Parallel Chess Searching and Bitboards

David Ravn Rasmussen

August 2004

Abstract

In order to make chess programs faster and thus stronger, the two approaches of parallelizing the search and of using clever data structures have been successful in the past. In this project it is examined how the use of a specific data structure called a bitboard affects the performance of parallel search. For this, a realistic bitboard chess program with important modern enhancements is implemented, and several experiments are done to evaluate the performance of the implementation. A maximum speedup of 9.2 on 22 processors is achieved in one experiment and a maximum speedup of 9.4 on 12 processors is achieved in another experiment. These results indicate that bitboards are a realistic choice of data structure in a parallel chess program, although the considerable difference in the two results suggests that more research could be done to clarify what factors affect such an implementation, and how.

Contents

Chapter 1

Introduction

Since the early 1950s when Claude E. Shannon and Alan M. Turing wrote the first chess programs based on the idea of game tree searching, much research has been done to make such programs better and stronger. Early on, it was understood that search algorithmic enhancements were not enough to make a strong program. Raw speed was a major strength factor as well. Apart from using specialized hardware to gain speed, the two main solutions to this problem have been parallelism and specialized data structures.

An idea for a particular data structure to represent information about the chess game state comes from the two observations that many computers work fast with 64-bit integers and that the chess board has exactly 64 squares. This data structure, called a bitboard, is a 64-bit integer holding boolean information about the chess board. Much more than one bitboard is needed to represent the full game state, however. In fact, this way of representing the game state requires more space than most other representations, but speed is more important than space in this case. Even though the idea of bitboards is not new, it is relevant now more than ever because 64-bit computers (which work fast with 64-bit integers) are becoming more and more common.

In this project, we try to examine how well bitboards are suited for parallel chess programs. While the space requirements of the representation of the game state are usually not important for sequential programs, they are important for parallel chess programs, because we have to transfer the entire game state between different processors frequently. It is unclear how the use bitboards will affect the performance of a parallel chess program.

We implement a parallel bitboard chess program using most known modern enhancements and techniques to get a realistic test case. We perform several experiments with the program to get some useful statistics giving us insights into parallel chess searching in general and into the performance of the implemented bitboard program in particular.

Our implementation achieves acceptable speedup in the experiments performed, although systematic fluctuations seem to affect the performance. We also get differing results in the various experiments we perform, indicating that memory bandwidth is an issue. The experiments also show us that while the parallelization makes the search speed larger as more processors are used, it also inherently makes the amount of information to search considerably larger, simultaneously.

After this introduction, Chapter 2 presents the rules and terminology of the game of chess, which is necessary to discuss more complicated issues later. In Chapter 3 we describe the fundamental elements of game tree searching, from the plain minimax algorithm to a modern formulation of the alpha-beta search algorithm, and Chapter 4 continues by explaining the most important algorithmic improvements to the alpha-beta algorithm, as used in practice by modern chess programs. Chapter 5 discusses the problem of data structures in chess programs and presents the solution we are using in our implementation: bitboards. In Chapter 6 we discuss the problems of parallelizing the alpha-beta algorithm, and describe the solutions we have implemented. Chapter 7 describes the experiments we have performed and discusses the results of these experiments. Chapter 8 contains our conclusions and our suggestions for further work. Appendix A contains the input positions and output data of our experiments and Appendix B contains the source code of our implementation.

Throughout this paper, it is assumed that the reader has good knowledge of data structures, algorithms, algorithmic complexity and parallelism.

I would like to thank Povl Ole Haarlev Olsen for helping with scripts for automating the process of gathering and processing data from experiments, and also for proofreading and making useful comments. I would also like to thank Ditte Marie Johansen and Silas Frederik Johansen for proofreading and for useful comments. Finally, I would like to thank my supervisor Professor Jens Clausen for helping me keep focus.

Chapter 2

The Game Of Chess

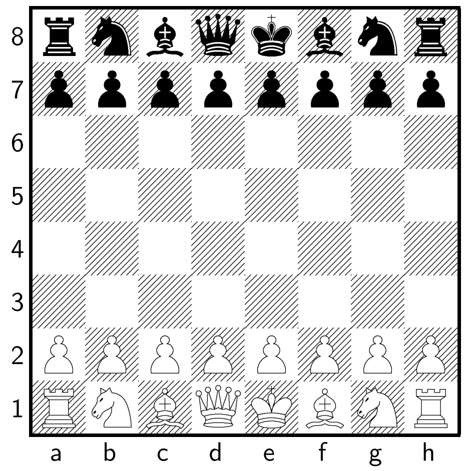

Figure 2.1: The initial board position in chess.

Chess is a board game played by two players called "white" and "black". The players alternately make a move, but white always makes the first move. It is a game of full information: there are no hidden or random elements; the players know the full state of the game at all times. In the following, we will briefly describe the rules of the game [27].

2.1 The Board

The game is played on a square board that is subdivided into 64 small squares. The 64 squares are arranged in an 8 by 8 grid. The vertical columns of the board are known as “files”. These are denoted by the letters a–h, from left to right . The horizontal rows are known as “ranks”. These are denoted by the numbers 1–8, from bottom to top. A square is denoted by the letter of the file it is on, followed by the number of the rank it is on. For example, the square in the lower left corner of Figure 2.1 is called “a1”. The diagonal lines are known as “diagonals”. To make it easier to see and think about the diagonals, the squares of the board have alternating colors called “light” and “dark”, in such a way that any diagonal consists of squares of only a single color, light or dark. As a fixed reference, the color of the square a1, and hence of the diagonal it is on, is always dark. The rest of the squares are colored accordingly.

2.2 The Armies

At the start of the game, both players have an army of 16 men. As shown in Figure 2.1, which shows the starting position of any chess game, white’s army starts at the 1st and 2nd rank while black’s army starts at the 7th and 8th rank. The men on the 2nd and 7th rank are the Pawns. On the 1st and 8th rank are, from the a-file to the h-file: Rook, Knight, Bishop, Queen, King, Bishop, Knight and Rook. That is, each player initially has 8 Pawns, 2 Knights, 2 Bishops, 2 Rooks, 1 Queen and 1 King. Most of the time, all these are referred to collectively as “pieces”, but sometimes a distinction is made between Pawns and pieces (non-Pawns).

2.3 The Moves

When it is a player’s turn to move (he is “on the move”), he chooses a piece of his own color. For each kind of piece, there is a predefined pattern of squares that the piece can move to. These patterns will be described later. A piece

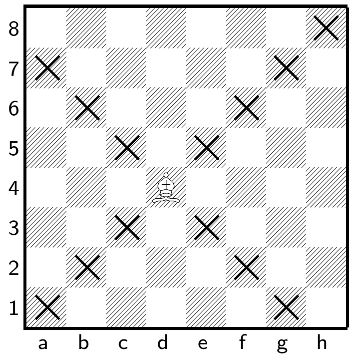

Figure 2.2: The Bishop.

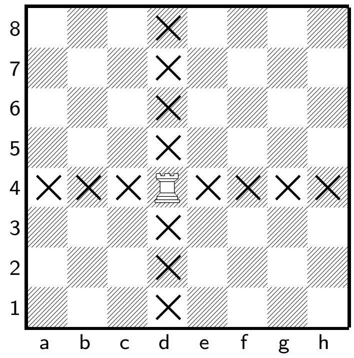

Figure 2.3: The Rook.

can only move to a square if the square is empty or is currently holding a piece of opposite color. Moving to a square with a piece of opposite color is called a capture, and the captured piece is removed from the board. A piece cannot capture another piece of the same color. The scope of all pieces is limited by the edge of the board. That is, no piece can go off one end of the board and appear at the other. No piece can move trough other pieces (except the Knight as described below). Thus, the scope of a piece is limited by all other pieces blocking it. A square is said to be attacked by a piece if the piece can move to the square. The Pawn, the Knight and the King all have special moves that will be described below. A move by one player is called a half move. A (half) move by both players is collectively a (full) move.

2.3.1 The Bishop

The Bishop moves along diagonals and thus always stays on squares of the same color. See Figure 2.2.

2.3.2 The Rook

The Rook moves along files and ranks. See Figure 2.3.

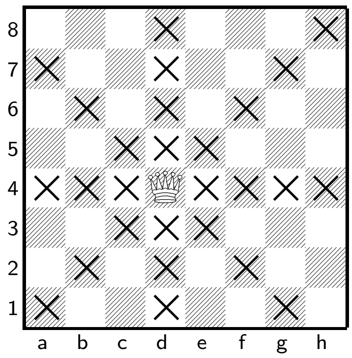

Figure 2.4: The Queen.

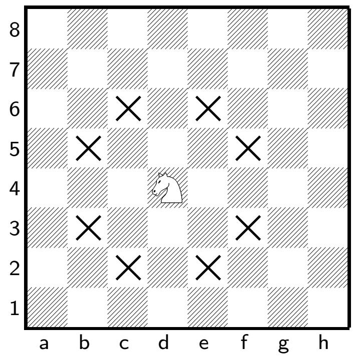

Figure 2.5: The Knight.

2.3.3 The Queen

The Queen moves like both the Bishop and the Rook: along diagonals and along files and ranks. See Figure 2.4.

2.3.4 The Knight

The Knight is the only piece that can jump over other pieces. It moves two squares along a rank and then one square along a file, or two squares along a file and then one square along a rank. It always moves from a light square to dark square or vice versa. See Figure 2.5.

2.3.5 The King

The King can move to any immediately adjacent square. See Figure 2.6.

When a player's King is on a square that is attacked by an opponent piece, the player (and his King) is said to be in check. It is illegal for a player to make a move that leaves his own King in check. Furthermore, the player who is in check must immediately make a move so that he is no longer in check. That is, either evade the check by moving his king, capture the checking piece of the opponent or block the check by moving a piece in between the king and the checking piece. If a player cannot make such a

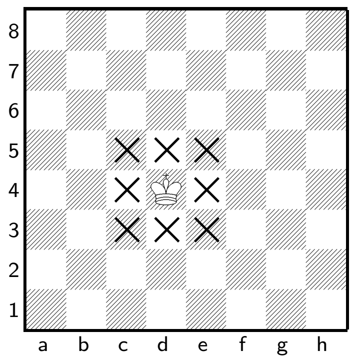

Figure 2.6: The King.

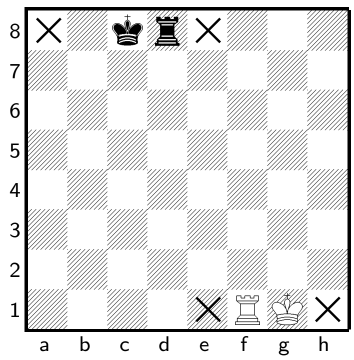

Figure 2.7: White has castled kingside. Black has castled queenside.

move, he loses, as described later.

The King has a special move: the castling. Castling involves moving both the King and a Rook. For castling to be legal, neither the King, nor the Rook in question may have made any moves earlier in the game. Also, the squares between the King and the Rook must be empty. Castling with the Rook on the h-file is called kingside castling. Castling with the Rook on the a-file is called queenside castling. In both cases, castling is performed as follows: the King is moved two squares in the direction of the Rook, and the Rook is placed beside the King, but at the opposite side. See Figure 2.7. In this diagram, the marks show the position of the pieces before castling; since neither the King or the Rook must have made any moves, they are placed as in the initial position.

Castling cannot take place if the King is currently in check, if the King moves through check when castling, or if the King would be in check on its destination square. Example: As long as any of the squares e1, d1 or c1 are attacked by black, white cannot castle queenside.

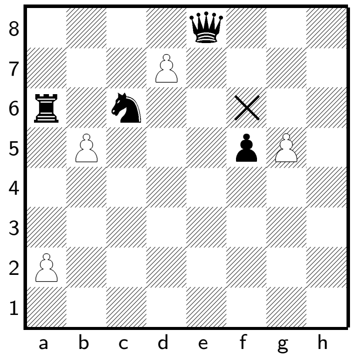

Figure 2.8: The Pawn has many special moves.

2.3.6 The Pawn

The basic move of the Pawn is simple: it moves "forward" one square. White Pawns move from the 2nd rank to the 8th rank. Black Pawns move from the 7th rank to the 1st rank. A Pawn can never go backwards. In Figure 2.8 white's Pawn on g5 can go to g6 and black's Pawn on f5 can go to f4. If the Pawn has never moved before (and thus is on the 2nd or 7th rank), it can move forward two squares if it is not blocked by any pieces. In Figure 2.8 white's Pawn on a2 can go to either a3 or a4. These two moves, moving forward one square and moving forward two squares, can only be made as non-capture moves. The Pawn's capture moves are different: it captures one square diagonally forward. In Figure 2.8 white's Pawn on b5 can go to b6, but it can also capture on either a6 or c6.

When a Pawn reaches the 1st or 8th rank, it is promoted to a piece of its own color, except a Pawn or a King. The player making the pawn move chooses what piece to promote to. In Figure 2.8 white's d7 Pawn can either promote on d8 or on e8. In the latter case white will also capture black's Queen.

Finally, the Pawn has a special move known as the "en passant" capture. This option arises when:

- The last move of the opponent was to move a Pawn two squares forward from its initial square

- In doing so, this Pawn passed over a square that is attacked by an enemy Pawn

If this happens, the player on the move can choose to capture the enemy Pawn as if it had just moved one square forward instead of two. The option to capture en passant is only available at the move immediately following the two-squared Pawn move. In Figure 2.8, if black's last move was to move his f7 Pawn to f5, white now has the option of moving his Pawn to f6 capturing the f5 Pawn, just as if the f5 Pawn was on f6.

2.4 The End

A chess game can end in several ways:

Checkmate
: When a player's King is in check, and the player has no legal moves (capturing the checking piece, blocking the checking piece or evading the checking piece), the player is said to be checkmated. The player that checkmates his opponent wins the game.

Stalemate
: When a player's King is not in check, but the player has no legal moves, the situation is called a stalemate. The game is a draw.

Draw by Repetition
: If a position occurs three times in a game (with the same player on the move), the game is a draw.

Draw by the Fifty Moves Rule
: If the last 50 moves (100 half moves) have been made without any captures or Pawn moves, the game is a draw.

Figure 2.9 shows a more involved example. White has checkmated black. Blacks King is in check because it is attacked by white's Bishop on d8. Black

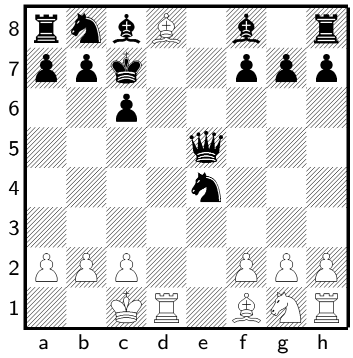

Figure 2.9: White has checkmated black.

has no pieces that can capture the d8 Bishop. All possible squares of escape for black's King are attacked by the d8 Bishop and the d1 Rook.

Chapter 3

Game Trees and Searching

In this chapter we will explain the concept of a game tree and describe the basic algorithms used by computers to search such game trees to find the optimal line of play.

3.1 Game Trees

For any two-player game of full information like chess, there is a so-called game tree which is a collection of all the possible positions (or states) of the game. The structure of this collection is, as suggested by the name, a mathematical tree. The root node of this tree is the initial position of the game, and the children of a node are the positions that follow directly from the node by performing the legal moves from this position. The leaves of the tree are the terminal positions of the game. Figure 3.1 shows an outline of the chess game tree.

The number of legal moves in a position or node of the game tree, is known as the "branching factor" of that node. For the entire tree, the average branching factor can be calculated or estimated.

We define the depth of a node to be the length of the path from the root node to the node in question. In other words, the depth of a node is the number of moves from the root node to this node. The unit of depth, one half move, is called a ply.

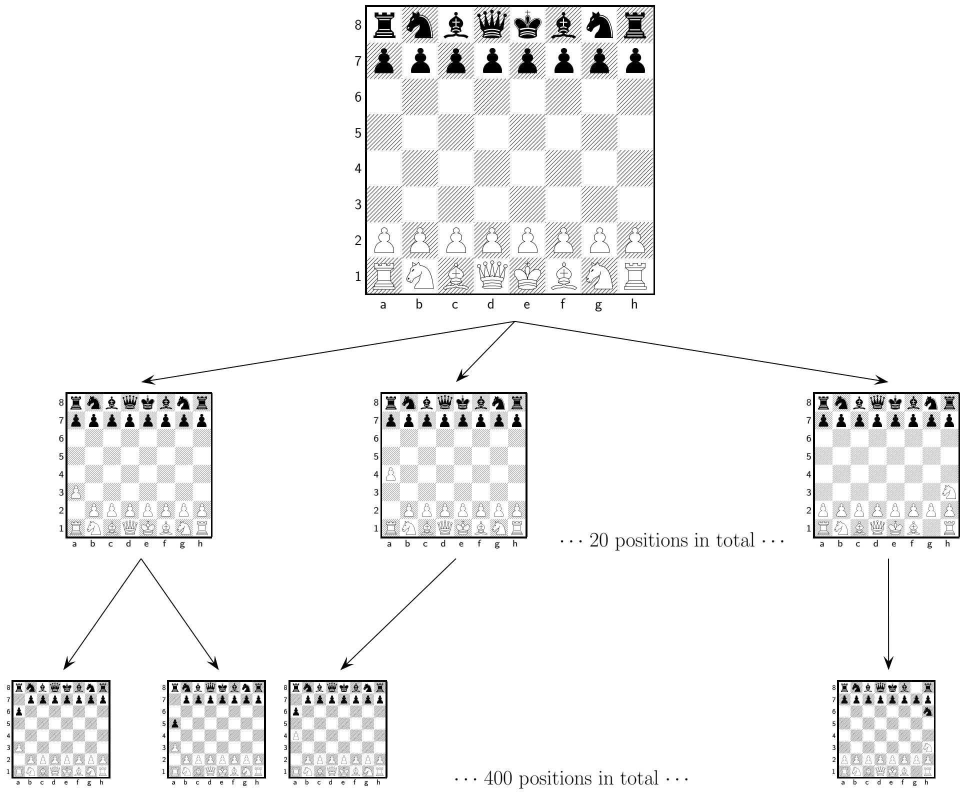

Figure 3.1: An outline of the chess game tree.

