tests. The results presented come from Crafty's matches against four different opponents. Both the regular version of Crafty 25.2 with the YBWC parallelisation and the modified Lazy SMP version were tested. Crafty played 30 matches against each opponent (15 times as white and 15 times as black), with a time control of 0/3 (a three-minute timer without extensions). The matches were repeated with one core, two cores, and four cores allocated to Crafty.

Figure 6.1 shows two graphs depicting the calculated Elo-ratings for Lazy SMP and YBWC for the three core counts. The Elo-ratings have been calculated with three different K-factors (10, 20, 30) to show how this choice affects the rating. The K-factor determines how much a player's Elo should increase or decrease after a match, and as such beginners typically use high K-factors, while more experienced players use lower K-factors. This is clearly seen in both graphs by how the plots are spaced out, with $K = 10$ giving the lowest Elo and $K = 30$ the highest. It is also seen in the YBWC graph in the use of two cores, where K-factor 10 gives a small Elo increase over one core and the two higher K-factors give a small decrease. The Lazy SMP version, on the other hand, shows a clear two-core Elo increase for each K-factor.

Both graphs also have logarithmic functions that best fit the plots overlaid to visualise how additional cores are projected to impact the Elo-ratings. These functions were calculated in Apple Numbers by plotting the data points and selecting the "logarithmic" trendline. For Lazy SMP these functions and their associated R^2 values are:

- $K = 10 : y = 36.789 \times ln(x) + 3028.5, R^2 = 0.9988$
- $K = 20 : y = 47.609 \times ln(x) + 3064.3, R^2 = 0.9951$
- $K = 30 : y = 48.330 \times ln(x) + 3091.2, R^2 = 0.9739$

and for YBWC:



(a) Lazy SMP (b) YBWC

Figure 6.1: Plots showing how Elo-ratings change for Lazy SMP (a) and YBWC (b).

- $K = 10 : y = 36.067 \times ln(x) + 3049, R^2 = 0.8527$
- $K = 20 : y = 38.231 \times ln(x) + 3087.5, R^2 = 0.7217$
- $K = 30 : y = 39.674 \times ln(x) + 3109.2, R^2 = 0.6823$

The same Elo-ratings are also presented numerically in table 6.1.

Cores	K=10	K=20	K=30
1	3029	3063	3088
2	3053	3100	3131
4	3080	3129	3155

(a) Lazy SMP Elo-ratings.

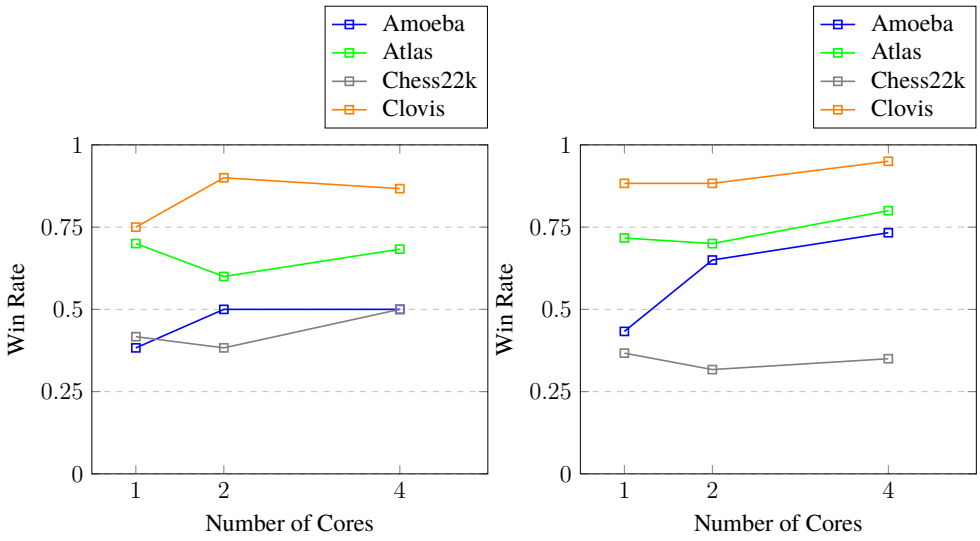
Cores	K=10	K=20	K=30
1	3055	3097	3120
2	3062	3095	3115
4	3105	3150	3175

(b) YBWC Elo-ratings.

