COMP 4905 Honours Project:
Implementation of a Distributed Chess Engine

Carleton University

William Lee Croft
Supervising Professor: Dr. Jörg-R. Sack
April 19, 2013
Abstract

The game of chess is a popular choice for the study of distributed artificial intelligence and presents many interesting challenges. These challenges include finding effective ways to split up the work across processors, designing efficient communication protocols, finding effective ways to search the tree of gamestates, and even finding efficient ways to implement the game of chess itself. Numerous interesting ways of dealing with these challenges exist and there is always the possibility for discovery of new methods. In this project I have decided to tackle the implementation of a distributed chess engine by researching the various known methods and trying to piece together what I believe will make up a good distributed chess engine.
Acknowledgements

The work done in this project is heavily based researching and learning from algorithms, papers and work published by others. The following covers the major sources which were used to put this project together. A complete list of sources can be found in the References section at the end of the report.

Bitboards

An excellent description of nature and uses of bitboards is given by Peter Keller. This short instructional segment describes how to understand and make use of bitboards for a chess engine. Bitboards were first used in a context for chess programming by the team Kaissa of the Soviet Union.


Piece Move Generation

A detailed description on the implementation of a complete chess engine has been written by Stef Luijten. This was used as the basis for my implementation of sliding and non-sliding pieces as well as the generation and storage of moves.

http://www.sluijten.com/winglet/index.htm

Young Brothers Wait Concept

Details on the implementation of the YBWC for the splitting of work in a parallel search were derived from the paper “Transposition-Driven Scheduling in Parallel Two-Player State-Space Search” by Jan Steenhuisen. This splitting method was devised by Rainer Feldmann et al.

http://www.st.ewi.tudelft.nl/~renze/doc/MSc_2005_Steenhuisen.pdf
Zobrist Hashing

My understanding and implementation of Zobrist hashing was pieced together from a description by Adam Berent on Chess Bin as well as the information available on the Chess Programming Wikispaces page regarding Zobrist hashing. The hashing method was developed by Albert Zobrist.

http://chessprogramming.wikispaces.com/Zobrist+Hashing

Transposition Table

My implementation of the transposition table is based off an explanation given by Jonatan Pettersson. The replacement scheme I used was devised by Ken Thompson and Joe Condon; it is described on the Chess Programming Wikispaces page for transposition tables.

http://chessprogramming.wikispaces.com/Transposition+Table
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>6</td>
</tr>
<tr>
<td>Motivation</td>
<td>6</td>
</tr>
<tr>
<td>Problem Domain</td>
<td>6</td>
</tr>
<tr>
<td>Challenges</td>
<td>7</td>
</tr>
<tr>
<td>Implementation</td>
<td>7</td>
</tr>
<tr>
<td>Solution Domain</td>
<td>7</td>
</tr>
<tr>
<td>Details on the Implementation of the Game of Chess</td>
<td>8</td>
</tr>
<tr>
<td>Gamestate Heuristic Evaluation Function</td>
<td>11</td>
</tr>
<tr>
<td>Gamestate Tree Searching and Move Ordering</td>
<td>12</td>
</tr>
<tr>
<td>Parallelization and Splitting</td>
<td>15</td>
</tr>
<tr>
<td>Testing and Results</td>
<td>18</td>
</tr>
<tr>
<td>Searchable Depth</td>
<td>18</td>
</tr>
<tr>
<td>BT2450 ELO Test Suite</td>
<td>19</td>
</tr>
<tr>
<td>Conclusion</td>
<td>20</td>
</tr>
<tr>
<td>Potential Improvements</td>
<td>20</td>
</tr>
<tr>
<td>References</td>
<td>22</td>
</tr>
</tbody>
</table>
Tables and Figures

FIGURE 1 – Visual Representation of Bit Positions in a Bitboard.............. 10
TABLE 1 – Piece Values........................................................................ 11
TABLE 2 – Depth/Time Search Results.................................................. 18
TABLE 3 – Depth 5 BT2450 Results...................................................... 19
TABLE 4 – Depth 4 BT2450 Results...................................................... 19
Introduction

Motivation

Having taken courses on both artificial intelligence (AI) and parallel computing in the previous semester I have developed and interest in the concept of distributed AI. The focus in the AI course was on AIs which play strategy based games and I learned that most game-playing AIs are limited in their strength by how far ahead in a game they can look. An AI typically works by building and searching a tree of gamestates in order to look over all (or many depending on how it is built) possible paths which can be taken through the game and then selects the move which will lead it down the path that it has deemed to be the best. Unfortunately, for most games an average computer does not have the ability to run an AI which could look all the way to the end of the game so instead the AI is limited to only look a certain distance ahead. In the course I took on parallel computing I have learned how to address problems like this by using multiple computers in order to give a greater computing ability to a program.