For simple games like "tic-tac-toe", the game tree is relatively small because of the relatively small branching factor (9 at the most) and because all games are short: the maximum depth is 9. Solving a game like "tic-tac-toe" (that is, finding the optimal line of play) is easy because the entire game tree can be considered.

For chess, the case is not so easy. The average branching factor of chess has been estimated to be about 35 [9]. It can vary wildly, though, depending on the position. In positions where a player is in check, there are usually only few legal moves. In a reasonably dynamic middlegame position, there may be 40–80 legal moves. In artificial positions with lots of Queens on the board, there may be more than 200 legal moves.

Also, a game of chess can be much longer than a game of "tic-tac-toe". A typical game of chess is maybe 30–80 moves (60–160 half moves) long. The longest real game ever recorded is 269 moves long. The longest possible game of chess is about 6000 moves long [26]. That means that some of the branches of the chess game tree are about 12000 nodes deep. The reason that a game of chess cannot continue forever is because of the rules of draw by repetition and draw by the 50 moves rule.

The bottom-line is this: the chess game tree is huge, and it is utterly infeasible to solve chess by considering all of it, no matter how many present-day computers we use. Even when considering the solution to interior positions deep within the chess game tree, we can only consider a small local subtree.

Therefore, instead of considering the entire game tree or the complete subtree of a position, we use heuristic search algorithms to search part of the subtree to obtain an approximate solution.

3.2 Search Algorithms

Below, we will describe the basic search algorithms used in game tree searching.

3.2.1 Minimax

The fundamental algorithm used for searching a subtree of a node to obtain an approximate solution, is called the minimax algorithm. The minimax algorithm works by minimizing the maximum damage that the opponent can do [24, 21, 23].

Imagine a position in which white is on the move. White will list all his legal moves or more precisely all the child positions immediately resulting from the current position by performing these legal moves, one by one. All of these child positions are assigned a value of either loss for white (-1), draw (0) or win for white ($+1$). White chooses the position with the **maximum** value. Of course, the "magic" here is how the value of each child position is determined.

In a child position, C, in which black is on the move, the value is determined as follows: If the position is a terminal game position (mate or draw), the value is immediately known. If it is a non-terminal position, black lists the child positions following from his legal moves, in the same way that white did before. These child positions are assigned one of the above values. Black then chooses the child position that has the **minimum** value. The value determined for the position C is this minimum value.

The values of these new child positions, in which white is again on the move, are of course determined in the same way as white did before. Thus, the minimax algorithm is recursive.

As described above, it would seem that the minimax algorithm deals with one level of the tree at a time, assigning values to all child positions of a position, before looking at any other positions. But because the value of a position cannot be determined until we have determined the value of its children (and so on recursively), minimax traverses the tree in a depth-first fashion. The first positions that are actually given a value, are the terminal positions at the leafs of the tree. These values are then propagated back trough the tree, minimizing or maximizing at each level.

Figure 3.2 shows an example of how the value of the root position is determined by the minimax algorithm. In this example, we have used more than

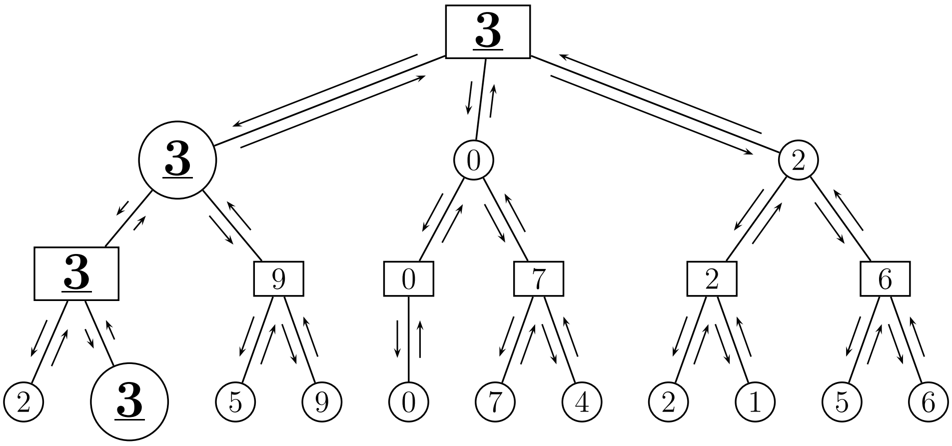

Figure 3.2: A minimax example.

```
score minimax(position)
{
  if (isTerminal(position))
    return (valueOf(position));

  if (turn == white)
    best = -infinity;
  else
    best = infinity;

  children = getChildren(position);
  while (not children.empty())
  {
    child = children.removeOne();

    value = minimax(child);

    if (turn == white and value > best)
      best = value;
    else if (turn == black and value < best)
      best = value;
  }
  return best;
}
```

Figure 3.3: The minimax algorithm.

just -1, 0 and 1 for the values. This is because it makes the example clearer and also because it is closer to the practical case where we use a so-called evaluation function, as described later. The boxed nodes are maximizing nodes (white's turn), and the circled nodes are minimizing nodes (black's turn). The arrows show how the algorithm traverses the tree. The enlarged nodes are the nodes on the so-called principal variation. They represent the optimal line of play for both players.

Figure 3.3 shows the pseudo-code for the minimax algorithm.

3.2.2 Negamax

To simplify the minimax algorithm, we can modify it so that it always maximizes regardless of which player is on the move. We just have to make sure that the values we are working with are always relative to the player on the move. For this, two changes are needed: The `valueOf()` function no longer returns absolute values, but relative ones. The value -1 now means "loss for the side on the move", 0 means draw, $+1$ means "win for the side on the move". Also, when a value is returned from a child node, it is negated. This formulation of the minimax algorithm is called negamax. Figure 3.4 shows the negamax algorithm. Note that these two algorithms are identical in that they visit exactly the same nodes and return the same answer. It is just a practical simplification that will make later modifications less difficult.

3.2.3 Evaluation

As explained earlier, the complete subtree of a chess position may be (and most often is) too large to search exhaustively given typical time and resource constraints. Therefore, instead of searching every branch of the tree until a terminal position is met, we only search a branch until a certain depth is reached. When a position at this depth is reached, we do not really know which player will win, unless it happens to be a terminal position. So we use a so-called heuristic evaluation function to obtain an approximate value. The greater the value, the more likely the player on the move is to win. Of course, writing such an evaluation function is not easy. It is domain-specific

```
score negamax(position)
{
  if (isTerminal(position))
    return (valueOf(position));

  best = -infinity;

  children = getChildren(position);
  while (not children.empty())
  {
    child = children.removeOne();

    value = -negamax(child);

    if (value > best)
      best = value;
  }
  return best;
}
```

Figure 3.4: The negamax algorithm.

```
score negamax(position,depth)
{
  if (isTerminal(position))
    return valueOf(position);

  if (depth == 0)
    return evaluate(position);

  best = -infinity;

  children = getChildren(position);
  while (not children.empty())
  {
    child = children.removeOne();

    value = -negamax(child, depth - 1);

    if (value > best)
      best = value;
  }
  return best;
}
```

Figure 3.5: The negamax algorithm with evaluation.

and usually cannot be used for evaluating positions of any other games. For domains complex enough to be interesting, it is usually impossible to write a perfect evaluation function that will always give a consistent and correct answer. After all, if we had such a function, we would not need to search at all. We could just evaluate the children of the current position and choose the best one.

A lot of the heuristic knowledge in a chess program lies in the evaluation function. It typically takes into account such things as material balance, mobility, control of important squares, development of pieces, pawn structure and king safety. The most important of these is material balance.

Figure 3.5 shows the negamax algorithm with depth limitation and evaluation.

3.2.4 Alpha-Beta Search

Studying the minimax algorithm, it is realized that it examines a lot of positions within the tree, that have no effect on the outcome of the search. These are positions that are so good for one player, that the opponent will never allow them to be reached. The alpha-beta algorithm is a simple but important modification of the minimax algorithm that prunes these positions [18].

Imagine this scenario: Examining the position at hand, player A has already found one good move, m_1, that wins the opponent Queen. The player now continues to examine another of his possible moves, m_2. Among player B's possible replies to this move, there is one move, r, that will force player A to only win an opponent Rook. As soon as this opponent reply is found, player A knows that examining the rest of the opponent replies resulting from m_2, is futile. Because m_2 will at most win a Rook, whereas m_1 will win a Queen. The move r is called a refutation of m_2.

This idea can be applied recursively throughout the tree search. The algorithm just has to keep track of two values at every node: a lower and an upper bound. The lower bound designates how good a position the player on the move can at least force, from the moves examined so far. The upper

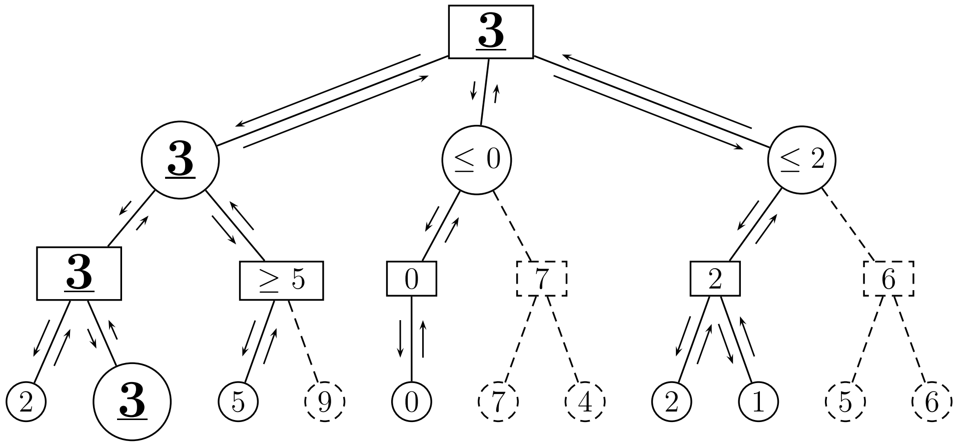

Figure 3.6: An example of alpha-beta pruning.

bound designates how good a position the opponent will at most allow. The lower and upper bounds are called *alpha* and *beta* respectively.

Figure 3.6 shows an example of how the alpha-beta algorithm works. It is the same tree as in the minimax example given earlier in Figure 3.2. As shown, we do not know the precise value of all nodes, just a bound on the value. But it is enough to make the correct decision. The nodes that are pruned are shown in dashed style.

The negamax formulation of the alpha-beta algorithm is shown in Figure 3.7.

As can be seen from Figure 3.7, the benefit of the alpha-beta algorithm is the early return that happens when the value becomes greater than or equal to *beta*. The rest of the children are pruned. It is obvious then, that if a refuting move exists at a node, we would like to find it as early as possible. If it is the first child searched, we can prune all the other children. If it is the last child searched, we have examined no fewer nodes than the plain minimax algorithm would. On the other hand, in the best case scenario where a refuting move is always searched first if one exists, pruning saves a lot of time. Searching an artificial tree of fixed branching factor W and depth D with the minimax algorithm takes $\mathcal{O}(W^D)$ time. Searching the same tree with the alpha-beta algorithm only takes $\mathcal{O}(W^{\frac{D}{2}})$ time in the best case, meaning

```
score alphabeta(position,alpha,beta,depth)
{
  if (isTerminal(position))
    return valueOf(position);

  if (depth == 0)
    return evaluate(position);

  children = getChildren(position);
  while (not children.empty())
  {
    child = children.removeOne();

    value = -alphabeta(child, -beta, -alpha, depth - 1);

    if (value > alpha)
    {
      if (value >= beta)
        return beta;
      alpha = value;
    }
  }
  return alpha;
}
```

Figure 3.7: The alpha-beta algorithm.

an exponential speedup [13]. In other words, the alpha-beta algorithm takes only in the order of the square root of the time that the minimax algorithm takes. Or put another way: given the same time, the alpha-beta algorithm searches twice as deep as the minimax algorithm, a vast improvement. Thus, the order in which moves are searched is very important, contrary to plain minimax. Heuristics are used to order the moves at each node so that good moves are generally tried before bad ones. But we do not have perfect move ordering heuristics. If we did, we could just order the moves and pick the best one, instead of searching.

Chapter 4

Search Algorithm Improvements

In this chapter we will describe some of the most important improvements of the plain alpha-beta algorithm. They are an important part of a realistic chess program, and they affect the practicalities of the parallelization described later.

4.1 Aspiration Windows

When the alpha-beta search function is called from the root position, the bounds *alpha* and *beta* given as arguments are $-\infty$ and ∞, respectively. Or more practically, *alpha* is given a value less than the lowest possible value and *beta* is given a value greater than the highest possible value. This is because we do not know the true minimax value of the position at this point.

Studying Figure 3.7, though, it is clear that when the range between *alpha* and *beta* (the so-called alpha-beta window) is small, more cutoffs (prunings) will occur, all else being equal. If *beta* is "small", more moves will have a value larger than *beta*, causing a cutoff. Also, a "large" *alpha* will mean a small *beta* when searching the children nodes. Naturally, this small *beta* value will also cause more cutoffs.

The basic aspiration window idea [20] is to call alpha-beta with an initial smaller window than $(-\infty, +\infty)$ at the root position. Basically, we "lie" to the children nodes below and pretend that both players have already found a reasonably good move, and that we are only interested in moves that are better. If this bluff works and the true minimax value of the root position is within the window given, we will get this value back from the search. We will have pruned more nodes than we would have with a full windowed search, though. That is, we have found the true minimax value of the position, but we have searched a smaller tree.

If the true minimax value of the position is outside the window given, the search will return *alpha* or *beta* depending on whether the true value is below *alpha* or above *beta*. In this case, we re-search the position but with a new window. If the value returned was *alpha*, we know that the true minimax value is at most *alpha*, so the new window becomes $(-\infty, alpha)$. If the value returned was *beta*, we know that the true value was at least *beta*, so the new window becomes $(beta, +\infty)$.

If we choose the aspiration window to be too small, we will have too many re-searches, searching more nodes than necessary. Choosing a reasonable aspiration window around the expected value of the search, will give a net gain by searching fewer nodes overall.

4.2 Transposition/Refutation Tables

Essential to a modern implementation of the alpha-beta algorithm is the transposition/refutation table [22]. The idea is this: whenever we have searched a node, we store the result of the search in a hash table. Specifically, we store the best move of the node (if the score is above alpha, because if it is not, we do not really know which move is best), and as much information about the value as we know. Because we use the alpha-beta algorithm, we do not always have the exact value of the node. Most often, we only know a bound on the real value: that it is at most *alpha*, or at least *beta*. So in the table we store this value (exact value or bound), and a flag telling whether the stored value is an upper bound, an exact value, or a lower bound. We also store the depth of the search that gave us the information we store.

Whenever the recursive search encounters a node, instead of just doing a recursive search of all the legal moves, it probes the hash tables first to see if we have any known information about this node. If so, we check whether the stored information comes from a search deep enough to replace the current search. If the search finds useful information from a deep enough search in the hash table, it can sometimes return an answer immediately without performing an expensive recursive search. This happens if the hash table has an exact value, or if the bound stored is good enough. That is, if an upper bound is stored and it is *alpha* or lower, or if a lower bound is stored and it is *beta* or higher.

By "deep enough" we mean this: If the stored information comes from a 2-ply search, it is not good enough to replace a 7-ply search. On the other hand, we can get even better information than we asked for. If the search is asked to perform a 7-ply search, and it finds information in the hash table from an 11-ply search, the information from the table is better than the information that we would have gotten from just performing the 7-ply search.

The utility of this table is two-fold: We can sometimes avoid expensive searches of positions that have been searched before. And in the cases where the search cannot be avoided, we have a good guess about the best move of the position, which will improve move ordering, as described later. Positions can occur more than once in the search for several reasons. First of all transpositions of moves can lead to the same position. If position P leads to position Q by the move sequence $a \rightarrow b \rightarrow c$, most often P will also lead to Q by the move sequence $c \rightarrow b \rightarrow a$. Secondly, performing re-searches of positions is an important part of several algorithmic enhancements described later. Doing a re-search of a position will be cheap if information about most of this subtree is already in the hash table.

4.2.1 Zobrist Hashing

A central question about the hash table is which hash function to use for our purpose. In [25], Zobrist describes a general way of hashing for the purpose game playing programs. In chess, this can be applied as follows: For each of the 64 squares, we generate a 64-bit random number for each kind of piece that can occupy the square. There are six different pieces in two different colors, so we generate $2 \cdot 6 \cdot 64 = 768$ random numbers. Of course, there are more things defining the position that we also have to put into the hashing scheme: castling rights and en passant square. We generate random 64-bit numbers for these too, one for each state that these can be in. The hash key of a position is then defined as the XORing of all the 64-bit random numbers corresponding to the state of each square, and of the numbers corresponding the the state of the castling rights and the en passant square. The resulting 64-bit number is the hash key for the chess position. Of course, all the 64-bit random numbers are pre-computed, and the hash key of the position is being incrementally updated when moves are performed. This is simple: We just XOR out the 64-bit number for the piece at the source square, XOR in the 64-bit number for this piece at the destination square, and XOR out any captured piece on the destination square. Special moves like castling, en passant and promotion are dealt with trivially. We also have to account for the side to move. We do this by having two separate hash tables, one for positions where white is on the move, and one for positions where black is on the move.