Table 6.1: The Elo-ratings calculated with different K-factors as core-counts increase for Lazy SMP (a) and YBWC (b).

Another way to visualise the results of the matches is shown in figure 6.2. These graphs plot the win rate of the two versions of Crafty against the four opponents as core counts increase. The win rate is calculated as the average of all game outcomes against an opponent where winning is 1, losing is 0, and drawing is $\frac{1}{2}$.

The figure shows that the win rates against all opponents are very similar between Lazy SMP and YBWC when using one core. With two cores, Lazy SMP sees a sizably increased win rate against Clovis and Amoeba, a small decrease against Chess22k, and a sizable decrease against Atlas, with YBWC



(a) Lazy SMP

(b) YBWC

Figure 6.2: Plots showing how the win rates against four opponents change for Lazy SMP (a) and YBWC (b).

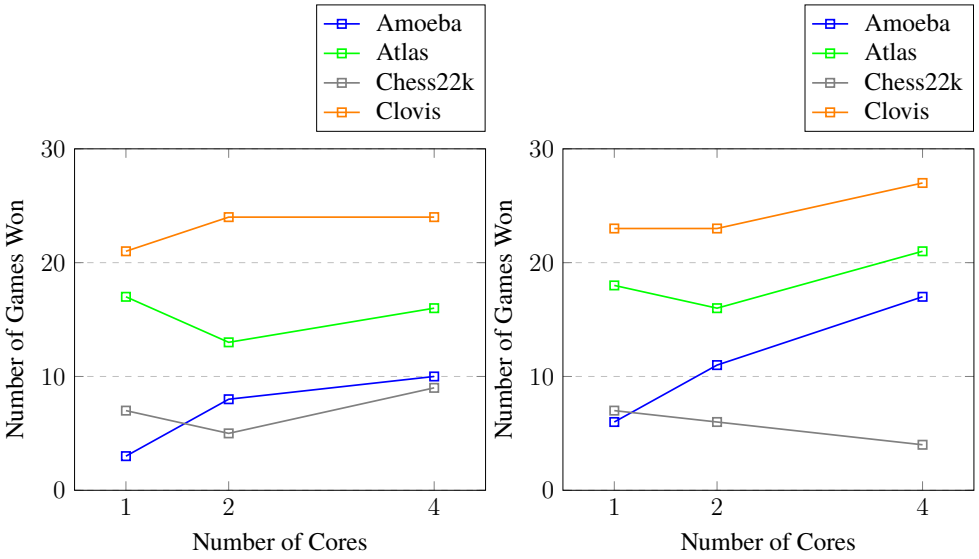
having a large increase against Amoeba, no change against Clovis, a slight decrease against Atlas, and a small decrease against Chess22k. Finally, Lazy SMP has a sizable increase against Atlas (though still slightly less than single-core) and Chess22k, no change against Amoeba, and a slight decrease against Clovis, with YBWC having a sizable increase against Amoeba, Atlas, and Clovis, and a slight increase against Chess22k (though still less than single-core).

The results can further be visualised by the total number of wins in figure 6.3, the number of losses in figure 6.4, and the number of draws in figure 6.5.

6.2 Reliability Analysis

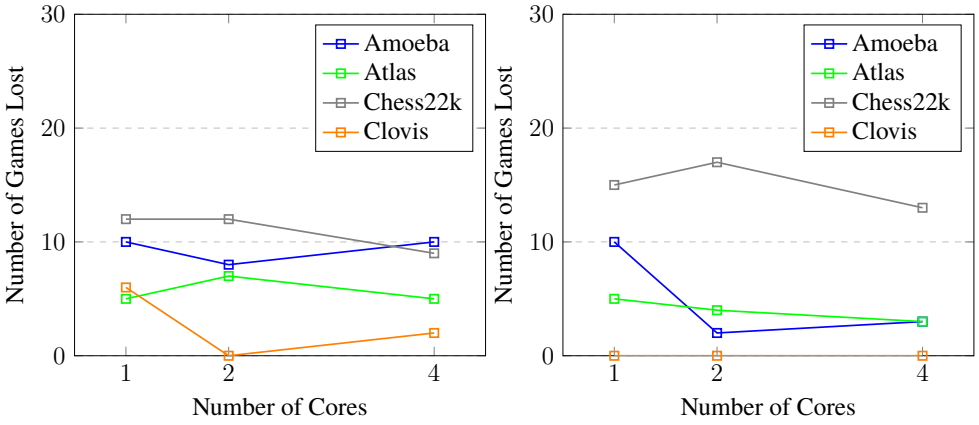
Access to computer equipment with 16 cores would have been useful for the time-to-depth tests. This setup would better demonstrate how effectively the two algorithms utilize abundant resources.