As such I have found the idea of building a distributed AI very interesting. Although there are many games for which an AI would benefit from a distributed design, I found chess to be one of the more interesting choices. The game itself is interesting for the fact that, although it seems simple at a glance, there is quite a bit of complexity behind how you would want to implement an AI for it. Furthermore, it is a very popular topic for the study of AI and so there are quite a few interesting algorithms in existence about how to tackle different aspects of building a chess engine. Parallel splitting approaches such as PVS, EPVS, YBWC and ABDADA [3],[5],[6],[10],[11],[13] have been designed for this purpose as have many complementary techniques of move ordering [2],[9]. Numerous distributed chess engines have been built using these splitting methods and various combinations of move ordering techniques. One of the most famous examples is IBM’s engine, Deep Blue, which played against the grandmaster Garry Kasparov.

Problem Domain

Building a distributed chess engine has many options in terms of how this can be accomplished. First some specifics about the target machine should be determined. Many distributed chess engines are designed for a specific cluster (group of computers working together) so it is important to begin development with the target machine in mind. Information such as the number of computers in the cluster and the speed at which they can communicate is key. In some large clusters the topology is also important to the design of the program. The engine I have built is designed to run on the Lambda cluster at Carleton University which is a cluster of 16 quadcore machines running Linux. Since this is a relatively small cluster I have designed the program without consideration for topology specifics but under the assumption that communications will be a bottleneck since this cluster does not provide very rapid transfers.
such as some specialized clusters do. The program could likely run on other clusters although if
the number of machines were increased significantly beyond 16 then some modifications
would be required.

Challenges

Building a distributed chess engine presents numerous challenges, both in the general sense of
building a chess engine and also in the sense of building a distributed-specific engine. One of
the biggest challenges with building any kind of chess engine is the efficient management of
memory. The gamestate tree in chess is huge and so doing things in the most compact way
possible is generally preferable. Additionally, it is desirable to search this tree in a fashion that
is as efficient as possible. In the allowable time given to make a move in the game of chess only
a small portion of the tree can be searched. By searching more efficiently, a larger portion of
the tree may be searched.

When trying to make a distributed engine there are two main challenges to be dealt with. The
first is the splitting of the work and distribution of appropriate information. Since the
processors do not share the same memory, when work is to be assigned to a processor it must
be sent over to that processor along with any other required information. This means that
some method to split and distribute work must be determined. This leads to the second
challenge which is the communications overhead. On most distributed machines,
communications are very slow compared to computations. Due to this, very efficient
communications are needed.

Implementation

Solution Domain

In order to address the main challenges of building a distributed chess engine I have made the
following design decisions. For a compact design I have represented moves using unsigned
integers. The game board is represented using bitboards which is a slightly larger
representation than alternatives such as an array of chars but has a tradeoff of offering very
quick operations for applying changes to the board. In order to efficiently search the gamestate
tree I decided to use a minimax search with alpha-beta pruning assisted by move ordering from
a transposition table. In terms of distributed design I chose to use the Young Brothers Wait
Concept for splitting and distribution of work. Additionally I dedicated a single processor to act
as a communications intermediary in order to cut down on the communications overhead.
Details on the Implementation of the Game of Chess

The first step for building the AI is to build the game of chess itself in order to give the AI the ability to simulate the game. It must be mentioned that in the game of chess white will always make the first move, and in this implementation, white is located on the bottom of the board.

Board Representation
The game has been developed using bitboards to represent the game board. A bitboard representation makes use of the fact that a chess board has the same number of squares as there are bits in a 64 bit integer \(^\text{[7]}\). This allows for an integer to represent one entire aspect of the gameboard. The obvious choices are the 6 types of pieces in each color. As a result, using 12 bitboards, the entire gameboard can be represented. For example the bitboard representing the white pawns in their initial position at the start of the game would look like the following:

```
0000000000000000000000000000000000000000000000001111110000000000
```

This shows that the entire second rank is occupied by white pawns. In order to visually display the board, the bitboards would have to be converted, but for the engine, this is a very useful representation. At the expense of a bit of extra memory, much can be gained from adding on some extra bitboards. The main advantage of bitboards is the speed and ease with which important game operations can be done. In order to effectively make use of this there is also a need for bitboards to represent all black pieces, all white pieces and all game pieces (black and white). These 15 bitboards must be kept up to date with all moves in order to accurately represent the board. The reason that bitboards can perform game operations so quickly is that in order determine potential moves or make the moves all that needs to be done is to make some simple bitwise operations on the relevant bitboards. For example, when a pawn is going to move, the start and end locations are also represented using 64 bit integers. A new bitboard to represent the white pawns after one has moved can be produced by taking the original, ANDing it with the complement of the start location of the move and then ORing it with the end location of the move. Using the same bitboard from above for the white pawns, if a pawn were to move one space forward, the operations would look as follows:

Before: 0000000000000000000000000000000000000000000000001111110000000000
~& 0000000000000000000000000000000000000000000000000000000100000000 | 0000000000000000000000000000000000000000000000000000000000000000
After: 0000000000000000000000000000000000000000000000000000000000000000

Sliding and Non-Sliding Pieces
At this point it is important to note the difference between sliding and non-sliding pieces since they will be treated differently. A sliding piece is any piece that can move as far as the board configuration allows in one of the piece’s movement directions. The sliding pieces are the rooks, bishops and queen. The non-sliding pieces are all others which have more restricted
movement; these are the knight the king and the pawn (the pawn is not always considered a non-sliding piece but for simplicity it will be considered one here).