The hash tables are implemented as simple arrays of entries holding the stored information and the hash key of the position. The hash key is stored so that we can later ensure that we only use this information with the same position; other positions will hash to the same slot. The hashing function becomes very simple: We just take the hash key modulo the hash table size to get the hash slot.

The quality of the random numbers used is important to keep the probability of collisions low. To ensure this, we do not use the pseudo random number generator (PRNG) provided by the programming language. Instead, we use the so-called “Mersenne Twister” [17], a renowned PRNG with excellent properties.

4.2.2 Repetition Detection

To play sensible chess, we have to be able to detect the case of draw by repetition. Hash keys are also useful for solving the problem of repetition detection. We do this by keeping the game history as a list of hash keys. Whenever we enter a node, we look back in the list to see if the current hash key is in the history, indicating a repetition. This may sound tedious, but by definition, we only have to look back until the latest irreversible move (Pawn move or capture) for repetitions. The list of hash keys that have to be considered, is usually very small as irreversible moves are frequent in chess.

4.3 Improving Move Ordering

As described earlier, good move ordering is essential to the efficiency of the alpha-beta algorithm. Below we will describe the move ordering heuristics used in modern implementations [10].

The most important move ordering heuristic is to use the best move from the hash table information, if it exists. If such a move exists, it is always tried first.

Next, all seemingly winning captures are tried, sorted by expected gain. A winning capture is a capture that has a positive net gain of material. Common chess knowledge dictates that the material value of the chess pieces, in the unit of one Pawn, is roughly 1, 3, 3, 5 and 9 for Pawn, Knight, Bishop, Rook and Queen respectively. This means that a capture that wins a Queen and loses a Pawn is an example of a winning capture. A capture that wins a Rook and loses a Queen is a losing capture. A capture that wins and loses pieces of the same value is an equal capture or a trade.

A simple but inaccurate way to estimate the net gain of a capture is just to subtract the value of the attacker from the value of the victim, assuming that the attacker will be lost. A more accurate but also more time consuming way, is to implement a so-called static exchange evaluator, which looks at all the pieces that attacks the square of the capture (directly as well as indirectly), and then works out the net result of capturing repeatedly at this square, in a way similar to minimax, but much simpler and without recursion. This is not perfect, but it is considerably better than the simple approach [6].

The reason for caring so much about the accuracy of the estimation, is that it improves move ordering immensely. This is because a very large part of the moves considered in a typical alpha-beta search, are in fact very bad moves that immediately lose material without gaining any other advantage. Of course, this does not mean that we can just dismiss these moves beforehand, but when recursively considering the opponent's answers to such a bad move, we want to consider his obvious refutation move as early as possible, because it often produces a cutoff immediately. If the move from the hash table does not exist or yield a cutoff, an easy winning capture is the next best guess. And of course, the more accurate we can estimate the net gain of a capture, the better move ordering is. After the winning captures, the equal captures (trades) are tried.

After all the winning and equal captures have been tried, the so-called killer heuristic [2] is applied to the non-capture moves. If a player has an exceptionally good non-capture move in one position, then often it will be the case that the same move will be good in a lot of its sibling positions. That is, even if the opponent had chosen another move at the previous ply, our good move (our killer move) would still have been good nonetheless. This is the idea behind the killer heuristic. Every time we try a move that is so good that it causes a cutoff, we remember, that for this ply, this is a good move. Then when searching sibling positions (that is, at the same ply), we will search the killer move as the first move after the hash table move and the winning and equal captures. In typical implementations, we keep a prioritized list of a couple of killer moves, instead of just one killer move.

The so-called history heuristic [19] is applied next, also to the non-capture moves. This is a general and less efficient way of remembering good moves. Whenever a cutoff is produced, a table entry indexed by the source and destination squares of the move, is incremented by some amount. When all other heuristics have been applied to the move ordering of a node, the remaining non-capture moves are sorted by the values in this history table. In this way, moves that have often been good in other parts of the tree, will be searched before moves that we know less about.

The remaining moves are the losing captures. These are tried in order of expected loss, least loss first.

To measure how good our move ordering heuristics are, we can maintain two counters during search: the number of nodes where a cutoff was produced, and the number of nodes where the cutoff was produced by the first move tried. The ratio between these two can of course at most be 100%, indicating perfect move ordering. With all of the above heuristics applied, this ratio is typically about 90% or higher. This means that whenever there is a move that produces a cutoff, we guess correctly in about 90% of the cases, and search this move first. In many tactical positions where a single move is considerably better than the rest (for instance a winning capture or move forcing mate in number of moves), the ratio is typically around 98% or higher.

4.4 Iterative Deepening

The obvious way of using the alpha-beta search function is just to call it once from the main program, when we want to find the best move of the current position. A problem with this in practical settings, is that we have to choose in advance how deeply we want search, because we have to supply a depth parameter. First of all, we do not know how shallow a search we can make, and still "solve" the position, that is, find the actual best move of the position. Also, we do not know how long the search will take before it finishes or before it at least finds a good move. This can be a problem if there are time constraint issues.

To this end, a technique known as iterative deepening is used. The alpha-beta search function is called repeatedly with increasing depth until either a good move has been found, time has been used, some fixed depth has been reached, or whatever criteria we would like to satisfy. The initial objection to this, is that it seems to waste time doing many shallow searches instead of just one deep search. In practice however, iterative deepening actually helps search efficiency immensely, for several reasons. First of all, the hash table remembers many of the important results of the previous shallow searches, which means that searches will be cheaper than if the hash table was unused. Also, a great deal of move ordering information is found; the hash table, killer move tables and history tables are filled with useful information, making move ordering better on subsequent searches. Finally, the resulting value from one call of the search function can be used as a good candidate for the expected value of the next search, which helps the efficiency of aspiration windows, as described earlier.

4.5 Principal Variation Search

Many modern programs use a variant of the plain alpha-beta algorithm called principal variation search or PVS [15]. The idea is to utilize the fact that we can normally achieve near-perfect move ordering as described earlier. This means that most often, the first move searched is the best, and if a cutoff is not produced, *alpha* is raised to the value of this move. If this is the case, the rest of the moves will return a value not greater than *alpha*, the value of the first move. Instead of searching the remaining moves with a full alpha-beta window, we search with a so-called minimal window around *alpha*. That is, we pretend *beta* is $alpha + 1$, and search the remaining moves with this window. We assume here that evaluation is done in integers and that the smallest possible difference in evaluation is 1. Such a minimal window search is cheaper and visits fewer nodes, than a full-windowed search. If it is true that the value of these moves are not greater than *alpha*, the value returned from searching these moves will be *alpha*. When we are occasionally wrong and a later move is better than the first, search will return *beta*, which in this case is $alpha + 1$. If this happens, we do a re-search of the move with a normal alpha-beta window to find out the true value of the move. With the move ordering heuristics described, this happens infrequently enough to give a considerable net gain: we search fewer nodes overall.

4.6 Quiescence Search

Some of the most difficult things for the evaluation function to evaluate fairly accurately, are the wild and complex exchanges of material that often happen in chess. In general, it is easier to write an evaluation function that focuses on the more static and long-term features of the position, than the more dynamic and short-term features. When we do a fixed depth alpha-beta search, there is no guarantee that the leafs of the trees, which are the ones that are getting evaluated, are not wild and complex positions. Therefore, instead of just evaluating when remaining depth is zero, we conduct a selective search, called quiescence search, [4] that takes only "interesting" and non-quiet moves into account. These are the moves that will potentially upset the evaluation greatly. In chess these are mainly the captures and the promotions, moves affecting material. Whenever the quiescence search reaches a position where there are only quiet moves, it relies on the evaluation function.

If quiescence search is to mindlessly catch every possible interesting move, the tree it searches will explode in size. A trivial implementation of quiescence search as described here, will result in a tree where much more than half of the nodes searched in total, will be searched by the quiescence search. This means that less time is available for the more accurate normal search. Therefore, in good implementations of quiescence search, seemingly futile moves are not searched. A move is futile if it seems to be a losing capture. A move is also futile if the estimated gain from the capture is not enough to raise the material evaluation up near $alpha$. Example: If $alpha$ is zero, meaning that no player has an advantage, we know that we can already achieve such a fairly balanced situation. If we are then searching a line in quiescence search where the opponent captures our Queen, then replying by capturing a Rook or a Pawn normally is not enough to bring the value back near $alpha$. A non-futile reply in this case would be also capturing the opponents Queen. Removing futile moves from quiescence search will cause it to only take up about 10–30% of the entire search tree, typically.

4.7 Search Extensions

Another problem with fixed depth searches is the so-called horizon effect. The horizon effect is what happens when the search, because of limited depth, tries to avoid something unavoidable or in general tries to achieve an unachievable goal. Example: White searches to a fixed depth of 5 plies, and chooses to sacrifice a Pawn and a Knight to avoid losing a Rook. If he had searched to a depth of 7 plies, he would have seen that the loss of the Rook was unavoidable, and would have kept his Pawn and Knight.

Quiescence search as described above, is a way of dealing with a specific class of problems related to the horizon effect. A more general way, that is not limited to the leaves of the normal search tree, is to extend the search depth a little bit in subtrees where something "interesting" or forced happens. This shapes the tree so that more time is used on "interesting" and forced lines of play, and less on uninteresting or nonsensical lines.

The most common extensions, and the ones we use, are:

Check extension

Whenever a checking move is played, search depth is extended in this particular subtree [22].

One reply extension

If we are in a position with only one legal move (typically a reply to a checking move), search depth is extended in this subtree [14].

Recapture extension

If the opponent just captured one of our pieces, and we then capture his capturing piece, search depth is extended. Repeated exchanges on the same square are common in chess and often forced: each move in the exchange is the only reasonable move [5].

Pawn extension

Whenever a Pawn is nearing promotion, search depth is extended. Typically, extension is done when a Pawn is moved to the 7th (or 2nd) rank [14].

One problem to be aware of is that the search tree can explode in size if extending is done too often or too much. One way to avoid this is to limit the amount of extension done at each node or in each line of play. To control extensions in more detail, so-called fractional extensions [14] can be used. Instead of always extending an integral number of plies, we can use fractions of one ply instead. Example: Both the check extension and the one reply extension could be made to extend the search 0.75 plies. Then the search is only actually extended if more than one of these extensions happen in the line searched, because only an integral number of plies can be searched, of course. In this way, even if either one of the extensions are of less than one ply, the recursive accumulation of extensions will cause extensions in lines of play that are particularly "interesting".

4.8 Null Move Pruning

As described earlier, many positions in a typical chess search tree are nonsensical in that they are immediately losing for one side. Null move pruning [3] is a way to avoid spending too much time on such positions. It works like this: before actually searching any moves of the position (but after having probed the hash tables for a quick return), a so-called null move is performed and the position is searched with reduced depth with a minimal window around *beta*. A null move is effectively a pass; we make no move and give the right to move to the opponent. If this search returns *beta*, we return immediately from search. The idea is that a null move (which of course is not legal in chess), would normally be a very bad move. If we can perform a null move and still have a value above *beta*, a real search would probably just find an even better move that would also have a value larger than *beta*, causing a cutoff anyway. In other words, the opponent just made a very bad move, too bad to be considered.

The reason why a minimal window around *beta* is used, is that we are only interested to know whether this position has a value below *beta* or not. A search with a minimal window around *beta* is enough to prove this, and such a search will be cheaper than a wider window search.

Null move pruning is speculative and inaccurate for at least two reasons: Firstly, we are making a reduced search depth to establish the value of the position after a null move. A normal reduction used is two plies. Such a reduced search is not as accurate as non-reduced search. Secondly, the assumption that a pass is always bad is not always true. In certain very special positions called *zugzwang* (German for "compulsion to move") positions, which typically occurs in the endgame, doing nothing would actually be the best move. In such a case, the null move search might tell us that the position is very good, when in fact it is a zugzwang position where every legal move at our disposal makes our game worse. To counter this problem, null move pruning is typically disabled in endgame situations.

The great advantage of null move pruning comes from the fact that in many positions, the full depth searches of all the legal moves is replaced by one much cheaper reduced depth minimal window search. In practice, null move pruning helps immensely and allows searches on the order of two plies deeper within the same time.

Chapter 5

Data Structures

Throughout computer chess history, alpha-beta search has been the dominant algorithmic framework, although a lot of improvements to the plain algorithm have been found as described earlier. But even within this algorithmic framework, the choice of the fundamental data structures have varied wildly. The choice of data structures will affect what information will be hard to retrieve, and what will be easier. Also, it will affect how fast the implementation of the fundamental operations of the alpha-beta algorithm can perform. These operations are:

- Generation of legal moves of a position
- Generation of child positions from a legal move and a parent position
- Evaluation of a position

It is normally not feasible to actually generate a new "physical" child position from the parent position. Instead, an operation (makeMove) is used to transform the parent position into the child position. Another operation (unMakeMove) is used to undo this transformation. Therefore, the second bullet above can be replaced by these two: Transformation of the parent position into the child position and transformation of the child position back into the parent position.

Some of the most important questions to be able to answer quickly, are these:

1. Is there a piece of type x (meaning: of a certain type and color) on square s?
2. Which piece is on square s?
3. Where are the pieces of type x?
4. Are there any pieces of type x on the board?
5. How many pieces of type x are on the board?
6. Which squares are attacked by piece p?
7. Is square s attacked by any piece (of color c)?
8. Which pieces are attacking square s?

5.1 Board Representation

In every implementation, we need to represent the chess board state. Apart from the board state, to play fully legal chess we must also remember whose turn it is, the castling rights of each player, the en passant square, the half move clock and (the hash keys of) the previous positions of the game. The half move clock is the number of half moves played since the last irreversible move (capture or Pawn move), and is necessary to enforce the rule of draw in case of 50 consecutive reversible moves. We have to know (the hash keys of) the previous positions of the the game because we have to be able to detect draw by repetition. Because Pawn moves and captures are irreversible, no position after such a move can be a repetition of a position from anytime before such a move. Therefore, we only need to remember the positions of the game back to the last irreversible move.

The simplest implementations use a plain array of 64 bytes to represent the chess board. Each square can be in 13 different states: empty, holding one of the 6 different white pieces or holding one of the 6 different black pieces. This implementation makes it fairly easy to answer questions no. 1 and 2 above. It can be done in constant time. Questions no. 3, 4 and 5 are

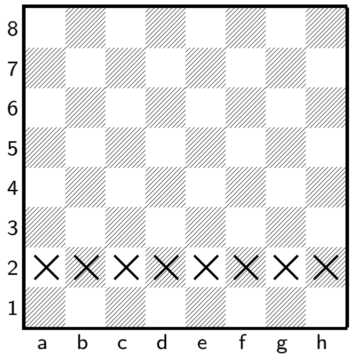

Figure 5.1: The `whitePawns` bitboard.

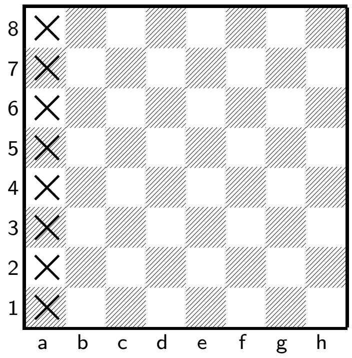

Figure 5.2: The a-file bitboard.

a little harder, and can be answered in linear time. Questions no. 6, 7 and 8 are not as easily answered.

Of the fundamental operations, making and unmaking moves is fairly easy with this implementation. Evaluation and generating legal moves is not so easy. Many evaluation terms will be tedious to implement and will be slow to execute. Generation of legal moves will be complicated since we essentially have to be able to answer the above question 7 about a square being attacked, to determine whether a move leaves the King in check, making it illegal. Also, several problems related to systematically detecting the boundaries of the board will occur, specifically with the Knight and the sliding pieces.

5.2 Bitboards

Many modern computers work natively or at least very fast with 64-bit integers. Bitboards [1, 22, 12, 10] are a way to represent information about the chess board state, that utilizes this fact. A bitboard is simply a 64-bit (unsigned) integer that holds boolean information about the board state. Example: We might have a bitboard called `whitePawns` that holds information about the white Pawns on the board. The 1st bit corresponds to the square a1, the 2nd bit to the square b1 and so forth. Whenever a white Pawn is

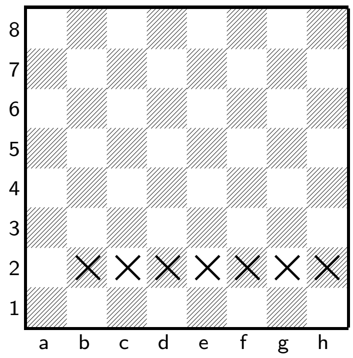

Figure 5.3: The Pawn on the a-file has been removed.

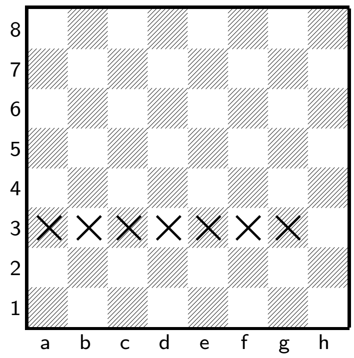

Figure 5.4: The attacked squares.

present on a square, the corresponding bit is 1, and 0 otherwise. In the initial position, this bitboard can be graphically represented as in Figure 5.1.