The measurements of Elo gained with an increased number of cores for the two algorithms would have been more accurate if the project's computer equipment had supported the use of 8 cores for a chess match. This would have provided four measurement points, an improvement over the current three.



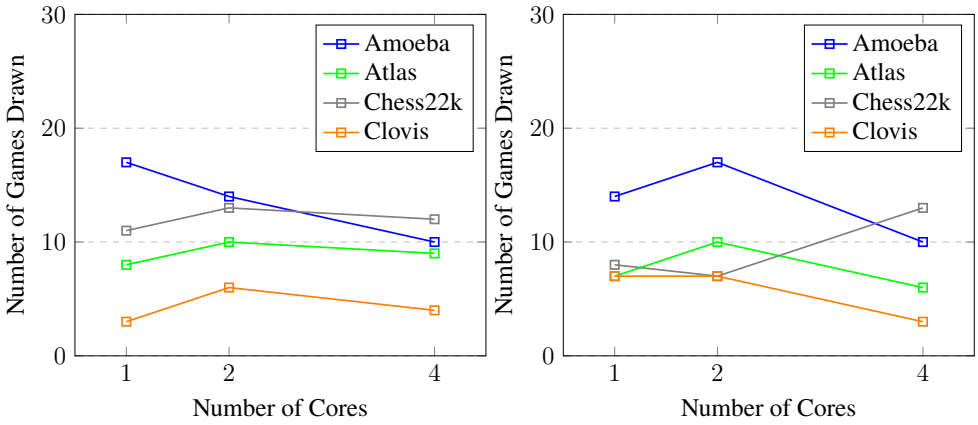
(a) Lazy SMP (b) YBWC

Figure 6.3: Plots showing how the number of wins against four opponents changes for Lazy SMP (a) and YBWC (b).



(a) Lazy SMP (b) YBWC

Figure 6.4: Plots showing how the number of losses against four opponents changes for Lazy SMP (a) and YBWC (b).



(a) Lazy SMP

(b) YBWC

Figure 6.5: Plots showing how the number of draws against four opponents change for Lazy SMP (a) and YBWC (b).

Considering that the Lazy SMP algorithm was implemented by two novice programmers, while the YBWC algorithm was developed by professionals, it is reasonable to assume that YBWC is more robustly implemented. This suggests that the Lazy SMP implementation may contain more flaws and be less fully optimized in comparison.

6.3 Validity Analysis

The Elo measurements in this project follow a standardized approach, using the same algorithm employed by organizations like FIDE and websites such as Chess.com. This alignment with common practice ensures that our method is reliable and produces comparable results.

The method used for the time-to-depth tests in this project is straightforward: select a chess position, measure how quickly the algorithms solve it using x cores, and repeat the process y times to calculate the median. In this study y was set to 50, but increasing this number would have improved the accuracy of the measurements. Our analysis revealed that a larger sample size would have resulted in data points more closely following a normal distribution when graphed. This underestimation of the required number of tests affected the precision of the results

Chapter 7

Discussion

The results, as shown in section 6.1.2 and illustrated in graph 6.1, indicate that the Elo ratings achieved by the YBWC and Lazy SMP algorithms are roughly similar. While there are slight differences in Elo gains depending on the number of cores used, their final Elo ratings are nearly identical. This suggests that both algorithms perform comparably. As discussed in section 3.3, predicting the winner whether a bot or human becomes increasingly difficult when the Elo difference is small. Consequently, if these two algorithms were to compete using the same number of cores (e.g., 1, 2, or 4 cores), the match would most likely result in a draw.

7.1 Notes

Structure for document

Chapter 8

Conclusions and Future work

8.1 Conclusions

8.2 Limitations

8.3 Future work

8.3.1 What has been left undone?

8.3.2 Next obvious things to be done

8.4 Reflections

References

[1] U.s. chess federation membership 2023. [Online]. Available: https://www.statista.com/statistics/1460028/membership-us-chess-federation/ [Page 1.]