The movement and attacks of non-sliding pieces are much simpler to compute than those of sliding pieces for the fact that if one of the moves of a non-sliding piece is blocked then that move is simply not possible but has no influence over the rest of its moves. With sliding pieces however, if one of its moves is blocked, then all other moves beyond the blocked point in the same direction are blocked as well. The non-sliding pieces will have their moves and attacks computed by taking the bitboard that represents their own location and preforming a left or a right shift X number of bits in order to achieve a potential move location (the number required to shift will always be the same for each piece) and then preforming and bitwise AND on this new bitboard and the complement bitboard of pieces of its own colour. The reason for using the complement is to represent all empty spaces and spaces occupied by enemy pieces (these are the valid move and attack locations). By doing this for each potential move and combining them all with a bitwise OR a new bitboard can be produced which represents all possible movements and attacks for a non-sliding piece. In the case of pawns, a couple of extra checks are needed to account for the ability to move two spaces from the start location and also to check for en passant captures (when a pawn moves two spaces but if it had moved only one and stopped on a location where an enemy pawn could have captured then on the next turn the enemy pawn may still capture that pawn).

As mentioned before, sliding pieces are slightly more complex. Some work needs to be done beforehand in order to facilitate the operations for these pieces. In this implementation, many of these operations will rely on pre-computed attack and movement bitboards stored in lookup arrays. By spending a bit of time on startup, it is possible to generate and store bitboards which can represent the potential moves and attacks of every piece in every possible “significantly distinct” gamestate. The important thing to note here is what makes the gamestates significantly distinct. First of all, when simply computing pieces to attack, there is no need to identify what type of piece is being attacked, only whether the piece is enemy (able to attack) or ally (not able to attack). From this point it is conceivable that if only a single rank is to be observed (a horizontal row) it would be possible to generate all possible states. A two dimensional lookup array is used for this. The first dimension represents the 8 possible locations in which the attacker could be found, and the other dimension represents all possible states in which the other spaces of the rank could be configured. These two pieces of information are used as the indices and the data which is found represents the attack and movement pattern for the specified configuration. This information is very valuable because it can now be used to populate other lookup arrays which will later be drawn upon for all sliding operations.

The most obvious use of the single rank sliding attack array is to populate a sliding attack array for all ranks on the board by applying some simple bit shifting. In order to demonstrate this, the table below is included to show which space of the board each bit from the bitboard maps to.
It should be easy to see that by shifting by a multiple of 8, any other rank can be easily achieved. In the example below, the reference array can be applied to the 3rd rank by shifting everything left by 16 bits. All entries in the reference array will pertain to the 8 least significant bits since that is all that has been computed so the entries will look something like the following:

0000000000000000000000000000000000000000XXXXXXXX

To apply this to the 3rd rank a left shift of 16 will produce a result that looks like the following:

000000000000000000000000000000000000000XXXXXXXX0000000000000000

The same idea can be applied for sliding file (vertical column) and sliding diagonal movements and attacks though it is not quite as simple. In order to make the conversion to file and diagonal another intermediate lookup array is needed. These arrays are filled using “magic numbers” for the specified type of conversion. The computation of such magic numbers is not trivial and since there are well defined magic numbers meant for this use they were simply directly referenced for this project. Once all lookup arrays have been populated, the sliding attacks and movements can be found very quickly and easily by doing some simple bit shifting, passing in the corrected indices and applying some other quick bitwise operations to obtain valid movements.

Moves Representation
With all of the steps mentioned above, it is possible to obtain a bitboard which represents all possible movements and attacks of any piece on the board but this information is not yet in a useful form. This bitboard needs to be broken down into individual moves. This is necessary regardless of whether the information is to be used by a human or computer player. For an efficient storage of each move, a Move class is used to store the move as an integer and to get and set data on the move using member functions. Within the bits of the integer of the Move class, information is stored to represent the start and end locations of the move, any capture made, and any promotion made (pawn reaching the end of the board and being promoted to a higher piece).
On each player’s turn, all moves will be found and stored in a list of Moves. This is done as described above, however there are a couple of cases in which some additional details must be taken into account. With pawns, options must be provided for promotions when a pawn has reached the opposite end of the board. With kings, they are given the option to perform a castling move as long as the following conditions are clear: the king and the rook with which it will castle have not yet moved, and the passage of the king during the course of castling must not pass through any spaces which are under attack. In order to verify this, a helper function is used which check whether a specific space is under attack by a specified player. This function will also be used in other situations in order to verify checks and checkmates on the king.