Bitboards are not only important because they fit a lot of information into a small space, but also because most computers support many useful operations on bitboards. First of all, the common boolean operators are very useful with bitboards. Example: We want to know what squares the white Pawns attack capturing diagonally to the left in the initial position. We start with the bitboard in Figure 5.1. Then we mask out the Pawn(s) on the a-file since they can never attack diagonally to the left. We do this by performing a bitwise AND with the bitwise negation of the bitboard in Figure 5.2 which is a bitboard representing the a-file. The resulting bitboard can be seen in Figure 5.3.

Finally, we do a logical shift of the result 7 places to the left. Note that doing a logical left shift of a bitboard actually shifts it graphically to the right in our figures. The resulting bitboard holding the attacked squares, is shown in Figure 5.4.

In pseudo-code, this entire calculation can be expressed as:

```
attackedSquares = (whitePawns & ~fileA) << 7;
```

On most architectures this calculation will be fast, because the AND, the NOT, and even the logical shift are all done in constant time and in few clock cycles. The calculation deals with all the Pawns at once instead of one Pawn at a time, and can be regarded as a form of simple parallelism.

If we wanted to answer the same question but with the white Pawns capturing to the right, we would have masked the Pawn(s) on the h-file out instead, and we would have made a logical shift 9 places to the left instead.

Besides the `whitePawns` bitboard, similar bitboards for all the other types of pieces, both black and white, are of course also useful. Also useful are two bitboards holding the positions of all the white pieces and black pieces, called `whitePieces` and `blackPieces` respectively.

If we maintain these bitboards in addition to a traditional array of 64 squares, we can now easily answer question 3. The bitboard of pieces of type x holds exactly this information. But of course, to be really useful, we have to be able to retrieve the answer in the form of a set of squares, not just a bitboard. For this, a function called `firstBit()` is used. It returns the position of the first 1-bit in the bitboard. To get the position of all the 1-bits in the bitboard, `firstBit()` is used repeatedly to find and clear the first 1-bit in the bitboard, until it is empty. If `firstBit()` was a very complex function, bitboards would lose much of their potency. Fortunately, very fast implementations of this function exist. The simplest is based on table lookups and is typically very fast. On many architectures, `firstBit()` is even a native machine code instruction. It can be regarded as a fast constant-time operation. Answering question 3 with only the simple array representation takes linear time proportional to the size of the board, because all squares must be scanned one at a time. With bitboards and the `firstBit()` function, it takes linear time proportional to the number of pieces of type x, because we get the position of each of them in constant time.

Question 4 is very easily answered: The bitboard holding information about the pieces of type x can be used as a boolean variable. If it is zero, it means that there are no pieces of type x. If it is non-zero, it means that there are some.

Answering question 5 with the simple array representation takes linear time proportional to the size of the board. With bitboards, a function called `popCount()` is used to answer this specific question. Again, fast implementa-

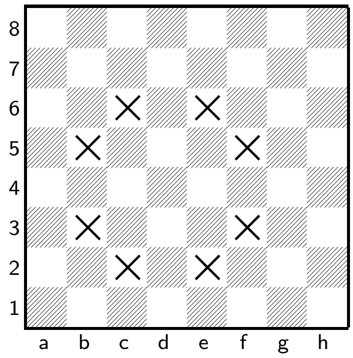

Figure 5.5: The pre-computed `knightAttack[d4]` bitboard.

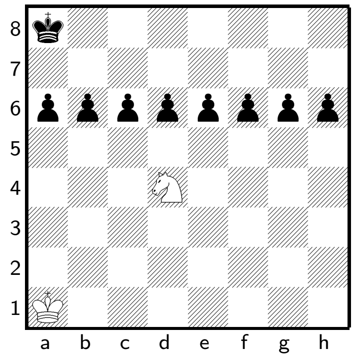

Figure 5.6: White Knight and black Pawns.

tions exist for this function that take linear time proportional to the number of 1-bits in the bitboard. And it too is a native machine code instruction on many architectures, in which case it is usually executes in constant time.

5.2.1 Attacks

Answering questions 6, 7 and 8 is where the power of bitboards is truly shown, compared to the simple array representation. With the simple representation, to know what squares a Queen attacks in a given board state, we would have to iteratively try all squares in all directions that the Queen can move in, until we hit a piece or the end of the board. This would take linear time in the number of attacked squares. With bitboards we can use pre-computed attack tables for all pieces.

Let us start with the Knight. We have an array called knightAttack holding the attack patterns of the Knight. Figure 5.5 shows the bitboard `knightAttack[d4]`.

With this, we could quickly calculate which opponent pieces the Knight attacks. If the position looks like Figure 5.6, we can find out which pieces the Knight attacks by calculating

```
attackedPieces = (knightAttack[d4] & blackPieces);
```

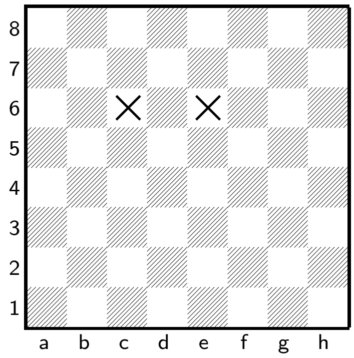

Figure 5.7: The squares of the pieces that the Knight attacks.

The result would be the bitboard in Figure 5.7.

The exact same thing can be done for the attacks of the King. When it comes to the sliding pieces, things get a little more complicated.

5.2.2 Rank Attacks

In the case of a sliding piece such as the Rook, we could try the same approach as with the Knight and the King. But in most real positions the attack pattern of the Rook will depend on the board state, that is, it will depend on the placement of the other pieces on the board. If we have the position in Figure 5.8, the attack bitboard for the Rook at d4 will not be as trivial as with the Knight, because the Rooks scope is blocked by various pieces.

The solution is to pre-compute the attack bitboards for all possible states on a file or rank, and then index the attack tables by square as before, but also by rank and file state. To determine the scope of a piece, it is not important to know exactly what kind of pieces are blocking it. It is enough to know what squares these pieces are on. This means that there are really only 256 different states possible on a rank. To get the rank state, we start with the bitboard of occupied squares (which we can choose to maintain in general or calculate easily from `whitePieces` and `blackPieces` by ORing). The `occupied` bitboard of the position in Figure 5.8 is shown in Figure 5.9.

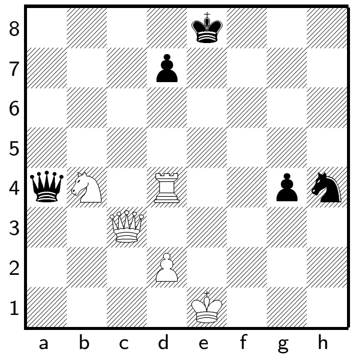

Figure 5.8: Limiting the scope of the Rook.

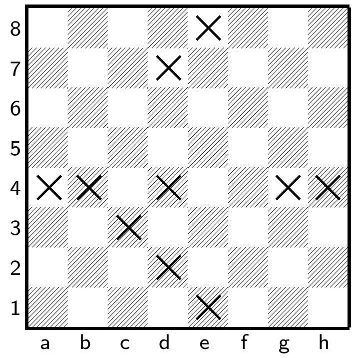

Figure 5.9: The bitboard of occupied squares.

We then perform a logical shift right of the `occupied` bitboard until the rank in question (the 4th rank in this case) is in the least significant bits of the bitboard. In this case this means shifting 24 places to the right. The resulting bitboard can be seen in Figure 5.10.

The 8 least significant bits of this bitboard are 11010011 or in decimal: 211. This can be used as an index into a pre-computed table called rankAttack. The bitboard `rankAttack[d4][211]`, shown in Figure 5.11, contains exactly the rank attack pattern of the Rook in the position shown earlier.

The entire calculation can be expressed as:

```
rankAttack[d4][(occupied >> 24) & 255];
```

This last bitboard suggests that the Rook attacks both b4 and g4, even though the Knight on b4 is white, just as the Rook itself. This is because we do not differentiate between differently colored pieces when extracting the state. Given this attack bitboard, we can choose which pieces we want to actually attack, if any. To generate non-captures, we would AND out all pieces from the attack bitboard, using the `occupied` bitboard. If we want to generate captures also, we would AND out only our own pieces, using the `whitePieces` bitboard in this case.

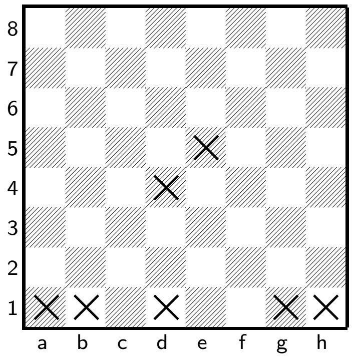

Figure 5.10: The shifted `occupied` bitboard.

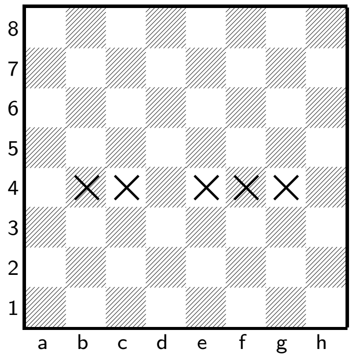

Figure 5.11: The pre-computed `rankAttack[d4][211]` bitboard.

5.2.3 Rotated Bitboards

Computing the attacks of the Rook along a file is done in a similar way. The difference is that it is not as easy to get the file state as it was to get the rank state. We cannot just shift the `occupied` bitboard a certain number of times to get the file state, because it is not collected in one consecutive 8-bit part. The solution is called rotated bitboards. We maintain a version of the `occupied` bitboard that is rotated 90 degrees, called `occupied90` and use it to extract the file state in exactly the same way as with the rank state: by shifting. If we rotate the `occupied` bitboard in Figure 5.9 counterclockwise 90 degrees, and shift the resulting bitboard 24 times to the right and AND with 255, we get the file state of the d-file. We can then get the file attack pattern of the Rook by using a pre-computed table called fileAttack, similar to rankAttack. The entire calculation becomes:

```
fileAttack[d4][(occupied90 >> 24) & 255];
```

The compound attacks of the Rook is then computed by simply ORing the two resulting rank and file attack bitboards.

The same idea is used again to calculate diagonal attacks. We maintain two more versions of the `occupied` bitboard: one rotated 45 degrees clockwise called `occupied45R` and one rotated 45 degrees counterclockwise called `occupied45L`. It may not be exactly clear what it means to rotate a bitboard 45 degrees. But the important thing to note is that the 64 squares of the board of course still fits into 64 bits regardless of how they are mapped and that we want to be able to easily extract the state of a diagonal.

If we rotate the board 45 degrees clockwise, the diagonal a1–h8 and all diagonals parallel to it are now horizontal. We can then map the "first" diagonal at the bottom (containing only h1) to the first bit of the rotated `occupied` bitboard. The next diagonal (g1–h2) can be mapped at the two next bits. The next diagonal (f1–h3) can be mapped at the three next bits and so on, until the last diagonal (containing only a8) which is mapped at the last bit.

We can now use this bitboard to get the state of a diagonal by logical shifting. But because not all diagonals have the same length, the amount of shifting needed will not be a simple multiple of 8. The first diagonal (containing only h1) will need no shifting. The next diagonal (g1–h2) will need to be shifted 1 place, because the previous diagonal contained 1 square. The next diagonal (f1–h3) will need to be shifted 3 places because the previous diagonals contained 3 squares in total. This pattern continues with 6, 10, 15, 21, 28, 36, 43, 49, 54, 58, 61 and 63 places. This pattern is of course kept in a lookup table.

To make things easy, instead of exactly extracting the right number of bits for indexing, we always extract 8 bits. With diagonals shorter than 8 squares, this means that the most significant bits (of the 8 state bits) can contain "garbage" from other diagonals. But if we make our diagonal attack lookup table hold redundant information so that it has the right answer regardless of the extra "garbage" bits, things become easy again.

The calculation of the attacks on one diagonal from a Bishop on d4 becomes:

```
diagonalAttack45R[d4][(occupied45R >> 28) & 255];
```

Of course, a bishop attacks in two diagonal directions, so the same thing is done for the remaining diagonals, the ones that are parallel to h1–a8. For this, we use an array called diagonalAttack45L instead.

We can now calculate the attacks of all pieces. Rook attacks are calculated by ORing file and rank attacks. Bishop attacks are calculated by ORing the two diagonal attacks. Queen attacks are calculated by ORing Bishop and Rook attacks. Knight and King attacks are taken directly from lookup tables and Pawns can be dealt with by shifting and/or by lookup tables.

5.2.4 Answers

We can now answer questions 6, 7 and 8 fairly easy.

The answer to question 6 is exactly the piece attack bitboards described in the last section.

Answering question 7 is done as follows: For each kind of piece, we pretend that such a piece is standing on square s, and if this imaginary piece attacks any piece of its own type, we know that the square is attacked. Example: If an imaginary Queen standing in d4 would attack a real Queen, then it means that this real Queen attacks d4. To be precise, we just AND the attack bitboard of a Queen at d4 with the bitboard of Queens. To answer question 7, we just have to do the same for all kinds of pieces (of a certain color), until we find a piece that attacks s, in which case we can answer yes, or until we have tried all kinds of piece and found none attacking s, in which case we can answer no. All this is fairly fast with bitboards.

Question 8 is dealt with in much the same way: For all kinds of pieces, we calculate the bitboard of such pieces attacking s, as above. We then just OR together the results together to have a bitboard of all attackers of s.

5.2.5 Operations

Bitboards affect how the fundamental operations of the alpha-beta algorithm are performed.

Move Generation

Generating the legal moves of a position is fairly easy with bitboards. We just use `firstBit()` to sequentially find all pieces of the relevant color. For each piece found, we calculate the attack bitboard. We then use `firstBit()` again to extract each attacked square. This gives us both the source and the destination square of the move. Dealing with en passant captures and promotions is trivial. The last move is castling. To check whether castling is legal, we have to check whether the squares between the King and the Rook in question, are empty. This is easily done by just ANDing a precomputed bitboard of these squares with the `occupied` bitboard. We also have to ensure that the squares that the King travels through, including the source square and the destination square, are not attacked by the opponent. With the simple representation, this would be difficult and time consuming. With bitboards, it is just a matter of answering question 7 for these squares.

To really generate only legal moves, we would have to check every move to see if it left the King in check. Since this is time consuming and can be avoided easily, we are actually generating pseudo-legal moves: moves that are legal if leaving the King in check is not considered illegal. If we want to know whether a move is really legal, we can make the move, check if it leaves the King in check, and unmake the move again. Most of the time, true legality of moves is only important when searching. And it is cheaper to let the search itself check for legality of moves when needed, because the search is already making and unmaking the moves. And if a cutoff occurs, we have not wasted time checking the remaining moves for legality.

Making And Unmaking Moves

Making and unmaking moves is a little more involved in the bitboard representation than in the simple representation. Apart from updating the simple array of squares, we also have to update the bitboards of the piece types involved, and also the four versions of the `occupied` bitboard. Updating the bitboards when making a move is just a matter clearing the source square by ANDing, setting the destination square by ORing, and in the case of the rotated bitboards, looking up the rotation mapping. All this is typically very fast on modern computers. When unmaking a move, the same things are done, but with squares in opposite roles.

We have to do a little bit of trivial extra work in the case of the special moves such as en passant, promotion and castling. But this would be true with the simple representation too.

Bitboard Evaluation

Evaluation can be done in an infinite number of ways. It is up to the imagination, knowledge and good taste of the author of the evaluation function to decide exactly what is important, how important it is and how to implement it. But bitboards makes it very easy to answer many questions that would otherwise be difficult or near impossible to answer.

First of all, we do not have to consider the entire board to find all pieces. We can use `firstBit()` to iterate through the pieces that are actually on the board. This makes evaluation faster especially in endgame situations with few pieces on the board.

Also, attack information is cheap and can be used in all sorts of ways. Two evaluation terms that are very common are mobility and king safety.

Mobility is the ability to move around quickly, and a good measure for this is the number of attacked squares. If our pieces are blocked early in many directions, they will attack few squares. In contrast, an unblocked piece standing near the center of the board will attack many squares. Calculating mobility is easy with bitboards. We just use `popCount()` on the attack bitboard of the piece.

King safety is a measure of how well protected the King is. During the middlegame with many pieces on the board, The King is better off near the corner of the board with a shield of Pawns protecting him, than he is in the center of the board. We can use bitboards to quickly evaluate the pattern of Pawns in front of the King, and to evaluate which opponent pieces are in the same quadrant as the King and similar things.

A huge and important evaluation subject in itself is pawn structure. As described in many books on chess, the formation of the Pawns on the board dictates the long term direction and strategy of the game, for several reasons: First, Pawns are the least valuable pieces, primarily because of their limited mobility. This, ironically, make them the most forcing attackers. If a Pawn attacks some opponent piece worth more than a Pawn, the opponent will be wise to respond to this threat, because otherwise he will lose a strong piece for a lowly Pawn. On the other hand, if the mighty Queen attacks an opponent piece, this opponent piece will most often be worth less than a Queen. And so, the opponent need not fear a trade if his piece is protected, since it would be advantageous to him. The second reason why Pawns are important is because they can only make slow, irreversible moves. If a Pawn in the Pawn shield of the King has moved forward, the safety of the King is compromised permanently. The Pawn can never go back. If the central Pawns of both players are blocking each other in the center of the board, mobility will in general be limited for all pieces on the board. If all central Pawns are traded and off the board, the game can take on a much more dynamic and tactical nature, because mobility is not being limited in the center. The pawn structure changes only slowly and forms the strategic terrain of the chess board. Evaluating the pawn structure can involve many complicated calculations. Bitboards can make a lot of these calculations much easier.