[2] A. E. Soltis. Chess - history | britannica. [Online]. Available: https://www.britannica.com/topic/chess/History [Page 2.]

[3] C. Stapczynski. History of chess | who invented chess. [Online]. Available: https://www.chess.com/article/view/history-of-chess [Page 2.]

[4] C. E. Shannon, "Programming a computer for playing chess," in *Computer Chess Compendium*, D. Levy, Ed. Springer, pp. 2–13. ISBN 978-1-4757-1968-0. [Online]. Available: https://doi.org/10.1007/978-1-4757-1968-0_1 [Page 2.]

[5] L. Clark, "Remembering alan turing: from codebreaking to AI, turing made the world what it is today," section: tags. [Online]. Available: https://www.wired.com/story/turing-contributions/ [Page 3.]

[6] "The reconstruction of turing's "paper machine"." [Online]. Available: https://videolectures.net/turing100_kasparov_friedel_paper_machine/ [Page 3.]

[7] C. c. Team (CHESScom). Kasparov vs. deep blue | the match that changed history. [Online]. Available: https://www.chess.com/article/view/deep-blue-kasparov-chess [Page 3.]

[8] F. Friedel and G. Kasparov. Garry kasparov on how it all started. [Online]. Available: https://en.chessbase.com/post/garry-kasparov-on-how-it-all-started [Page 3.]

[9] A. Håkansson, "Portal of research methods and methodologies for research projects and degree projects." CSREA Press U.S.A, pp. 67–73. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-136960 [Pages 6, 7, and 40.]

[10] R. Hyatt, "Chess program board representations." [Online]. Available: https://craftychess.com/hyatt/boardrep.html [Pages 10, 11, and 12.]

[11] P. Keller. Chess and bitboards. [Online]. Available: https://pages.cs.wisc.edu/~psilord/blog/data/chess-pages/index.html [Pages 10 and 13.]

[12] P. Bijl and A. P. Tiet, "Exploring modern chess engine architectures." [Pages 14 and 15.]

[13] S. Vrzina, "Piece by piece: Building a strong chess engine." [Online]. Available: https://www.cs.vu.nl/~wanf/theses/vrzina-bscthesis.pdf [Pages 15, 22, 30, and 47.]

[14] V. Vuckovic, "Candidate moves method implementation in MiniMax search procedure of the achilles chess engine," in *2015 12th International Conference on Telecommunication in Modern Satellite, Cable and Broadcasting Services (SIKS)*. doi: 10.1109/ SKS.2015.7357795 pp. 314–317. [Online]. Available: https://ieeexplore.ieee.org/document/7357795 [Page 17.]

[15] D. J. Edwards and T. P. Hart, "The alpha-beta heuristic," accepted: 2004-10-04T14:39:10Z. [Online]. Available: https://dspace.mit.edu/handle/1721.1/6098 [Pages 18 and 19.]

[16] D. M. Breuker, J. Uiterwijk, and J. van den Herik, "Information in transposition tables." [Page 19.]

[17] Chessprogramming wiki. [Online]. Available: https://www.chessprogramming.org/Main_Page [Page 20.]

[18] Getting started - chessprogramming wiki. [Online]. Available: https://www.chessprogramming.org/Getting_Started [Page 20.]

[19] Programming a simple java chess engine - logic crazy - YouTube. [Online]. Available: https://www.youtube.com/playlist?list=PLQV5mozTHmaffB0rBsD6m9VN1azgo5wXl [Page 20.]

[20] Evaluation - chessprogramming wiki. [Online]. Available: https://www.chessprogramming.org/Evaluation#Basic_Evaluation_Features [Pages 20 and 21.]

[21] Simplified evaluation function - chessprogramming wiki. [Online]. Available: https://www.chessprogramming.org/Simplified_Evaluation_Function [Page 21.]

[22] B. A. Abdelghani, J. Dari, and S. Banitaan, "Comparing traditional and deep learning approaches in developing chess AI engines," in *2023 3rd International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)*. doi: 10.1109/ICECCME57830.2023.10252232 pp. 1–7. [Online]. Available: https://ieeexplore.ieee.org/document/10252232 [Page 22.]

[23] J. Madake, C. Deotale, G. Charde, and S. Bhatlawande, "CHESS AI: Machine learning and minimax based chess engine," in *2023 International Conference for Advancement in Technology (ICONAT)*. doi: 10.1109/ICONAT57137.2023.10080746 pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/10080746 [Page 22.]