**Gamestate Heuristic Evaluation Function**

In order for the AI to make effective decisions it must have some understanding of the state of the board. To do this it must evaluate the board and assign a number to represent the current value. In this implementation the number is represented in terms of the black player; if the number is negative then the board is in favor of the white player.

The primary factor used in board evaluation is material (the pieces possessed by each player). The point system chosen for this implementation is the Hans Berliner point system which values the pieces at:

<table>
<thead>
<tr>
<th>Piece</th>
<th>Basic Value</th>
<th>Centipawn Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pawn</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>Knight</td>
<td>3.2</td>
<td>320</td>
</tr>
<tr>
<td>Bishop</td>
<td>3.33</td>
<td>333</td>
</tr>
<tr>
<td>Rook</td>
<td>5.1</td>
<td>510</td>
</tr>
<tr>
<td>Queen</td>
<td>8.8</td>
<td>880</td>
</tr>
</tbody>
</table>

**TABLE 1 – Piece Values**

The basic value is converted to Centipawn value for use in the actual evaluation. This is done in order to accommodate other factors which will be taken into account which offer smaller point values. The other values are easier to represent as a percentage of a pawn. As such, making the pawn worth 100 makes the system easier to work with and avoids decimals. Additionally a small bonus is given for still having the pair of bishops since this offers a certain tactical advantage that is lost once one of the bishops is captured.

The next factor worth observing is the location of each piece. Pieces are more valuable in certain areas of the board and less valuable in others. For this reason each piece has a lookup array for every space in which it could be located. The array contains the value of the piece being in that space. This value is added (or subtracted) from the current total value. For example, knights are encouraged to stay near the center of the board since center locations are best suited for their mobility. Kings on the other hand are encouraged to stay near the corners.
on their own side. By placing higher values on a piece being in these desirable locations and lower or even negative values on the pieces being in less desirable areas it encourages the AI to select moves which will guide pieces towards the appropriate areas.

It is also important to take into account which pieces are under attack and which pieces are defending other pieces. \[^{[15]}\] Defending and attacking in this sense does not refer to an actual action taken, it just means that the piece is in a position which allows it to attack another piece or in which it can destroy the attacker of the piece it was defending. To compute values here each type of piece is assigned and attacking and defending value which is added (or subtracted) from the score for every piece which is being attacked or defended by said piece. Pawns are given the greatest value because if a pawn can take on the tasks of attacking or defending then this leaves the more valuable pieces open to take on more important tasks. Likewise if all major pieces are locked into place because they value their current attacking or defending positions too highly then mobility is hampered so the higher pieces are given a smaller bonus for these things.

Finally, points are awarded for maintain a pawn shield around the king. This is done by assigning extra points to the pawns directly adjacent to the king. \[^{[15]}\]

---

**Gamestate Tree Searching and Move Ordering**

**Minimax Search**

In a two-player of perfect information (all information about the current state of the game is visible) such as chess, the minimax searching algorithm typically works very well. Most AI gamestate tree searching algorithms work by starting at a node which represents the current gamestate and creating a branch to a child node for each possible move the AI can make from this state. Those child nodes will then all branch to their own children based on moves that can be made by the other player. Once again the new level of children will branch based on moves of the AI. This effectively represents all gamestates that can be achieved by the players taking turns making X moves where X is the depth to which the tree can be searched. In minimax, the algorithm assumes that the AI will always select the move which will lead to the highest valued end result and the opponent will always select the move will lead towards the lowest valued end result. The value in this sense refers to how much the board is worth for the AI, so a value of 0 means the game is tied and a negative value means the board is in favour of the opponent. Furthermore the algorithm assumes that the opponent has the same paying strength as the AI. In order to properly evaluate the worth of each move, the tree must first be expanded and then evaluation can be done from the bottom up. Bottom-up evaluation is required because what most important is the end result of the path chosen, not the intermediary values of the board. \[^{[6]}, [11]\]

**Alpha-Beta Pruning**

The effectiveness of minimax can be improved by using alpha-beta pruning. This is a technique in which an upper and lower bound are used along with the search in order to identify areas
where continuing the search is not useful and cut off the search in those areas. For example if a
max node has already evaluated some of its branches and has a current best move, it will not
select any subsequent branches which offer moves with lower values. Therefore each
subsequent search done on its children can be run with a lower bound of the current value of
the max node. This means that if any of those subsequent searches on the children enter a
state where it is guaranteed that their result will be lower than the lower bound, they are
simply cutoff right at that point. Doing this saves time and space that would have otherwise
been spent on a useless subsearch.

\[6\], \[11\]

Due to the high effectiveness of alpha-beta minimax on two-player games I selected this as the
searching method to use in this program. However, in order to get the most out of the pruning
a good move ordering must be applied. By using various techniques (more information on this
under the Potential Improvements section) the list of potential moves from any given
gamestate can be ordered in a way such that with a high likelihood the best moves will be
searched first. Unfortunately there is no method currently known which can enforce a perfect
move ordering however there are many good techniques that can be used regardless. The
reason that a good move ordering is desirable is that if a good move is searched first then a
very tight bound will be used for pruning the subsequent searches and thus a smaller portion of
the search tree must be explored.