In general, many chess program authors choose bitboards not primarily for their speed, but for the simplicity and cleanness it gives to the design of the entire program in general and to the evaluation function in particular.

Chapter 6

Parallel Searching

In general, when parallelizing a sequential algorithm we have at least the following main problems:

- We have to find some way of dividing the work that has to be done, so that it can be done in parallel. We want to keep all available processors as busy as possible doing useful work at all times.
- There will be some waste processor time related to the startup of the parallel algorithm. Not all processors will be doing useful work right from the start. We would like to minimize this waste.
- There will be some waste related to splitting a problem up into subproblems to be searched in parallel. We would like to minimize this waste too.
- There will be some waste related to combining the solutions to subproblems solved in parallel into a solution for the original problem. Of course, we would like to minimize this waste as well.

In the following, we will address these problems in the specific case of parallelizing the alpha-beta algorithm with the modifications and heuristics described earlier.

6.1 Parallel Alpha-Beta

The alpha-beta algorithm is inherently sequential in nature; the tree is traversed in a certain order and knowledge about boundaries on the real value is propagated forward through the search. Still, opportunities for parallelism do exist. Many subtrees have to be searched even if we have near perfect move ordering. Searching different subtrees in parallel is one way to obtain parallelism. We split up the searching of all the legal moves of a node into several threads. There are problems with this approach, though, that prevents us from using it trivially. Some problems are equivalent to the general problems mentioned earlier, some are specific to this domain:

- If a node contains a move generating a cutoff, the remaining moves would not get searched in the sequential alpha-beta algorithm. In the parallel algorithm, if such a node gets split some or all of these remaining moves will be partially or fully searched. We want as few as possible of these cases, and we want to be able to stop the superfluous searches as quickly as possible when we detect them.

- Time spent splitting is not spent searching. If it takes a very long time to split, average search speed will be slow. We want splitting to be as fast as possible.

- Some global data structures, most notably the hash table, have access issues that we have to deal with in the parallel case. If two threads write to the same hash table entry at the same time, the result can be an inconsistent entry.

- Even if splitting is fast, it will take time. We have to be careful not to do a split in cases where we could have searched the node in the same or smaller amount of time without splitting. Not only will this waste time, it will also tie up threads that could possibly be used better elsewhere.

We will address these points in the following sections.

6.2 Alpha-Beta Revisited

In [13], Knuth & Moore analyzed the properties of randomly and perfectly ordered game trees. Since we can achieve near-perfect move ordering in the case of chess, the properties of perfectly ordered trees are interesting to us. Knuth & Moore found that nodes could be divided into three categories:

Type 1

All the nodes on the principal variation (PV) are type 1 nodes. This means that the root node is a type 1 node. The first child of a type 1 node is also a type 1 node, because we assume a perfectly ordered tree. The rest of the children of a type 1 node, are type 2 nodes. Because it is a node on the principal variation, no cutoffs can occur; all of the children of a type 1 node have to be searched.

Type 2

A type 2 node is a child of a type 1 node or a type 3 node. A type 2 node is a node where a cutoff occurs. Since we assume perfect move ordering, we will only have to search one child of a type 2 node. The children of type 2 nodes are type 3 nodes.

Type 3

Type 3 nodes are children of type 2 nodes. No cutoffs will occur, they require all of their children to be searched. Move ordering is irrelevant at these nodes because they all have to be searched, and none of their children are good enough to raise *alpha*; they will all be searched with the same bounds.

In [16], nodes of type 1, 2 and 3 are called PV, CUT and ALL nodes respectively. These names describe in a concise manner the difference of the node types.

The categories outline a basic strategy for doing alpha-beta search in parallel. Firstly, it is clear that type 2 (CUT) nodes are not suited for splitting. When the first child has been searched, a cutoff is produced, and any searches in parallel of the rest of the children is wasted work. Secondly, Type 3 nodes are excellent candidates for splitting. All their children have to be searched, and we can gain a lot by doing this in parallel. Thirdly, since all children of type 1 nodes must also be searched, these seem to be good candidates for splitting too. But they differ from type 3 nodes in an important way: In type 3 nodes, *alpha* and *beta* stays the same throughout searching of the children. In type 1 nodes, *alpha* and *beta* bounds have to be established by searching the first child before the rest of the children can be searched with these bounds. This means that the first child of a type 1 node should be searched sequentially. After this has been done and bounds have been established, we can split the node searching the rest of the children in parallel.

Of course, we are not searching perfectly ordered trees. But because of the the good move ordering heuristics in general and the hash table in particular, in type 1 nodes, we will in most cases actually search the best move first. And, as mentioned earlier, in about 90% of the nodes where a cutoff occurs (type 2 nodes), it happens already after the first move, as in the perfect case. Nodes of type 3 will have to be searched thoroughly in any case.

6.3 Young Brothers Wait Concept

A basic algorithm that takes the above knowledge into account is called Young Brothers Wait Concept or YBWC [8]. It works by always searching the first child sequentially. It then searches the rest of the children in parallel. This fits type 1 nodes perfectly. It also fits type 2 (CUT) nodes perfectly, because there will be a cutoff after the first child has been searched. It does not fit type 3 nodes as perfectly. The first child will be searched sequentially when we could instead search all the children in parallel. This is of course not optimal, but it is a simple splitting strategy that works fairly well. We can try to improve on this by guessing the type of a node and act accordingly. But it might not be easy to guess correctly often enough for it to be an advantage. If we guess wrong at a type 2 (CUT) node, we will search a lot of superfluous nodes.

6.4 Splitting With Bitboards

The time it takes to perform a split will depend on various implementational and architectural details. A very important part of splitting, though, is to transfer information about the node from the splitting thread to the idle threads that will help search the node. Necessary information to transfer is the board state, the side to move, the en passant square, the castling rights, the half move clock to be able to detect draw by the 50 moves rule, and enough of the game history to enable us to detect repetitions. Memory transfers are never free, so we want to minimize the amount of information transferred. The bitboard implementation is in direct opposition here because it takes up much more memory to represent the board state, than the simple representation does. In the simple case, the board state consists of 64 squares usually represented as bytes, taking a total of 64 bytes. In the bitboard implementation, we have this array too, and in addition we have the bitboards: 12 bitboards for the pieces, 2 `occupied` bitboards, one for each color, and 3 rotated `occupied` bitboards. In total: 17 bitboards of 8 bytes (64 bits) each. This means that we have to transfer 136 bytes more than with the simple representation. It is difficult to assess in advance whether bitboards will be an overall advance or a drawback in a parallel chess program. Splitting will take longer, but faster search and evaluation may balance this out. We will try to answer this question in the next chapter on experiments.

6.5 Global Data Structures

In the sequential case, the hash table and the data structures for the killer heuristic and the history heuristic are globally available to the search function at all times. In the parallel case, we have to decide what to do with the access issues that surface with these structures. Multiple simultaneous writes to these may have disastrous effects for the program.

6.5.1 The Hash Table

There are many different ways to deal with the hash table in the parallel case. One is to distribute the hash table among different threads so that each thread is responsible for a part of the table. Another idea is to keep the table global and then ensure mutually exclusive access from different threads with mutex locks. There are problems with these solutions, though. In the distributed case, inter-thread communication when one thread probes an entry that belongs to another thread, is slow. The mutex solution also introduces a performance penalty because mutex locks are slow on most architectures.

A recent idea [11], and the idea we use, is to keep the hash table global in shared memory and ignore the multiple simultaneous writes issue, and instead do a cheap but efficient form of error-detection on the hash table entries: Each hash table entry is 128 bits in size. We can think of this as consisting of two 64-bit parts. The first part is the hash key of the node about which information is stored, the second part is the actual information about the node. When storing a new entry, instead of just storing both parts normally, we store only the second 64-bit part normally. The first 64-bit part is XORed with the second 64-bit part before storing. When the table is later probed and this entry is found, we can check the consistency of the entry: The second part of the entry is just read normally. The first part of the entry is read and then XORed with the second part. This way, if the entry is consistent, we get the original first part because XORing with the same value twice is the same as doing nothing. The original first part was the hash key of the position about which information is stored in the entry. If the hash key of the position that is probing the hash table matches the hash key stored in the entry, we have a hash hit and the information in the entry can be used. If it does not match, it can either be because the information was stored from a different position hashing to the same slot or because the entry has been corrupted by simultaneous writes. In either case, we have a hash miss and we do not use the stored information.

A potential problem could be that too many nodes are corrupted, and while we do not use corrupted entries due to the error detection, the number of useful entries could become so low that the hash table is not fulfilling its purpose. On the other hand, corrupted entries will be overwritten with consistent nodes sooner or later so the problem may not be serious. We will return to this question in the next chapter.

6.5.2 Killers And History

The data structures used by the killer heuristic and the history heuristic face the same multiple simultaneous writes issue as does the hash table. The difference is that while a corrupted hash table entry could make the search return an untrue value or even crash the program if it attempts to perform a nonsensical best move from the hash table, the killer and history data is only used for move ordering purposes. The worst that can happen is bad move ordering. On the other hand, this can be bad enough. We do not want alpha-beta (or PVS) to degenerate into plain minimax.

What we try is this:

The killer heuristic data structures are kept local to each thread. This means that each thread only writes in its own data structures, but also that we do not get the benefit of having results from other threads. We could copy the killer heuristic data when splitting, but this would mean even more splitting time overhead, so we do not do this. In fact, we do not even reset the killer data structures in the helper threads. This means that they will contain bogus move ordering information from the last time this thread was active, in another part of the tree. We do this because limited testing has indicated that it does not affect move ordering in any serious way, and we would rather not waste time on this if it is not necessary. The killer heuristic data structures of these threads will quickly be filled with useful information as search progresses.

The history heuristic data structures are kept globally, with risks for potentially corrupt entries. This could potentially affect move ordering, but the history heuristic is the least important move ordering heuristic, and limited testing has indicated that this solution does not affect move ordering in any serious way either.

We will return to these question of move ordering in the next chapter.

6.6 Design

The overall algorithmic design of our parallel implementation is YBWC. But there are still many design decisions to take at the more detailed level. In this chapter we will describe our parallel design in more detail, without bothering too much with implementation details.

6.6.1 Inter-Thread Relationship

The YBWC idea can be used in a variety different ways. The parallel model could be message passing, pipes, master/slave and so on. In our case we use interacting peers. All threads can split a node and deploy other threads to help out. Of course, a main thread sets up data structures and creates the other threads at the the start of the program. And this main thread is always the one starting the search at the root. But during search, all threads are equal. This avoids centralized bottlenecks in the parallel infrastructure.

6.6.2 Parallel Operation

The important parallel operations of the program is as follows:

When the program starts, data structures are initialized and all threads are started. All threads except the main thread are going directly to an idle loop where they wait for work.

The main thread eventually calls the alpha-beta search. We then follow the YBWC idea at every node. The first child is searched sequentially. When the first child has been searched, it is time to split. To avoid wasting too much time splitting instead of searching, we only split if the remaining depth of the subtree is 3 plies or more. Even with extensions and quiescence search, it is still faster to search subtrees with remaining depth of two plies or less, sequentially. Splitting is done by copying all necessary information about the node into so-called split blocks. Each idle thread gets a split block to work on. The splitting thread is marking itself idle when splitting so it gets a split block too. A thread that has been assigned a split block calls a special search function that does essentially as it would in the sequential case at a normal node. It repeatedly searches a child. The difference is that instead of generating legal moves, it asks the original splitting node for a move to search. It then just gets the next move in the list. All threads participating in this split does this, until there are no more moves left.

Searching a child is done by performing the move and then calling the regular search function. From here on, the thread acts exactly the same way as the main thread described above. Most importantly, it itself can split at some point, if there are idle threads. The helping threads can split too, and so on recursively.

When a thread requests a new move to search, and finds that there are no more, it returns its findings to the splitting node. The splitting node remembers the best move (and its value) found yet by any helper thread. In this way, all the children of the node gets searched in parallel and the best move and value is found. The helping thread returns to the idle loop waiting for new work. Note that the subtrees of the different children can vary greatly in size so one helper thread can finish long before another. Since it goes back into the idle loop, it can help out other helper threads that are not finished searching, when these split.

Finally, we have to deal with the unfortunate case of finding a cutoff in a helper thread. Ideally, we should never find a cutoff in a node that has been split. But when it happens we want to stop the other helper threads as quickly as possible. We do this by signaling stop to all the other helper threads before returning. The helper threads may have split at some point too so they tell their helper threads to stop, and so on recursively.

Chapter 7

Experiments

We have implemented a bitboard chess program that uses all the techniques described in Chapters 3–6. In this chapter we will describe the experiments that we have done on this implementation, and attempt to explain the results.

7.1 Description

The basic experiment was done as follows: 10 selected chess positions were searched in an iterative deepening manner to a nominal depth of 11 plies. This was done using from 1 and up to 24 processors.

By searching to a nominal depth of 11 plies we mean that the search function is called with a depth parameter of 11 plies. Extensions will shape the tree, though, such that some lines are searches more than 11 plies. Also, quiescence search may selectively search some lines even longer. On the other hand, alpha-beta pruning will prune some lines long before they reach 11 plies. Thus, searching to a nominal depth of 11 plies, as in our case, is very different from searching a tree 11 plies deep without extensions and quiescence search.

The positions were selected from the 24 positions of the well-known Bratko-Kopec test suite [7]. Not all positions of this test-suite are suited for experiments in parallelism. The ones containing an unambiguous short-term tactical solution such as a mate in 5 plies, are not suited because the solution is found very early on in the 11 ply iterative deepening, leaving only little work for the rest of the search. This means that many processors will have little or nothing to do. Such positions will be solved very fast in the parallel case as well, but they will not show the full utility of searching in parallel. We are not parallelizing to solve such easy positions in roughly the same time as in the sequential case, but to solve more difficult positions faster than in the sequential case.

From running these experiments, we get some information after searching each position: the time used, the number of nodes visited, the hash table hit rate, the move ordering ratio and the percentage of quiescence nodes of the total number of nodes searched. From this information, we can obtain other important information, most notably the search speed (nodes per second) from which we will later obtain information about the speedup of the algorithm on multiple processors. We will not be looking at the results of the individual positions, but instead at the sum or average of results over all 10 positions.

7.2 Results

Our original plan was just to run the experiments once on a number of dedicated processors on a dedicated machine. Preliminary testing on a non-dedicated machine showed that the implementation worked and that speedup was reasonable with several processors. But when the results came in from the main experiment on the dedicated processors, it was obvious that something was not ideal. The results from the preliminary experiment and the results from the main experiment looked rather different. Because of this, we have decided to include both the results of the main experiment and the results of two other experiments as well, in the discussion of some of the measurements below.

The hardware used in the three experiments was this:

Main experiment

Sun Enterprise 15K with 384 GB RAM and 68 UltraSPARC-IIICu

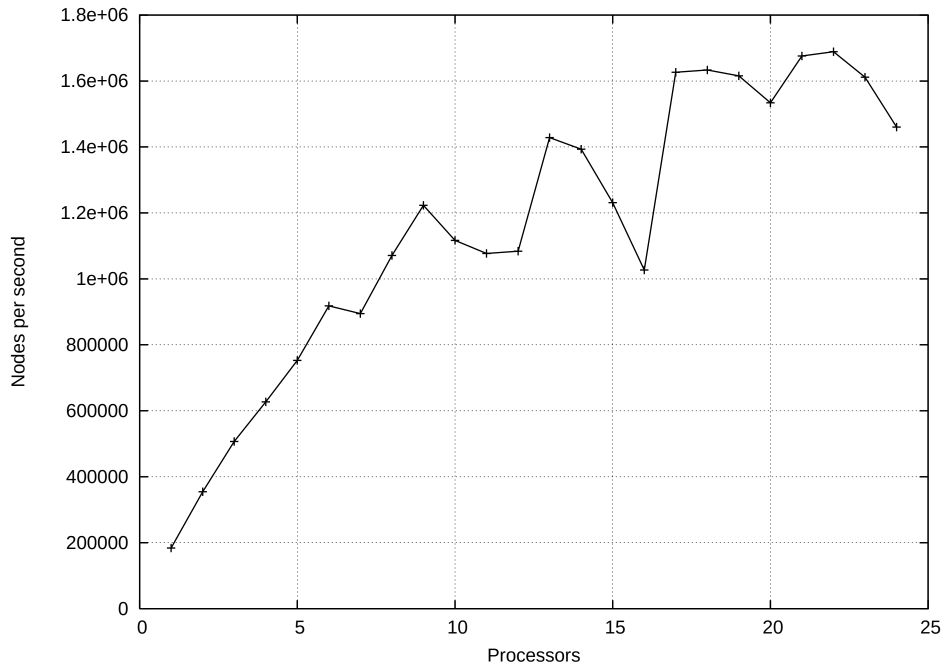

Figure 7.1: Search speed — Main experiment.

processors running at 900 MHz.

Preliminary experiment
Sun Fire 6800 with 48 GB RAM and 22 UltraSPARC-IIICu processors running at 1200 MHz.