[24] E. Vazquez-Fernandez and C. A. C. Coello, "An adaptive evolutionary algorithm based on tactical and positional chess problems to adjust the weights of a chess engine," in *2013 IEEE Congress on Evolutionary Computation*. IEEE. doi: 10.1109/CEC.2013.6557727. ISBN 978-1-4799-0454-9 978-1-4799-0453-2 978-1-4799-0451-8 978-1-4799-0452-5 pp. 1395–1402. [Online]. Available: http://ieeexplore.ieee.org/document/6557727/ [Page 22.]

[25] E. Vazquez-Fernandez, C. A. Coello Coello, and F. D. S. Troncoso, "An evolutionary algorithm coupled with the hooke-jeeves algorithm for tuning a chess evaluation function," in *2012 IEEE Congress on Evolutionary Computation*. IEEE. doi: 10.1109/CEC.2012.6252977. ISBN 978-1-4673-1509-8 978-1-4673-1510-4 978-1-4673-1508-1 pp. 1–8. [Online]. Available: http://ieeexplore.ieee.org/document/6252977/ [Page 22.]

[26] E. F. Østensen, "A complete chess engine parallelized using lazy SMP." [Online]. Available: https://www.semanticscholar.org/paper/A-Complete-Chess-Engine-Parallelized-Using-Lazy-SMP-Østensen/623825b7669ae053cdcacce69b51a6e8a927cbfb [Pages 22, 23, 24, 30, 47, and 57.]

[27] “Parallel algorithms for machine intelligence and vision.” [Online]. Available: https://link.springer.com/10.1007/978-1-4612-3390-9 [Page 23.]

[28] R. Feldmann, P. Mysliwietz, and B. Monien, “A fully distributed chess program.” [Online]. Available: https://www.semanticscholar.org/paper/A-Fully-Distributed-Chess-Program-Feldmann-Mysliwietz/6b75facdf4608cbd798092ec6eb5436b2209e361 [Pages 23, 27, and 47.]

[29] D. Homan. Lazy SMP, part 2 - TalkChess.com. [Online]. Available: https://www.talkchess.com/forum/viewtopic.php?t=46858 [Page 23.]

[30] D. R. Rasmussen, “Parallel chess searching and bitboards.” [Online]. Available: https://www.semanticscholar.org/paper/Parallel-Chess-Searching-and-Bitboards-Rasmussen/969a4b67c5e361bd34998c6e5b40221730526ad5 [Pages 26 and 47.]

[31] A. Ura, D. Yokoyama, and T. Chikayama, “Two-level task scheduling for parallel game tree search based on necessity,” vol. 21, no. 1, pp. 17–25. doi: 10.2197/ipsjjip.21.17. [Online]. Available: https://www.jstage.jst.go.jp/article/ipsjjip/21/1/21_17/_article [Pages 26 and 47.]

[32] K. Himstedt, “GridChess: Combining optimistic pondering with the young brothers wait concept,” vol. 35, no. 2, pp. 67–79. doi: 10.3233/ICG-2012-35202 Publisher: IOS Press. [Online]. Available: https://content.iospress.com/articles/icga-journal/icg35202 [Pages 26 and 47.]

[33] D. Kopec and I. Bratko, “THE BRATKO-KOPEC EXPERIMENT: A COMPARISON OF HUMAN AND COMPUTER PERFORMANCE IN CHESS,” in *Advances in Computer Chess*, ser. Pergamon Chess Series, M. R. B. Clarke, Ed. Pergamon, pp. 57–72. ISBN 978-0-08-026898-9. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780080268989500097 [Page 28.]

[34] H. Brange, “Evaluating heuristic and algorithmic improvements for alpha-beta search in a chess engine.” [Online]. Available: http://lup.lub.lu.se/student-papers/record/9069249 [Pages 31 and 47.]

[35] K. Hazama and H. Ebara, “Branch and bound algorithm for parallel many-core architecture,” in *2018 Sixth International Symposium*

on Computing and Networking Workshops (CANDARW). doi: 10.1109/CANDARW.2018.00058 pp. 272–277. [Online]. Available: https://ieeexplore.ieee.org/document/8590912 [Pages 32 and 47.]