**Transposition Table**

Although there were multiple techniques which I had wanted to include in this engine, due to
time constraints I was only able to implement one of them. The one which was implemented is
the transposition table. This is a table which keeps track of the results of past searches
(transpositions) in order to reuse past findings that would have otherwise been discarded after
their first use. The idea is that when a gamestate tree is being extensively searched, it is
inevitable that the same states will be encountered on different branches. This essentially
means that although a different path (sequence of moves) has been followed, multiple paths
have led to an identical state. In cases where this happens, rather than preforming the costly
action of redoing a search on the state, the previous result can simply be used again if it has
been kept. As such, the transposition table is a large table containing information regarding the
results of past searches.

\[2\], \[9\]

**Zobrist Hashing**

This sounds like a very simple idea but the implementation is not quite so simple. The first
problem is how to index the entries in the transposition table. This table will hold a large
number of transpositions and so iterating through them to look for a specific transposition is
not at all desirable. Thankfully there is a useful method which can be used called Zobrist
hashing. This is a way in which the gamestate can be made into a hash which can then be used
for indexing. In order to compute a Zobrist hash, a bit of setup is required. First an array of
pseudo-random 64 bit integers is required. This array is used to represent different elements of
the gamestate. The location of each piece is required as well as other important information
such as castling rights and the en passant square. Kings with varying castling rights could be considered as actually being different types of pieces for this purpose and the EP square can even be considered as a type of piece when used for Zobrist hashing since no other game piece will ever overlap with the EP square. Therefore the array could be comprised of a 2D array of 64x19 to represent each square by each piece type. This array will have each element filled with a pseudo-random 64 bit integer as mentioned.

[2], [17]

Having set up the array, the gamestate can now be hashed. The initial gamestate is found by starting with a representation as a 64 bit integer of all 0s. Then for each piece on the board, the representation is XORed with the pseudo-random integer from the correct index of the array. For example if a white pawn which we will consider to be piece type 1 is on the 9th square then the representation would be XORed with the entry at [9][1] of the array. Once this has been done for every piece on the board, including the EP square then the final representation is the hash of the current gamestate.

This may seem a bit extensive for finding a way to hash the gamestate but in actuality all of the hard work has already been done and finding the hash for each gamestate after this point is much simpler. The great benefit of Zobrist hashing is in the way that it is done through the XOR operation, which if applied again will result in the same state from before the operation. In other words, to apply the operation a second time is to undo it. Using this property, the gamestate hash can simply be updated rather than recomputed each time the game changes. All that must be done is to make changes to reflect where a piece no longer exists and to reflect a new place where a piece exists. This means that with usually 2 or 3 XOR operations the gamestate has can be updated after each move.

Next the hash must be made into an index. Ideally, the whole hash could be used as the index, but this would make the table much too large to be stored (2^64 entries). In order to index this hash into a smaller table a simple calculation can be used to get an index:

\[(\text{gamestate hash}) \mod (\text{table size})\]

or

\[\text{hash} \mod \text{size}\]

This allows for storage and lookup in constant time which is important since the whole point of this table is to speed things up.

**Collisions**

The next problem is collisions. A collision in this sense refers to two different items which are referred to in the same way. When using Zobrist hashing there are two types of potential collisions. A type 1 collision refers to multiple gamestates which have the same hash. Using a 64 bit integer as a hash means that there is a maximum of 2^64 possible different unique hash keys which can be made. This is a very large number but the number of valid gamestates in chess is much larger. When a collision of this type occurs, invalid data may be used without
really being able to identify when this has happened. This type of collision is not very common and is generally not addressed since the benefits of the transposition table outweigh any mishaps caused by type 1 collisions. The only way to decrease the likelihood of a type 1 collision is to increase the size of the hash key, but this places a higher strain on memory. [2], [17]

The other type of collision that can occur is the type 2 collision in which the derived index used for the transposition table collides. This type of collision is much more likely and is directly related to the size of the transposition table used. Again at the cost of more memory the collisions can be reduced. There are however other ways of dealing with this type of collision. Typically the entire hash is stored in the transposition table along with other information so after a lookup is done based on the index, the hash can be checked to ensure a collision (of type 2) did not occur. [2], [17]