12 Processors experiment
Sun Enterprise 4500 with 24 GB RAM and 12 UltraSPARC-II processors running at 400 MHz.

7.2.1 Search Speed

Figure 7.1 shows the development of the average search speed on the dedicated machine. As can be seen, the graph is less than smooth. Still, the general trend seems to indicate a considerable gain in search speed as we use more and more processors. And although the graph seems to flatten out at the end, 22 processors still gives better performance than any smaller number of processors. Compare this to the graph in Figure 7.2 from the preliminary

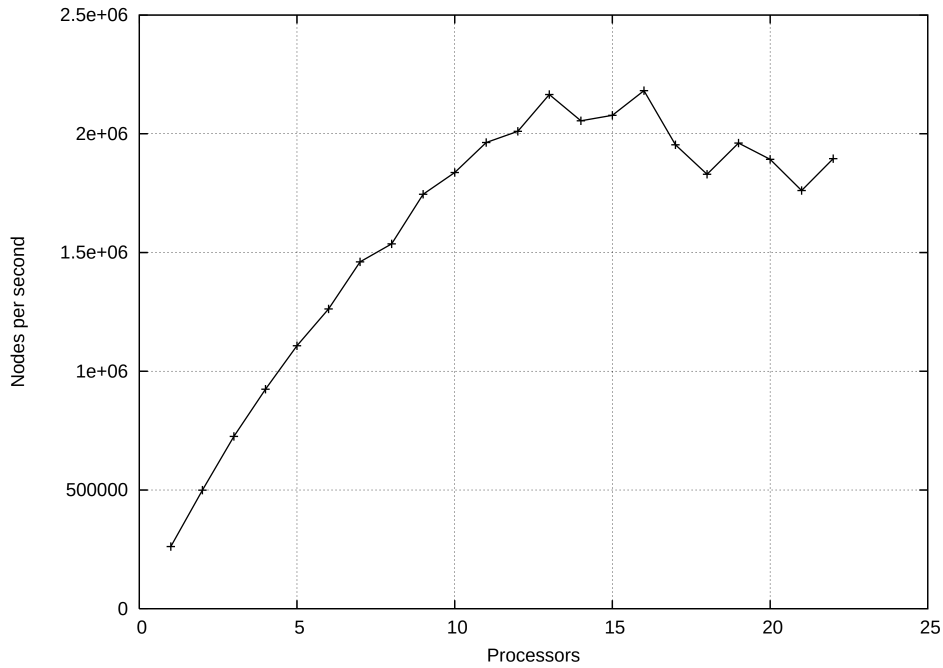

Figure 7.2: Search speed — Preliminary experiment.

experiment. This graph is smoother, but it has an earlier maximum and less than impressive performance at many processors, relatively. Because the preliminary experiment was done on a machine with faster processors, it still reaches a better speed in absolute terms.

The difference between the two experiments are these:

The main experiment was done on a machine with 68 processors of which we used at most 24. The processors used were dedicated exclusively to this experiment. The rest of the processors in the machine were used heavily by other scientific computing projects simultaneously.

In contrast, the preliminary experiment was done on a publicly available machine with 22 processors. No processors were dedicated exclusively to this experiment. All processors were used partially by other users simultaneously with the experiment, but in this case it was not for heavy duty scientific computing; instead, users were running low intensity applications such as web browsers, editors and word processors.

Initially, one would think that running on a dedicated set of processors would give a smoother graph than running on set of processors shared with other users. The graphs shows the opposite to be true in this case. A possible explanation is this: The parallel alpha-beta algorithm demands a lot of memory bandwidth, for memory copying between threads when splitting and for probing the shared hash table among other things. Even if we have a number of processors dedicated exclusively to the experiment, we cannot reserve memory bandwidth on the machine. It is not unlikely that the intensive scientific computations done on the rest of the processors (parallel inversion of huge matrices etc.) uses a lot of memory bandwidth on the machine. And while the processors used in the preliminary case were not dedicated to our experiment, processor load was low because of the nature of the applications running, and it is not unlikely that memory bandwidth use by these applications was low as well. Our guess is that the systematic drops in search speed in the main experiment (at 7 processors, 10–12 processors, 14–16 processors and 19–20 processors and maybe 23–24 processors), was due to heavy jobs running at these times, using lots of memory bandwidth.

If this is true, it suggests that we would need an exclusively dedicated machine (both processors and memory bandwidth) to do the experiment in the most accurate way. Unfortunately, the only machine we could get exclusive access to was a machine with only 12 processors. For comparison, the results of running the experiment on this machine is shown in Figure 7.3. Although we cannot know what would happen with more processors, it is clear that this graph is smoother and shows better scaling as more processors are used, than is the case with the main experiment and the preliminary experiment.

All of these three experiments have drawbacks: The preliminary experiment was only done on 22 processors and neither processors or memory bandwidth were dedicated exclusively to the experiment. The main experiment was done on 24 dedicated processors, but on a machine with a lot of other work was done on other processors by other people, limiting the memory bandwidth of the machine. In the last experiment, processors and memory bandwidth was exclusively used for the experiment, but the machine had only 12 processors. Ideally we would have liked to run the experiment

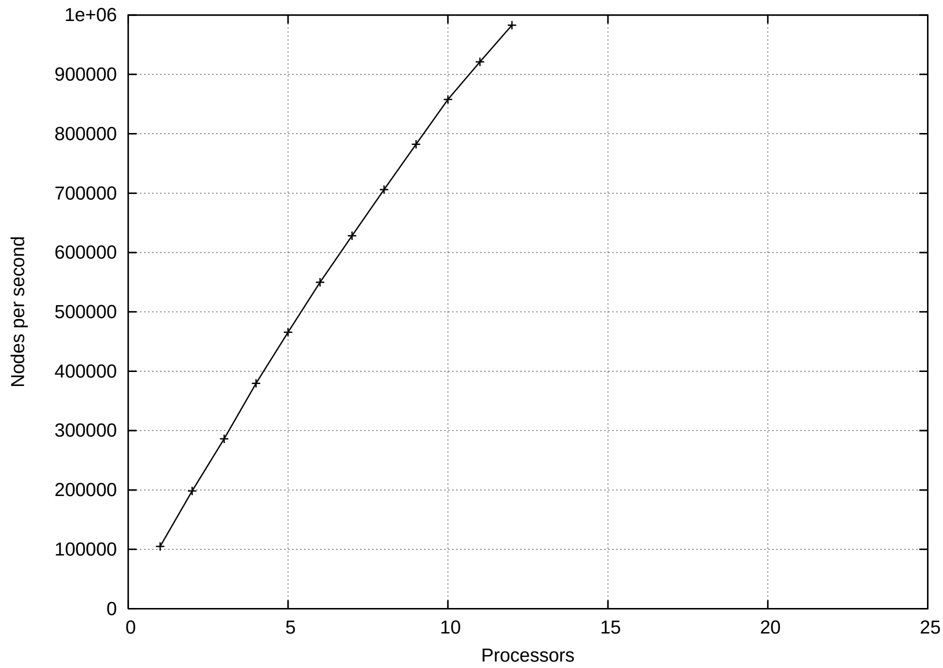

Figure 7.3: Search speed — 12 processors experiment.

on an exclusively dedicated machine and with at least 24 processors.

To verify the above memory bandwidth explanation, we could do another experiment: on an exclusively dedicated machine of 48 processors, we could run the usual experiment on 1–24 processors while periodically running other jobs using memory bandwidth on the remaining 24 processors. If the same periodic pattern emerges as in the main experiment, it would indicate that memory bandwidth is really the issue.

If we assume that sharing a machine with other people can never actually help search speed, only hurt it, we could remove the measurements that seem to be affected, to get a better idea of the general trend. There are several ways of doing this. In Figure 7.4 we have removed the measurements that are lower than some earlier measurement (on fewer processors). These are the measurements at 7, 10, 11, 12, 14, 15, 16, 19, 20, 23 and 24 processors.

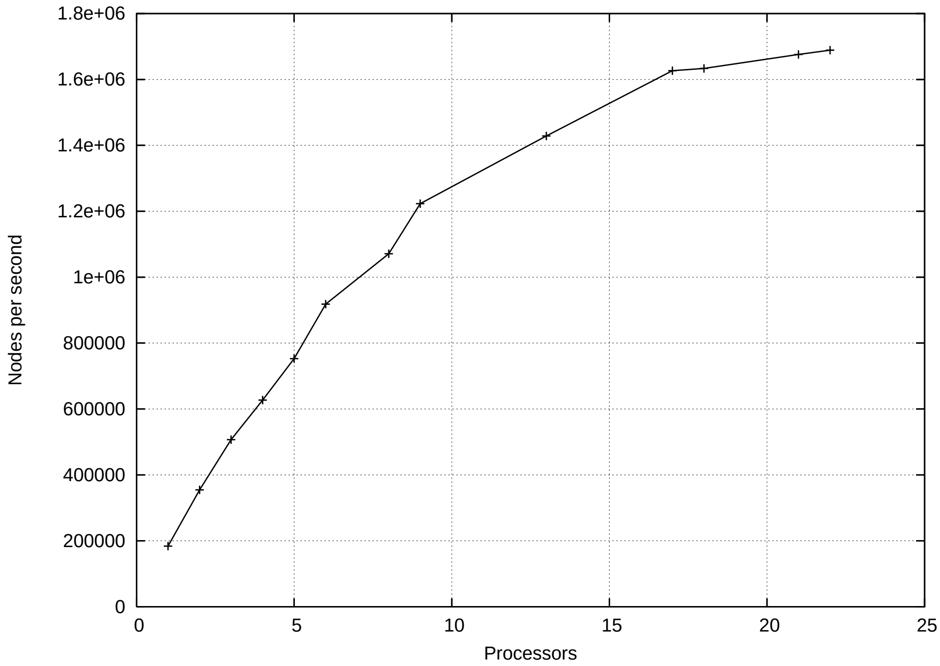

Figure 7.4: Search speed — Main experiment (general trend).

7.2.2 Speedup

One typical way of assessing the success of a parallel algorithm is to look at the speedup gained by using more and more processors, compared to the base speed of the first processor. Using n processors, we can at most hope for a speedup of n, giving ideal speedup. In practice, most parallel algorithms have sub-linear speedup, and this also true in our case. Figures 7.5, 7.6 and 7.7 show the speedup for the main experiment, the preliminary experiment and the 12 processors experiment respectively.

We have plotted the function $y = x$ in dashed style so that we can see how far we are from ideal speedup. Essentially, the speedup measurements are just the search speed measurements divided by the search speed at one processor, a constant. So the same systematical drops are of course present here too.

One important thing that these graphs show is that even at 12 processors the main experiment and the preliminary experiment shows poorer speedup than the experiment on the dedicated 12 processor machine. In fact, we get a

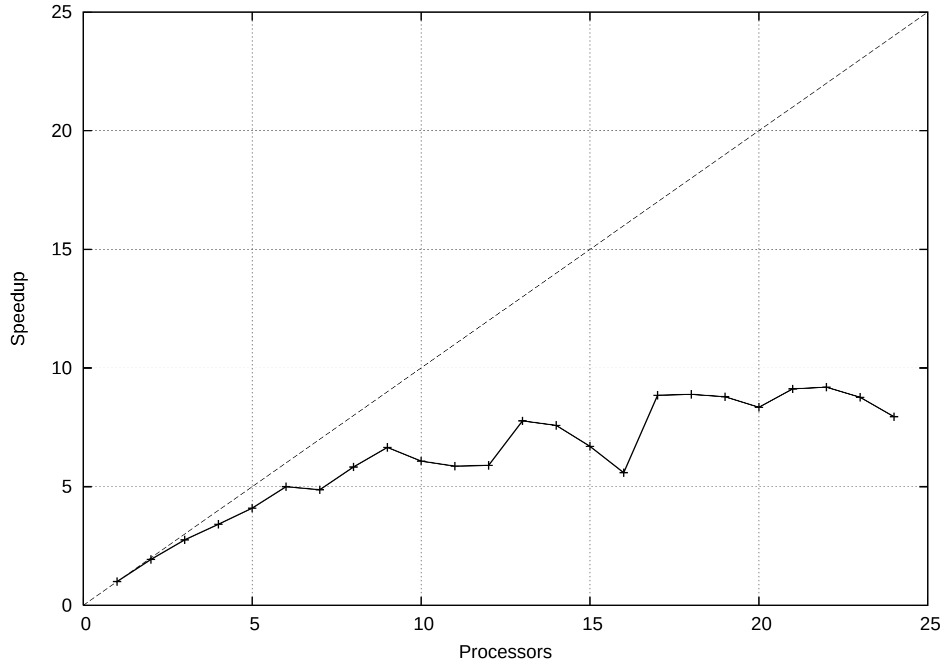

Figure 7.5: Speedup — Main experiment.

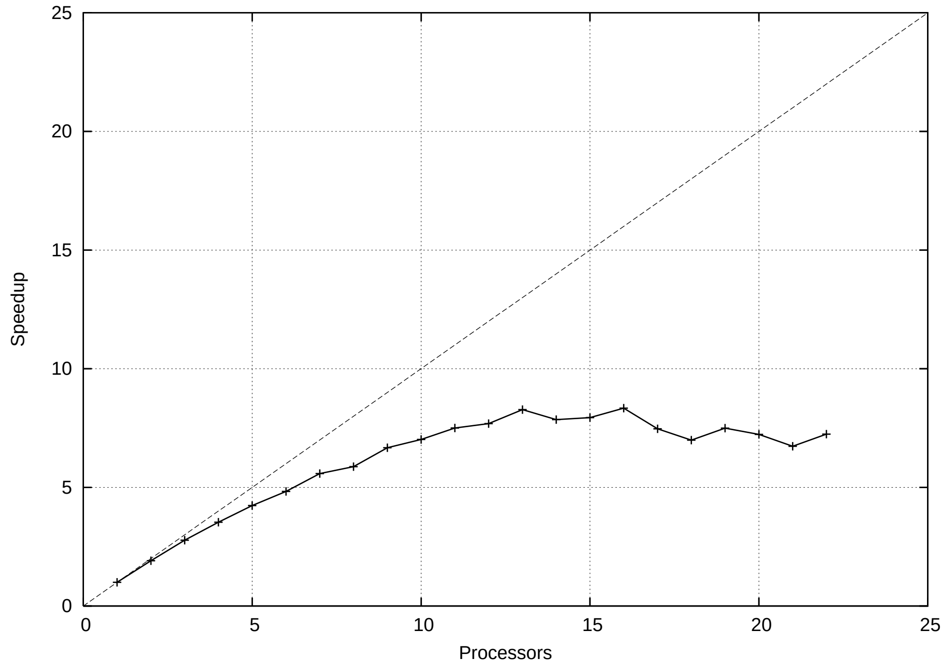

Figure 7.6: Speedup — Preliminary experiment.

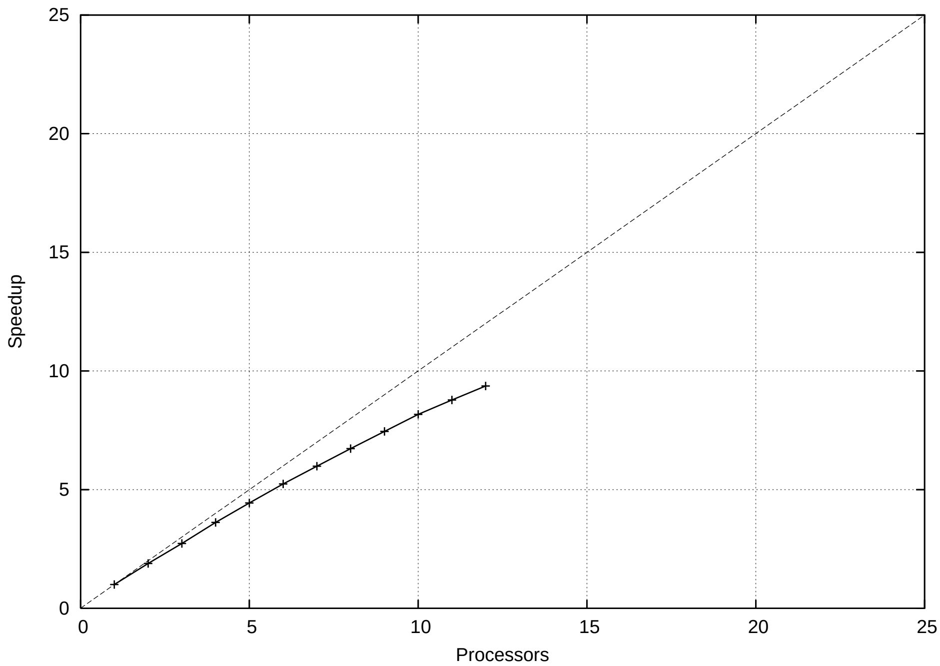

Figure 7.7: Speedup — 12 processors experiment.

better speedup at 12 processors on this machine than we do using any number of processors in the other two experiments. 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 12 processors experiment.

This suggests again that better performance could have been achieved if we had done the full 24 processor main experiment on a truly dedicated machine.

We also note that in the main experiment, speedup on many processors is far from ideal. On the other hand, at least we do get the maximum speedup on 22 processors and not on fewer processors.

Looking at other measurements in the sections below, we will not be comparing the main experiment with the other experiments, because the other experiments gave roughly the same results as the main experiment.