[36] F. Thabtah, A. J. Padmavathy, and A. Pritchard, "Chess results analysis using elo measure with machine learning." doi: 10.1142/S0219649220500069 Publisher: World Scientific Publishing Company. [Online]. Available: https://www.worldscientific.com/worldscinet/jikm [Pages 34 and 41.]

[37] Z. Fan, Y. Kuang, and X. Lin, "Chess game result prediction system." [Online]. Available: https://cs229.stanford.edu/proj2013/FanKuangLin-ChessGameResultPredictionSystem.pdf [Pages 34 and 41.]

[38] R. Mittal. What is an ELO rating? [Online]. Available: https://medium.com/purple-theory/what-is-elo-rating-c4eb7a9061e0 [Pages 36 and 38.]

[39] G. Purohit. Elo rating algorithm. Section: DSA. [Online]. Available: https://www.geeksforgeeks.org/elo-rating-algorithm/ [Page 38.]

[40] N. Fiekas. KvK – syzygy endgame tablebases. [Online]. Available: https://syzygy-tables.info/ [Page 47.]

[41] D. L. Mohr, W. J. Wilson, and R. J. Freund, "Chapter 7 - linear regression," in *Statistical Methods (Fourth Edition)*, D. L. Mohr, W. J. Wilson, and R. J. Freund, Eds. Academic Press, pp. 301–349. ISBN 978-0-12-823043-5. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128230435000072 [Page 48.]

[42] G. Blom, J. Enger, G. Englund, J. Grandell, and L. Holst, *Sannolikhetsteori och statistikteori med tillämpningar*, 6th ed. Studentlitteratur. ISBN 978-91-44-12356-1 [Page 48.]

[43] M. Sedlak. Lazy SMP in cheng - TalkChess.com. [Online]. Available: https://www.talkchess.com/forum/viewtopic.php?t=55188 [Page 56.]

[44] ZarsuMcAwesome and DaveM121. What does the task in task manager called CTF loader actually do? [Online]. Available: https://answers.microsoft.com/en-us/windows/forum/all/what-does-the-task-in-task-manager-called-ctf/629689ae-ba70-41de-9b4c-ad438a63e384 [Page 59.]

If you do not have an appendix, do not include the \cleardoublepage command below; otherwise, the last page number in the metadata will be one too large.

TRITA-EECS-EX- 2023:0000

€€€€ For DIVA €€€€

{
”Author1”: { ”Last name”: ”Säfström”,
”First name”: ”Alex”,
”Local User Id”: ”u100001”,
”E-mail”: ”alexsaf@kth.se”,
”organisation”: {”L1”: ”School of Electrical Engineering and Computer Science”,
}
},
”Author2”: { ”Last name”: ”Yllmark”,
”First name”: ”Simon”,
”Local User Id”: ”u100002”,
”E-mail”: ”syllmark@kth.se”,
”organisation”: {”L1”: ”School of Electrical Engineering and Computer Science”,
}
},
”Cycle”: ”1”,
”Course code”: ”II42X/II43X”,
”Credits”: ”15.0”,
”Degree1”: {”Educational program”: ”Degree Programme in Computer Engineering”
,”programcode”: ”TIDAB”
,”Degree”: ”Bachelors degree”
,”subjectArea”: ”Technology”
},
”Degree2”: {”Educational program”: ”Bachelor’s Programme in Information and Communication Technology”
,”programcode”: ”TCOMK”
,”Degree”: ”Bachelors degree”
,”SubjectArea”: ”Technology”
},
”Title”: {
”Main title”: ”Young Brothers Wait Concept or Lazy SMP?”,
”Subtitle”: ”A Comparative Study of Parallel Search Techniques in a Chess Engine”,
”Language”: ”eng” },
”Alternative title”: {
”Main title”: ”Young Brothers Wait Concept eller Lazy SMP?”,
”Subtitle”: ”En Jämförande Studie av Parallella Sökteknikеr i en Schackmotor”,
”Language”: ”swe”
},
”Supervisor1”: { ”Last name”: ”Al-Zarqawee”,
”First name”: ”Aws”,
”Local User Id”: ”u100003”,
”E-mail”: ”awsj@kth.se”,
”organisation”: {”L1”: ”School of Electrical Engineering and Computer Science”,
”L2”: ”Computer Science” }
},
”Examiner1”: { ”Last name”: ”Sung”,
”First name”: ”Ki Won”,
”Local User Id”: ”u1d13i2c”,
”E-mail”: ”sungkw@kth.se”,
”organisation”: {”L1”: ”School of Electrical Engineering and Computer Science”,
”L2”: ”Computer Science” }
},
"National Subject Categories": "10201, 10206",
”Other information”: {”Year”: ”2024”, ”Number of pages”: ”1,76”},
”Copyrightleft”: ”copyright”,
”Series”: { ”Title of series”: ”TRITA-EECS-EX” , ”No. in series”: ”2023:0000” },
”Opponents”: { ”Name”: ”A. B. Normal & A. X. E. Normalè”},
”Presentation”: { ”Date”: ”2022-03-15 13:00”
,”Language”:”eng”
,”Room”: ”via Zoom https://kth-se.zoom.us/j/ddddddddddd”
,”Address”: ”Isafjordsgatan 22 (Kistagången 16)”
,”City”: ”Stockholm” },
”Number of lang instances”: ”2”,
”Abstract[eng]”: €€€€
€€€€,
”Keywords[eng]”: €€€€
Chess Engine, Young Brothers Wait Concept, Lazy SMP, Parallel Search, Scalability, Elo Rating €€€€,
”Abstract[swe]”: €€€€
€€€€,
”Keywords[swe]”: €€€€
€€€€,
}