Due to the high number of collision that will occur of type 2 it is desirable to have a method for dealing with them. This is where a replacement scheme comes into play. It will not be long before the transposition table will fill up and new results will be contending for space. Decisions will have to be made on which information to keep when collisions occur. There are different schemes that can be used which base the decisions off factors such as the depth of the transposition (how deeply the stored information had searched) and the age of the transposition (older transpositions are less likely to be reused). The scheme I implemented is the two-tier system which holds two transpositions at each index. One is age based and one is depth based. In other words, when a type 2 collision occurs, the depth based transposition is first checked, if the new transposition is deeper than the depth based entry is replaced. If it was not deeper, then the age based entry is replaced instead. This scheme attempts to balance the desire for more valuable transpositions with the desire for more relevant transpositions. [17]

Parallelization and Splitting

In order to effectively make use of other processors in a distributed machine a good splitting technique must be used. If the work is split up in a naïve fashion then there are risks for many detrimental factors including redundancy, excessive idleness and poor sharing of important information (bounds, known results, etc…). Additionally on a distributed machine it is essential to keep the communication overhead as low as possible since excessive communication can end up being the bottleneck of the engine.

The splitting method chosen for this project is the Young Brothers Wait Concept (YBWC). This method essentially considers any node in the search tree to be a candidate for splitting under the condition that at least one branch of the node has been fully explored. This is good in the sense that it means there will be very little time that any processor will spend idling. The first
branch (eldest brother) of any node must be searched sequentially and then all other branches (young brothers) can be searched in parallel.

[6], [15]

Once the first branch of the tree has been searched, there will likely be little to no time spent idling until the end of the search when the final branches are being completed. In YBWC the processor conducting the initial search on a node where a split is performed is considered the owner of that node. Any processor which is idle can take one of the split paths and conduct a subsearch on that portion and then return the result to the owner. Once a processor has completed a subsearch it once again becomes idle and looks for more work. As mentioned above, move ordering plays a very important part in searching. This method relies on the hope that the first branch searched will provide a very tight bound and thus reduce the size of the subsequent searches.

While the theory behind YBWC is very simple, the implementation tends to be less so. The communication in this project is done using MPI. Due to the fact that the application is not multithreaded it was not possible to dedicate threads on each processor for blocking communications. An alternative was to use non-blocking communications however this would have introduced many issues involving timing and buffer safety. For these reasons it was decided to dedicate one of the processors as a communications intermediary. In this way blocking communications could be employed with minimal (if any) wait time while processes blocked for communications. It was assumed that the sacrifice of one processor from the search would be offset by the time saved in having a more efficient communications method. Unfortunately due to time constraints it was not possible to conduct test in order to determine whether this assumption was valid.

Processor 0 is the processor on which the game is run and thus will always be the owner of the root of the tree. Processor 1 (henceforth referred to as the communicator) will never actively participate in the search but will instead always be listening for communications from other processors and will react appropriately when it receives messages. Processors 2 through 15 will all initially be idle and will participate in the search once split points have been reached by processor. From this point, the other processors will identify more splits points and processor 0 will continue to do so as well so there will be a consistent supply of work until the search is complete. In such a setup work could either be distributed via push, where the communicator is constantly trying to send out work, or by pull, where the other processors request work when they are idle. Due to the nature of YBWC creating so many split points, the pull method is preferable here since there will always be work to take until the whole search is complete. This means that as soon as a processor requests work, it will receive it. With that in mind it is best to allow processors to request work as soon as they have become idle and receive it immediately rather than waiting for the communicator to push work to them.

Another important aspect of the parallelization is the use of the transposition table. In keeping with the idea of having a communicator processor, I decided to store the transposition table only on the communicator. This follows the same reasoning as with the splitting of the work in
the sense that it will minimize the communications overhead. Other possible methods are to keep a complete copy of the table on every processor or to keep portions of the table on each processor. Both of those methods require the other processors to be constantly passing information among each other and in many cases it will have been information that was never used in the places it was sent. To minimize this I decided to have each processor simply tack its transposition information onto its results which are sent to the communicator anyway. The communicator will receive this information and store it. Then when sending out work, the communicator can locally lookup transpositions and if there is any information available it can send it along with the work that it is sending out. This greatly reduces the data going around to accommodate the use of the transposition table.

The majority of the communication is done using custom MPI structures designed to send the required information in the smallest format possible. A high level overview of the different types of communications used follows:

**Send search package**
When a split point is determined on any processor, the processor will send the current gamestate to the communicator. The communicator will reconstruct the game and identify which moves could be branched off from the state it was sent. This information will be stored for later access as work to distribute. Additionally, when a processor has reached a split point, this means that it has completed the first branch of the split node and therefore has some transposition information. This transposition is also sent in the search package for storage on the communicator.