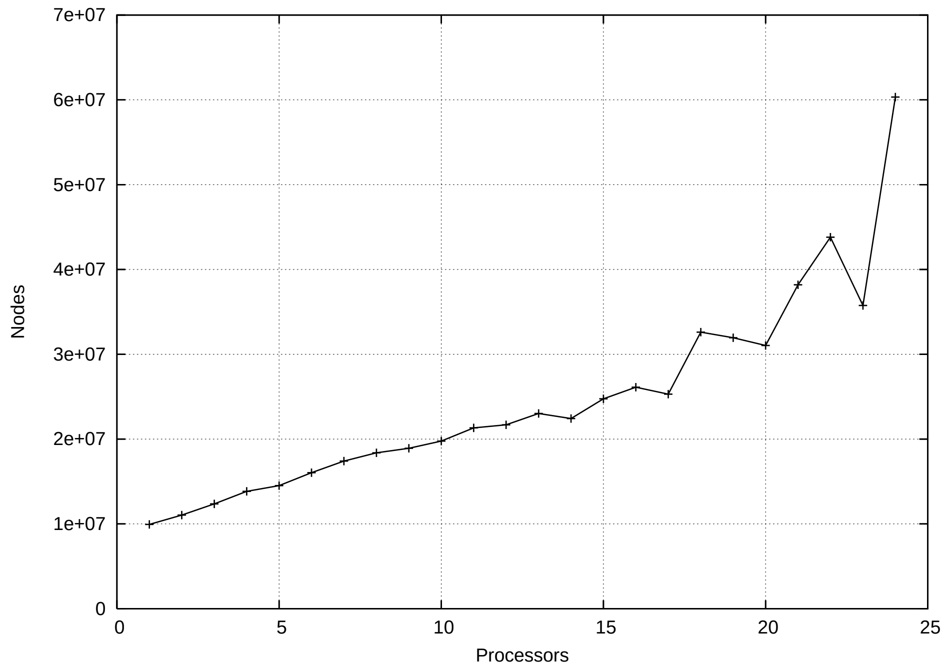

Figure 7.8: Tree size — Main experiment.

7.2.3 Tree Size

As explained earlier, the parallel alpha-beta algorithm will generally search more nodes than sequential alpha-beta, given the same position to search. Since this happens because several threads are searching a node where a cutoff occurs, we would expect that this overhead in the parallel case would increase as more processors were used. This is important because it means that while search speed increases with more processors, so does the amount of work that has to be done. Figure 7.8 shows growth of the average tree size in the main experiment.

The average tree size does indeed seem to grow as more processors are used. The growth is roughly linear except when we use more than 17 processors. It is not entirely clear what the fluctuations here mean. They do not seem to follow the pattern from the search speed graphs. Our guess is that it still might be related to the fact that the machine was not entirely dedicated to the project. Again, if we assume that this cannot help the algorithm, only hurt it, the measurements at 17, 20 and 23 processors seem to continue the same linear trend as the measurements from 15 or fewer processors. To know more about these fluctuations, we would have to make more experiments, on a truly dedicated machine.

What can be seen from the graph, and what is very important, is that the average tree size is approximately doubled at 10 processors and tripled at 20 processors, a fast rate. The average tree size at 24 processors is approximately 250% larger than at one processor. This is just as important for practical purposes as the speedup, because the actual time used to search a position will depend on both speedup and tree size. Since speedup was sub-linear and tree size growth is roughly linear, it will become infeasible at some point to use more processors.

7.2.4 Time Usage

Figure 7.9 shows the average time usage per position. The fluctuations roughly follow the ones mentioned earlier in the search speed graph. The minimum is achieved at 17 processors, but almost the same time usage is achieved already at 9 processors. This shows that the growth in tree size quickly dominates the speedup gained from using more processors.

7.2.5 Hash Hits

Figure 7.10 shows how using more processors affects the efficiency of the hash table. This is expected because more and more entries will get written to simultaneously by two or more threads, causing inconsistent entries.

The penalty of using the error detection technique is small, though. At one processor, the average hit rate was 34.7%. This means that in 34.7% of the times we probed the hash table, it contained an entry about the right position and the information stored in the table was from a deep enough search to replace the search we were about to perform. At 24 processors, this rate is 24.7%. This means that the average tree size will be about 10% larger at 24 processors than at one processor. This is only a minor increase in tree size compared to the 250% overall increase due mostly to the parallel

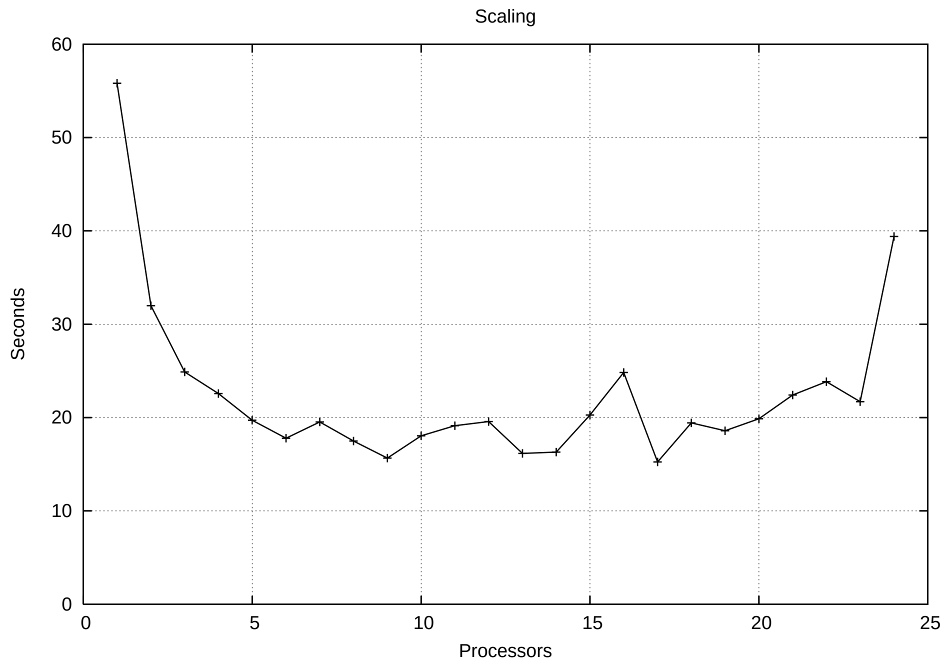

Figure 7.9: Time usage — Main experiment.

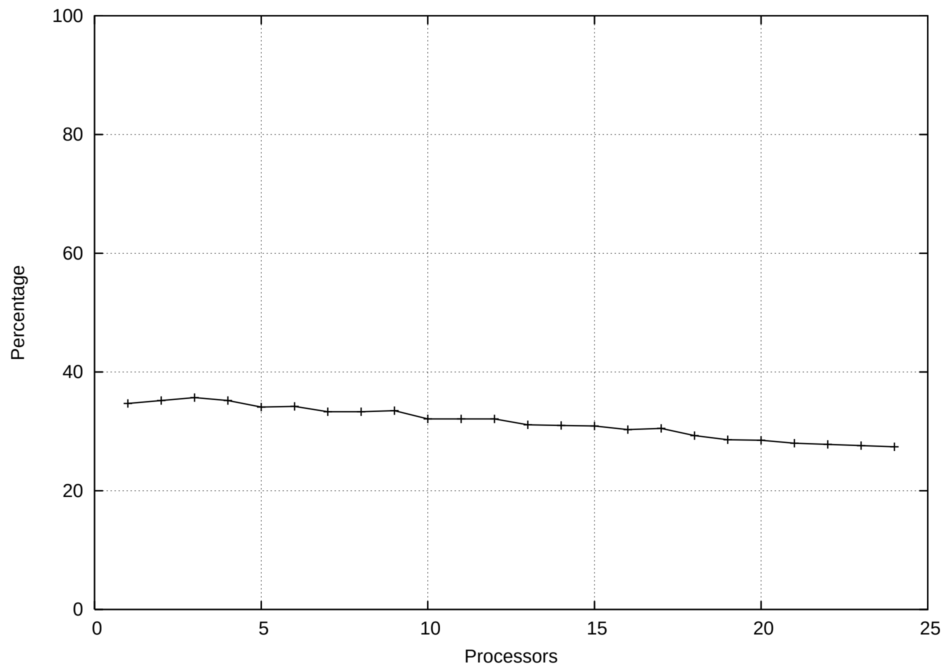

Figure 7.10: Hash hits — Main experiment.

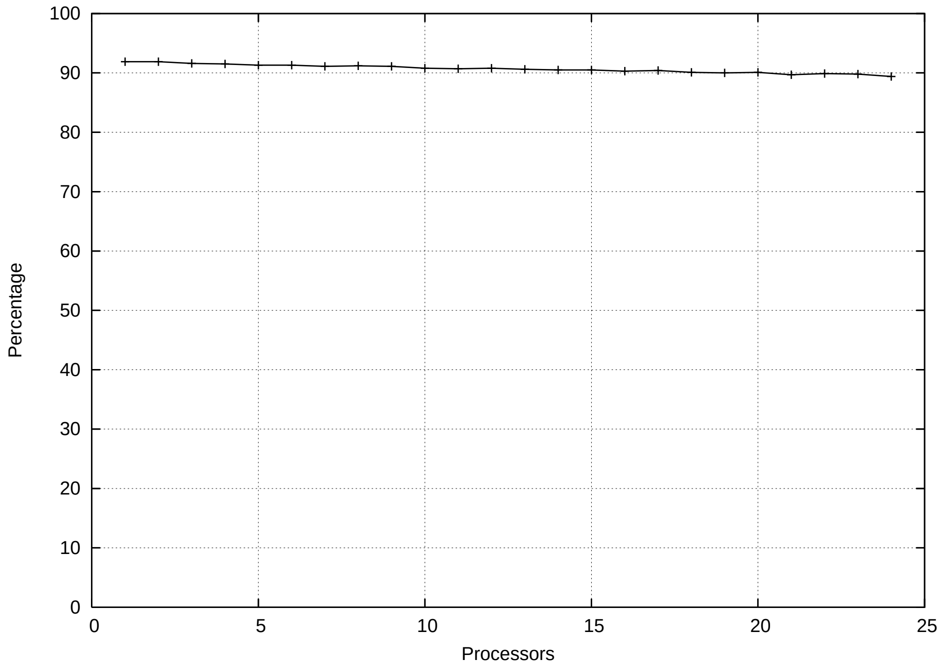

Figure 7.11: Move ordering — Main experiment.

search overhead. This suggests that the lock-free error-detection approach is feasible until the parallel search overhead is made much smaller.

7.2.6 Move Ordering

Figure 7.11 shows how the average move ordering is affected by the parallelization. In roughly 90% of the nodes where a cutoff move exists, we try this move first. With a maximum at one processor of 91.9% and a minimum at 24 processors of 89.4%, it is fair to say that move ordering is only affected very slightly. This suggests that the approach we took with the history and killer data structures when parallelizing, is practical and usable.

7.2.7 Quiescence Nodes

Figure 7.12 shows the average percentage of quiescence nodes in the trees searched. It is roughly constant around 20%. This confirms that the growth in average tree size is not due to some unforeseen explosion in the size of the

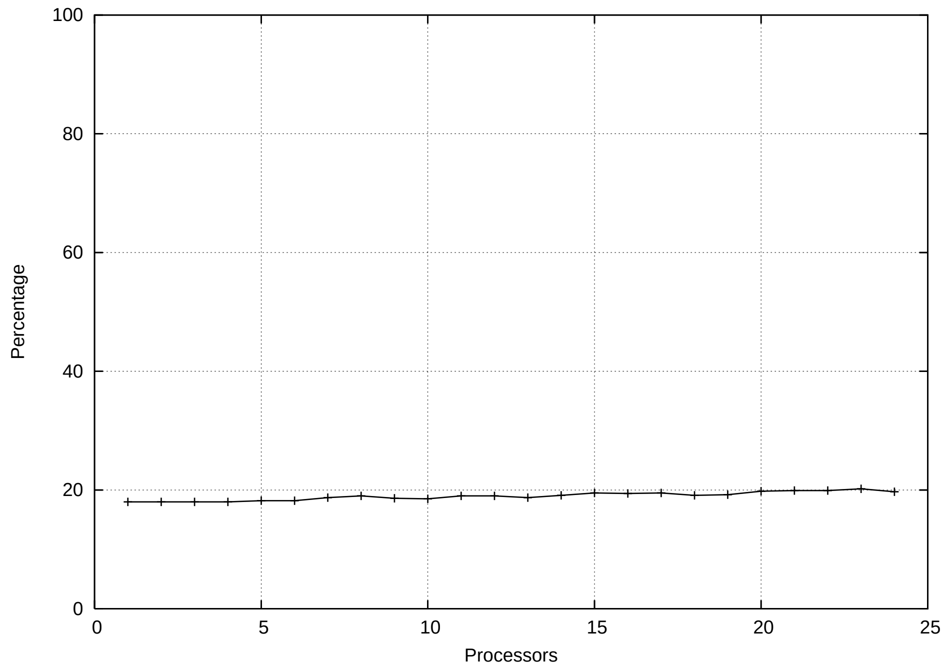

Figure 7.12: Quiescence nodes — Main experiment.

quiescence search subtrees as more processors were used.

7.3 Further Remarks

The memory transfer overhead during splitting that is caused by bitboards will affect only the raw speed of the program, not the size of the tree searched. While speedup is less than impressive when using many processors, we have still achieve a reasonable speedup using 15 processors or below. It is not entirely clear whether the lack of speedup at many processors is due to the use of bitboards. Other reasons might be limitations in the hardware architecture, the memory bandwidth, the threading library or simply the fact that the machine used was not dedicated to the project. The measurements on the dedicated 12 processor machine at least seem to show better speedup. Further experiments would be needed to find out more about this.

The hash hit rate, the move ordering quality and the quiescence nodes percentage are all major factors affecting the size of the searched tree. Two of these are roughly constant and the third is only affected slightly by the parallelization, so we attribute the major part of the tree size growth to the overhead caused by splitting type 2 (CUT) nodes. In other words, rapid tree growth seems to be inherent in this parallelization of the alpha-beta algorithm. Even if speedup had been more impressive, possibly by not using the bitboard representation, the linear tree size growth would have dominated the speed sooner or later, giving no further drop in time usage.

Chapter 8

Conclusions

We have implemented the YBWC parallelization in a chess program that uses the bitboard representation. We have used most of the modifications and enhancements common in modern chess programs to have a realistic and practical test case. We conducted one main experiment and two other experiments in different hardware settings, for comparison.

Our experiments showed that speedup was not near ideal using many processors. Whether or not this was due to the use of bitboards is unfortunately unclear. We would have liked to make more experiments to investigate this, but project time constraints did not allow this. Results from the different experiments indicated that to get reliable measurements, a totally dedicated machine is needed. It does not seem to be enough to have just the processors dedicated to the experiment. Memory bandwidth must be reserved too. Our experiments on a totally dedicated 12 processor machine showed reasonable speedup with all 12 processors: 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 experiments also showed that the tree size overhead related to the parallelization is considerable and roughly linear: The tree size is doubled at 10 processors (compared to using one processor), and tripled at 20 processors. Even if better speedup can be achieved, much effort could wisely be spent researching ways of lowering this overhead.

The combined effect of the sub-linear speedup and the linear growth in tree size in the main experiment means that the time needed to search a particular position to a particular nominal depth, is only dropping slightly when using more than 10 processors. We do, however, use the least amount of time at 17 processors.

Measurements of the hash hit rate, the move ordering quality and the quiescence search tree size (all of them major parameters affecting the overall tree size) showed that these was not affected much by the parallelization. Thus, the tree size growth seemed to mostly be an effect of the YBWC parallelization itself: too many nodes containing a cutoff are searched in parallel.

8.1 Extensions

There are some interesting statistics that were not gathered from our experiments, that it would be interesting to have. It would be relatively easy to implement and perform experiments to gather these experiments:

- We could have gathered statistics about how much time was used on splitting. These numbers would be interesting in their own right, and in relation to the total search time, or to the search time of a node; or we could calculate the ratio between search time and split time in every split node.

- The number of split nodes where a cutoff happens would be interesting in relation to the total number of split nodes, and in relation to the overall tree size.

- The number of superfluous nodes searched at each split node where a cutoff occurs would confirm in a direct way that the growth in tree size is primarily due to splitting of such nodes.

- The number of inconsistent hash entries detected would be interesting in relation to the number probes made and to the number of total entries in the table. This would help confirm that the decline in hash hit rate is due to inconsistent entries.

And of course, we would have liked to have a truly dedicated machine at our disposal for experiments.

8.2 Further Work

To explore in more detail whether or not bitboards are a practical in a parallel chess program, there are several experiments that could be done:

- Compare two parallelized chess programs. One using bitboards, and one using another representation. The programs should be as equivalent as possible except for representation. This is a relatively hard experiment to do. We would have to implement an entirely new chess program using another representation.

- Compare the performance of our parallelized bitboard chess program on several hardware architectures, especially with different memory bandwidth characteristics. This may be a hard experiment with respect to resources. Many different hardware architectures may not be available.

- Examine the effect of using more memory bandwidth during splitting. This could be done by transferring different amounts of "dummy" information during splitting, and see how search speed was affected. If we can transfer 50 more bytes than we do now, without affecting search speed considerably, it suggests that the extra transfer overhead of bitboards might not be a problem. This experiment should be relatively easy to implement and perform.

- Examine the effect of not transferring bitboards when splitting. Instead, the helper thread could itself calculate the bitboards from the simple board array. If memory bandwidth is an issue, it might actually be cheaper to recalculate than to transfer. This experiment should also be fairly easy to implement and perform. But it might require a lot of testing and tweaking to find the right ratio between calculated and transferred information.

- Examine the effect on parallel speedup of using assembly language bitboard primitives on architectures where they exist. If experience on assembly language programming (and interfacing with high level languages) on the hardware at hand is available, it should be an easy experiment.

- Examine the effect of using different threading libraries and primitives. If splitting time was reduced to zero, speedup would certainly improve. Assembly language threading primitives and locks might make a considerable difference. This should be easy to implement, but will require experience with different threading libraries and assembly language.