acronyms.tex

```
%%% Local Variables:
%%% mode: latex
%%% TeX-master: t
%%% End:
% The following command is used with glossaries-extra
\setabbreviationstyle[acronym]{long-short}
% The form of the entries in this file is \newacronym{label}{acronym}{phrase}
%                                      or \newacronym[options]{label}{acronym}{phrase}
% see "User Manual for glossaries.sty" for the  details about the options, one example is shown below
% note the specification of the long form plural in the line below
\newacronym[longplural={Debugging Information Entities}]{DIE}{DIE}{Debugging Information Entity}
%
% The following example also uses options
\newacronym[shortplural={OSes}, firstplural={operating systems (OSes)}]{OS}{OS}{operating system}

% note the use of a non-breaking dash in long text for the following acronym
\newacronym{IQL}{IQL}{Independent -QLearning}

% example of putting in a trademark on first expansion
\newacronym[first={NVIDIA OpenSHMEM Library (NVSHMEM\texttrademark)}]{NVSHMEM}{NVSHMEM}{NVIDIA OpenSHMEM Library}

\newacronym{KTH}{KTH}{KTH Royal Institute of Technology}

\newacronym{CB}{CB}{Chess-Bot}
\newacronym{CE}{CE}{Chess Engine}
\newacronym{Eval}{Eval}{Evaluation Function}
\newacronym{Neg}{Negamax}{Negamax Algorithm}
\newacronym{MT}{MT}{Multi-Threading}
\newacronym{SP}{SP}{Sequential Processing}
\newacronym{GM}{GM}{Grand Master}
\newacronym{UCI}{UCI}{UCI protocol}
\newacronym{GUI}{GUI}{Graphical User Interface}
\newacronym{Am}{Am}{Amdahl's law}
\newacronym{BS}{BS}{Board State}
\newacronym{SIMD}{SIMD}{Single Instruction Multiple Data}
\newacronym{TT}{TT}{Transposition Tables}
\newacronym{Bb}{Bb}{Bitboards}
\newacronym{PST}{PSVT}{Piece Square Value Table}
\newacronym{ST}{ST}{Search Tree}
\newacronym{EF}{EF}{Evaluation Function}
\newacronym{ML}{ML}{Machine Learning}
\newacronym{SEF}{SEF}{Static Evaluation Function}
\newacronym{CNN}{CNN}{Convolutional Neural Networks}
\newacronym{SMP}{SMP}{Symmetric Multiprocessing}
\newacronym{LSMP}{Lazy SMP}{Lazy Symmetric Multiprocessing}
\newacronym{YBWC}{YBWC}{Young Brothers Wait Concept}
\newacronym{ER}{ER}{Elo Rating}
\newacronym{Sped}{Sped}{Speedup}
\newacronym{Scal}{Scal}{Scalability}
\newacronym{UML}{UML}{Unified Modelling Language}
\newacronym{IDE}{IDE}{Integrated Development Environment}
\newacronym{FOSS}{FOSS}{Free and Open Source Software}
\newacronym{CCRL}{CCRL}{Computer Chess Rating Lists}
```