**Request work**
When a processor is idle it will request work from the communicator. If the communicator has work available it will select the highest split point in storage (this ensures that significant work is distributed rather than shallow searches which would be quickly completed and result in more communications) and send the subsearch to the processor in a work package. Before actually sending the work, the communicator will check the transposition table for any information regarding the work to be sent. If there is an exact transposition of depth greater than or equal to the current search, it will store the result as a completed search and move on to the next piece of work to send out. If the transposition was not exact but offered tighter bounds or move ordering information then it will attach that information to the search package to be sent out. The processor will then receive the work and transposition information and conduct the subsearch, after which it will return the result to the communicator along with the new transposition information produced by the search. The communicator will store the transposition and will keep the returned result until the owner is ready to receive it.

**Request update**
After the completion of each child, a processor will request an update from the communicator for the split node on which it is currently working. The communicator will send back all information regarding completed and pending searches of the other child nodes. This is important in order to both ensure that all essential search information is returned to the owner...
as well as to ensure that the owner does not repeat work which may have been started by another processor. Once the owner has received all updates it will in turn send an update of its own status (including transposition information from its past search) back to the communicator. This is done to ensure that any work the owner may have competed does not get repeated by other processors and is also used to update the bounds if they have been tightened. Finally the communicator will send a final reply back to the owner to provide any transposition information if possible on the next search the owner is about to do.

Testing and Results

Searchable Depth

In running samples games where the computer would play against itself the following times are what were typically observed for searches of certain depths:

<table>
<thead>
<tr>
<th>Depth</th>
<th>Typical Search Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>A few seconds</td>
</tr>
<tr>
<td>4</td>
<td>20 – 90 seconds</td>
</tr>
<tr>
<td>5</td>
<td>3 – 10 minutes</td>
</tr>
</tbody>
</table>

**TABLE 2 – Depth/Time Search Results**

These results are quite poor in terms of the desirable capabilities of a chess engine. It is likely that the largest reason for not being able to search more deeply in reasonable time frames is due to a lack of move ordering. With the use of only the transposition table as an aid for move ordering there was simply not a good enough order achieved to have effective pruning on the search tree. A transposition table is indeed an excellent tool for move ordering but on its own it is not enough. Furthermore, the deeper the searches are, the more effective the transpositions will be, so the gains from at transposition table at such shallow depths are likely minimal. A more detailed look at other aspects that may be the cause of these results can be found in the section Potential Improvements.

**BT2450 ELO Test Suite**

The BT2450 test suite is a set of test positions which are given to a chess engine to solve. Based on the results provided by the engine, a rating can be found to determine the ELO of the engine. ELO is a relative skill-based ranking which is used in chess as well as some other games.
Of course since the ranking is relative (to other players), a test suite like this cannot truly determine the ELO of an engine but it is designed in a way to provide a good estimate. To calculate the ELO, 31 different preset boards are provided to the engine via FEN strings (a string-based format in which a gamestate can be specified, the chess engine must have the ability to read and load FEN strings to make use of this) and the time taken for the engine to solve each gamestate is measured. Any times which exceed 900 seconds are simply recorded as being 900 seconds since this is the upper limit. Any position that is solved incorrectly is also counted as being 900 seconds. After all positions have been solved, the average of the times is taken and is then subtracted from 2450. The resulting number is the ELO rating given to the engine. A partial view of the results follows; these tests were left incomplete because it became obvious that the score for each position would be 900.

<table>
<thead>
<tr>
<th>Depth 5</th>
<th>Position</th>
<th>Time taken</th>
<th>Correctness of Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>376 sec</td>
<td>Incorrect</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>77 sec</td>
<td>Incorrect</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>778 sec</td>
<td>Incorrect</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1297 sec</td>
<td>Correct</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>27 sec</td>
<td>Incorrect</td>
<td></td>
</tr>
</tbody>
</table>

TABLE 3 – Depth 5 BT2450 Results

<table>
<thead>
<tr>
<th>Depth 4</th>
<th>Position</th>
<th>Time taken</th>
<th>Correctness of Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>35 sec</td>
<td>Incorrect</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>7 sec</td>
<td>Incorrect</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>50 sec</td>
<td>Incorrect</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>64 sec</td>
<td>Incorrect</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>8 sec</td>
<td>Incorrect</td>
<td></td>
</tr>
</tbody>
</table>

TABLE 4 – Depth 4 BT2450 Results

The reasons for such results can be attributed to the same causes discussed in the previous testing section.
Conclusion

Potential Improvements

There are a great number of improvements that I would have liked to make on this project. The reasons for this are that much of what I was doing was all researched and learnt as I was going so I did not have a clear picture of how everything would fit together as I was building it. If I were to start over I would likely structure things differently. Furthermore just about every portion of this project ended up taking much more time and effort than I had anticipated. Due to this I was rather rushed by the end and could not implement some elements that I would have liked to.

One of the concerns that I have with the engine in its current state is that although I have tried to minimize the communications, they may still be too heavy. Given the time, I would have liked to closely examine the packages for redundant data and excessive variables types used (ints which could have been chars). In an extreme case, some variables could be assimilated into a single variable to later be re-extracted. For example, MPI does not have a boolean type for sending data so booleans must be sent as chars. Instead a single 8 bit char could be used to represent 8 booleans.

In addition to communications being too excessive I feel that local memory use may also be excessive. In similar ways to how the communications could be reduced I could also make the use of local memory more compact. This would open up possibilities to store more useful data.

The use of transposition-driven scheduling may have been a better choice for the scheduling of work. The system works in a way such that each processor is responsible for a certain portion of the work to be done. The division of these portions is up to the programmer. As such each processor will also hold a portion of the transposition table related to its work. When a processor encounters work which is not part of its own area, it will send it to the appropriate processor. This method has the benefit of reduced communications and local lookups for transpositions.

The heuristic function is also an area that I would have liked to improve upon. Since the heuristic function is the underlying logic which decides the value of a gamestate it is one of the most important parts of the engine. I feel that putting the effort in to improve upon this function could be very valuable. It may not pay off in terms of being able to search more deeply but it would help the engine pick better moves and thus play more strongly. The only problem is that improving in this area would likely be very time consuming and also very difficult to get a measure of success.
Another way to give a boost to the capabilities of the engine is to hook up databases for opening moves and end games. Openings and ends to games have commonly recurring scenarios which have been recorded for quick reference in order to save the time that would be spent searching them. Implementing the ability to hook up such databases is an easy way to improve the strength of the engine with known and tested knowledge.

In terms of end games it is also beneficial to actually adjust the strategy used by the AI. Due to the reduced number of pieces on the board and the increasing need to move in and make a check mate the midgame strategy is not as effective anymore. This would mean changing the way the heuristic function to see the board differently in end game situations.

As mentioned there are many move ordering techniques which are commonly used and I did not have time to implement all of the ones that I would have like to. These would likely provide the largest gains since I feel that move ordering is where my engine fell short the most. The other techniques I would have liked to use are:

**Static exchange evaluation**
This identifies capture moves and calculates the trade-off value in a series of successive captures. In other words you may capture a piece but that means that on the next turn your piece will be captured and so on. Moves with positive trade-off values are ranked by how much can be gained.

**Killer heuristic**
This technique makes use of cutoff values in sibling nodes at different gamestates. In other words if a different (but similar) state had a good move which produced a cutoff and that move is possible to make from the current state then there is a good chance that the move will also be good in the current state and so it should be ordered more highly to try to cause a cutoff sooner.

**Iterative deepening**
This is a technique that can be used after the others or as a last resort to achieve a better move ordering. It works by doing small searches at reduced depths (meaning they will be done very quickly) in order to guess based on the results obtained which moves might be the best ones and order them more highly. Obviously this will not always have great accuracy as deeper searches are required to see if a move is truly good or not but this helps to get hints at where to start.
References


13. Weill, Jean-Christophe. The ABDADA Distributed Minimax Search Algorithm. Université de Paris. [https://docs.google.com/viewer?a=v&q=cache:P8eQYUsbj2IJ:citeseerx.ist.psu.edu/viewdoc/download%3Fdoi%3D10.1.1.78.6388%26rep%3Drep1%26type%3Dpdf+ABDADA+chess&hl=en&gl=ca&pid=bl&srcid=ADGEESj22Vuqwl8OH_zCCDVEgkZhSKcHrpwvdyIRGyjrn2tRZ2S1nYZDgMvsHUTCg5Az9QkKhwRwPX9W_7Vvy1-pT9ki0m-2_XvRNHTBg0UKCa1DfOyc_F3KplOqTJWYHRkfbfnA&sig=AHLIEtbTzhGM9jGRAq34Bs8t4KWy7-zdYEw](https://docs.google.com/viewer?a=v&q=cache:P8eQYUsbj2IJ:citeseerx.ist.psu.edu/viewdoc/download%3Fdoi%3D10.1.1.78.6388%26rep%3Drep1%26type%3Dpdf+ABDADA+chess&hl=en&gl=ca&pid=bl&srcid=ADGEESj22Vuqwl8OH_zCCDVEgkZhSKcHrpwvdyIRGyjrn2tRZ2S1nYZDgMvsHUTCg5Az9QkKhwRwPX9W_7Vvy1-pT9ki0m-2_XvRNHTBg0UKCa1DfOyc_F3KplOqTJWYHRkfbfnA&sig=AHLIEtbTzhGM9jGRAq34Bs8t4KWy7-zdYEw)
14 Information Unavailable

15 Information Unavailable
http://chessprogramming.wikispaces.com/Evaluation

16 Information Unavailable
http://www.economypoint.org/b/bt2450.html

17 Information Unavailable