These experiments might not uncover the truth about bitboards in parallel chess programs, but they would most certainly help in understanding the problem better.

Appendix A

Data

A.1 Positions

The 10 positions used for the experiments are given here in standard notation (FEN).

```
2q1rr1k/3bbnnp/p2p1pp1/2pPp3/PpP1P1P1/1P2BNNP/2BQ1PRK/7R b - - id "BK03"
2kr1bnr/pbpq4/2n1pp2/3p3p/3P1P1B/2N2N1Q/PPP3PP/2KR1B1R w - - id "BK09"
3rr1k1/pp3pp1/1qn2np1/8/3p4/PP1R1P2/2P1NQPP/R1B3K1 b - - id "BK10"
2r1nrk1/p2q1ppp/bp1p4/n1pPp3/P1P1P3/2PBB1N1/4QPPP/R4RK1 w - - id "BK11"
r3r1k1/ppqb1ppp/8/4p1NQ/8/2P5/PP3PPP/R3R1K1 b - - id "BK12"
r2q1rk1/4bppp/p2p4/2pP4/3pP3/3Q4/PP1B1PPP/R3R1K1 w - - id "BK13"
r2q1rk1/1ppnbppp/p2p1nb1/3Pp3/2P1P1P1/2N2N1P/PPB1QP2/R1B2RK1 b - - id "BK17"
3rr3/2pq2pk/p2p1pnp/8/2QBPP2/1P6/P5PP/4RRK1 b - - id "BK19"
r1bqk2r/pp2bppp/2p5/3pP3/P2Q1P2/2N1B3/1PP3PP/R4RK1 b kq - id "BK23"
r2qnrnk/p2b2b1/1p1p2pp/2pPpp2/1PP1P3/PRNBB3/3QNPPP/5RK1 w - - id "BK24"
```

A.2 Results

In Figure A.1, A.2 and A.3 is shown all of the collected data from the three experiments discussed. The eight columns of the tables contain the number of processors, search speed, tree size, search time, hash hit rate, move ordering and quiescence nodes percentage respectively. All of these are averages over all 10 positions, except of course column 1 and 3 (CPUs and Speedup).

CPUs	NPS	Speedup	Nodes	Time	Hash	Order	Qn
1	183788.6	1.000	9944716.9	55.808	34.7	91.9	18.0
2	354635.9	1.930	11035105.8	31.989	35.2	91.9	18.0
3	506924.0	2.758	12355723.8	24.884	35.7	91.6	18.0
4	627200.3	3.413	13834612.9	22.572	35.2	91.5	18.0
5	752641.6	4.095	14505082.6	19.718	34.1	91.3	18.2
6	918063.8	4.995	16036321.2	17.792	34.2	91.3	18.2
7	894707.3	4.868	17400039.4	19.529	33.3	91.1	18.7
8	1071176.5	5.828	18385655.9	17.482	33.3	91.2	19.0
9	1223009.4	6.654	18912869.1	15.658	33.5	91.1	18.6
10	1116726.9	6.076	19763484.5	18.048	32.1	90.8	18.5
11	1077282.7	5.862	21325735.7	19.132	32.1	90.7	19.0
12	1084126.6	5.899	21689951.2	19.578	32.1	90.8	19.0
13	1428279.8	7.771	23015273.8	16.152	31.1	90.6	18.7
14	1393070.8	7.580	22414490.7	16.303	31.0	90.5	19.1
15	1231179.7	6.699	24741002.4	20.265	30.9	90.5	19.5
16	1026975.4	5.588	26116843.8	24.822	30.3	90.3	19.4
17	1626518.5	8.850	25292034.3	15.240	30.5	90.4	19.5
18	1633365.0	8.887	32615379.9	19.434	29.3	90.1	19.1
19	1615603.8	8.791	31944211.0	18.585	28.6	90.0	19.2
20	1534064.5	8.347	31035152.2	19.865	28.5	90.1	19.8
21	1675977.3	9.119	38191994.3	22.418	28.0	89.7	19.9
22	1688979.4	9.190	43804388.8	23.840	27.8	89.9	19.9
23	1611493.2	8.768	35752776.6	21.704	27.6	89.8	20.2
24	1460198.5	7.945	60330499.7	39.401	27.4	89.4	19.7

Figure A.1: Main experiment.

CPUs	NPS	Speedup	Nodes	Time	Hash	Order	Qn
1	261581.2	1.000	9406997.9	37.082	28.4	91.4	17.7
2	499379.7	1.909	11532789.6	23.809	29.0	91.1	18.1
3	725614.4	2.774	12518008.9	17.822	28.3	91.1	18.2
4	924121.0	3.533	14030847.1	15.632	27.3	90.9	18.5
5	1107147.5	4.233	13934632.1	12.897	27.8	91.1	17.7
6	1262521.9	4.827	16015514.1	12.792	27.0	90.9	17.6
7	1460324.3	5.583	17835364.1	12.453	26.9	90.5	18.1
8	1536594.2	5.874	18083099.2	11.918	26.0	90.6	17.8
9	1745540.1	6.673	20267343.1	11.871	25.9	90.6	18.2
10	1836671.2	7.021	20688437.1	11.539	25.4	90.4	18.2
11	1963157.1	7.505	25600183.9	12.957	24.8	90.2	18.3
12	2010524.4	7.686	24955506.7	12.532	24.7	90.3	18.3
13	2165128.2	8.277	23806082.6	10.957	24.3	90.2	18.2
14	2054958.8	7.856	24417188.9	11.906	24.3	90.2	18.5
15	2077675.7	7.943	28147756.5	13.413	24.1	90.1	18.1
16	2181676.9	8.340	26484931.7	11.765	23.9	90.1	19.0
17	1953368.7	7.468	25102096.7	12.428	23.6	89.9	18.9
18	1829261.6	6.993	25323823.5	13.519	23.7	89.9	18.7
19	1961120.2	7.497	30313409.5	15.530	24.0	90.0	19.0
20	1892223.5	7.234	36279246.7	18.216	22.7	89.5	19.1
21	1761243.7	6.733	38705262.0	20.462	22.4	89.8	19.1
22	1895451.8	7.246	43013644.6	21.676	22.1	89.5	19.1

Figure A.2: Preliminary experiment.

CPUs	NPS	Speedup	Nodes	Time	Hash	Order	Qn
1	104965.5	1.000	9851018.1	97.005	31.6	91.7	17.9
2	198286.1	1.889	11410269.0	59.345	31.3	91.5	18.1
3	286226.5	2.727	11139063.0	39.430	31.1	91.5	18.2
4	379699.0	3.617	11533147.4	31.101	30.7	91.3	18.2
5	465639.0	4.436	14960107.8	32.690	30.3	91.0	18.6
6	549917.9	5.239	15527492.3	28.675	29.9	91.1	18.2
7	628227.2	5.985	16114517.8	26.082	29.2	91.0	18.5
8	706065.8	6.727	14714524.6	21.324	29.6	90.9	18.1
9	782236.1	7.452	18322282.0	23.638	28.9	91.0	18.8
10	857783.0	8.172	18819848.6	22.091	28.5	90.5	18.6
11	921175.3	8.776	19058333.1	20.908	28.6	90.5	18.5
12	982922.2	9.364	19765489.6	20.157	27.4	90.3	18.8

Figure A.3: 12 processor experiment.

Appendix B

Source Code

B.1 bitboard.cpp

```
// $Id: bitboard.cpp,v 1.5 2004/02/24 12:09:20 s958547 Exp $

#include <iostream>

#include "base.h"
#include "bitboard.h"

using namespace std;

int firstBit_[65536];
BitBoard mask_[Squareend];

void initBitBoard()
{
  // Fill the firstBit table
  firstBit_[0] = -1;
  for (unsigned int i = 1; i < 65536; ++i)
  {
    int count = 0;
    int j = i;
    while (!(j & 0x0001))
    {
      j = j >> 1;
      ++count;
    }
    firstBit_[i] = count;
  }
```

```
  forall (Square,s)
    mask_[s] = BitBoard(1ULL << s);
}

ostream& operator<<(ostream& os, const BitBoard b)
{
  os << "   +---+---+---+---+---+---+---+---+" << newline;
  for (Rank r = rank8; r >= rank1; --r)
  {
    os << " " << int(r+1) << " |";
    for (File f = fileA; f <= fileH; ++f)
      if (mask(toSquare(f,r)) & b)
        os << " 1 |";
      else
        os << "   |";
    os << newline;
    os << "   +---+---+---+---+---+---+---+---+" << newline;
  }
  os << "     a   b   c   d   e   f   g   h" << newline;
  return os;
}
```

B.2 evaluate.cpp

```
// $Id: evaluate.cpp,v 1.7 2004/07/26 14:26:10 s958547 Exp $

#include <iostream>

#include "evaluate.h"
#include "base.h"

using namespace std;

const int pieceSquareTable_[Pieceend][Squareend] =
{
  // none
  {},
  // pawn
  {
        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,   75,   75,    0,    0,    0,
        0,    0,   15,   75,   75,   15,    0,    0,
        0,    0,    0,   40,   40,    0,    0,    0,
        0,    0,    0, -100, -100,    0,    0,    0,
        0,    0,    0,    0,    0,    0,    0,    0
```

```
},
// knight
{
   -120, -120, -120, -120, -120, -120, -120, -120,
   -120,   25,   25,   25,   25,   25,   25, -120,
   -120,   25,   50,   50,   50,   50,   25, -120,
   -120,   25,   50,  100,  100,   50,   25, -120,
   -120,   25,   50,  100,  100,   50,   25, -120,
   -120,   25,   50,   50,   50,   50,   25, -120,
   -120,   25,   25,   25,   25,   25,   25, -120,
   -120, -120, -120, -120, -120, -120, -120, -120
},
// bishop
{
    -40,  -40,  -40,  -40,  -40,  -40,  -40,  -40,
    -40,   20,   20,   20,   20,   20,   20,  -40,
    -40,   20,   30,   30,   30,   30,   20,  -40,
    -40,   20,   30,   45,   45,   30,   20,  -40,
    -40,   20,   30,   45,   45,   30,   20,  -40,
    -40,   20,   30,   30,   30,   30,   20,  -40,
    -40,   20,   20,   20,   20,   20,   20,  -40,
    -40,  -40,  -40,  -40,  -40,  -40,  -40,  -40
},
// rook
{
      0,    0,   10,   15,   15,   10,    0,    0,
      0,    0,   10,   15,   15,   10,    0,    0,
      0,    0,   10,   15,   15,   10,    0,    0,
      0,    0,   10,   15,   15,   10,    0,    0,
      0,    0,   10,   15,   15,   10,    0,    0,
      0,    0,   10,   15,   15,   10,    0,    0,
      0,    0,   10,   15,   15,   10,    0,    0,
      0,    0,   10,   15,   15,   10,    0,    0
},
// queen
{
      0,    0,    0,    0,    0,    0,    0,    0,
      0,    0,    0,    0,    0,    0,    0,    0,
      0,    0,   75,   75,   75,   75,    0,    0,
      0,    0,   75,  100,  100,   75,    0,    0,
      0,    0,   75,  100,  100,   75,    0,    0,
      0,    0,   75,   75,   75,   75,    0,    0,
      0,    0,    0,    0,    0,    0,    0,    0,
      0,    0,    0,    0,    0,    0,    0,    0
},
// king
{
   -900, -900, -900, -900, -900, -900, -900, -900,
   -900, -900, -900, -900, -900, -900, -900, -900,
```

```
    -900, -900, -900, -900, -900, -900, -900, -900,
    -900, -900, -900, -900, -900, -900, -900, -900,
    -900, -900, -900, -900, -900, -900, -900, -900,
    -700, -700, -700, -700, -700, -700, -700, -700,
    -200, -200, -500, -500, -500, -500, -200, -200,
     200,  300,  100, -300,  300,  100,  300,  200
   }
  // king (endgame)
  /*
  {
    0,   30,   50,  200,  200,   50,   30,    0,
   30,   50,  100,  300,  300,  100,   50,   30,
   50,  100,  200,  400,  400,  200,  100,   50,
  200,  300,  400,  500,  500,  400,  300,  200,
  200,  300,  400,  500,  500,  400,  300,  200,
   50,  100,  200,  400,  400,  200,  100,   50,
   30,   50,  100,  300,  300,  100,   50,   30,
    0,   30,   50,  200,  200,   50,   30,    0
  }
  */
};

Score pieceSquareTable(Piece p, Square s)
{
  return Score(pieceSquareTable_[p][s]);
}

Score pieceSquareTables(const Position& pos)
{
  Score score = Score(0);

  forall (Color,c)
  {
    BitBoard pieces = pos.occupied(c);
    while (pieces)
    {
      Square s = firstBit(pieces);
      Square rs =
        c == white ? Square((7 - getRank(s))*8 + getFile(s)) : s;
      if (c == white)
        score += pieceSquareTable(pos[s].piece(),rs);
      else
        score -= pieceSquareTable(pos[s].piece(),rs);
      clear(pieces,s);
    }
  }

  return score;
}
```

```
Score evaluate(const Position& pos)
{
  Score score = Score(pos.material(white) - pos.material(black));

  score += pieceSquareTables(pos);

  if (pos.turn() == white)
    return Score(score);
  else
    return Score(-score);
}
```

B.3 hash.cpp

```
// $Id: hash.cpp,v 1.10 2004/07/26 14:26:10 s958547 Exp $

#include <algorithm>

#include "hash.h"
#include "random.h"

using namespace std;

HashKey hashPiece_[Colorend][Pieceend][Squareend];
HashKey hashEpSquare_[Squareend];
HashKey hashCastling_[Colorend][CastlingRightend];
HashKey hashColor_[Colorend];

void initHash()
{
  initRandom(68);

  forall (Color,c)
    forall (Piece,p)
      forall (Square,s)
        if (p == none)
          hashPiece_[c][p][s] = 0;
        else
          hashPiece_[c][p][s] = random64();

  forall (Square,s)
    hashEpSquare_[s] = random64();

  forall (Color,c)
  {
    forall (CastlingRight,cr)
      hashCastling_[c][cr] = random64();
```

```
    hashColor_[c] = random64();
  }
}

TransRefTable::TransRefTable()
{
  if (sizeof(HashEntry) != 16)
  {
    std::cout << "sizeof(HashEntry) == " << sizeof(HashEntry) << " != 16."
              << newline;
  }

  hashEnabled = true;
//  hashEnabled = false;

  hashTableSize = ((1024*1024*16) / sizeof(HashEntry)) / 2;
  hashTableSplit = int(hashTableSize * (33/double(100)));
  hashTableWhite = new HashEntry[hashTableSize];
  hashTableBlack = new HashEntry[hashTableSize];
  hashID = 1;

  clear();
}

HashEntryType TransRefTable::probe(Node& node, Depth depth, int ply,
                    Score& alpha, Score& beta, Move& bestMove)
{
  const Position& pos = node.pos;

  bestMove = nullMove;

  if (!hashEnabled)
    return noHash;

  HashEntry* hashTable;
  if (pos.turn() == white)
    hashTable = hashTableWhite;
  else
    hashTable = hashTableBlack;

  unsigned int index = int(pos.hashKey() % hashTableSplit);

  ++node.transRefProbes;

  HashKey hashKey = hashTable[index].hashKey ^ hashTable[index].dataul64;

  if (hashKey  == pos.hashKey()) do
  {
    bestMove = Move(hashTable[index].data.move);
```

```
  Score hashScore = Score(hashTable[index].data.score - maxScore);

  if (hashTable[index].data.depth < depth)
    break;

  ++node.transRefHits;
  switch (hashTable[index].data.type)
  {
    case exact:
      alpha = hashScore;
      if (alpha > mate - 300)
        alpha = Score(alpha - ply);
      else
        if (alpha < -mate + 300)
          alpha = Score(alpha + ply);
      return exact;
    case lowerBound:
      if (hashScore >= beta)
      {
        beta = hashScore;
        return lowerBound;
      }
      break;
    case upperBound:
      if (hashScore <= alpha)
      {
        alpha = hashScore;
        return upperBound;
      }
      break;
  }
} while (false);

index = int(pos.hashKey() % (hashTableSize - hashTableSplit))
  + hashTableSplit;

hashKey = hashTable[index].hashKey ^ hashTable[index].dataul64;

if (hashKey == pos.hashKey()) do
{
  bestMove = Move(hashTable[index].data.move);

  Score hashScore = Score(hashTable[index].data.score - maxScore);

  if (hashTable[index].data.depth < depth)
    break;

  ++node.transRefHits;
```

```
    switch (hashTable[index].data.type)
    {
      case exact:
        alpha = hashScore;
        if (alpha > mate - 300)
          alpha = Score(alpha - ply);
        else
          if (alpha < -mate + 300)
            alpha = Score(alpha + ply);
        return exact;
      case lowerBound:
        if (hashScore >= beta)
        {
          beta = hashScore;
          return lowerBound;
        }
        break;
      case upperBound:
        if (hashScore <= alpha)
        {
          alpha = hashScore;
          return upperBound;
        }
        break;
    }
  } while (false);

  return noHash;
}

void TransRefTable::store(const Position &pos, Depth depth, int ply,
                          HashEntryType type, Score score, Move bestMove)
{
  if (!hashEnabled)
    return;

  HashEntry* hashTable;
  if (pos.turn() == white)
    hashTable = hashTableWhite;
  else
    hashTable = hashTableBlack;

  unsigned int index = pos.hashKey() % hashTableSplit;

  unsigned long int age = hashTable[index].data.age;
  age = age && (hashID != age);
  if (age || (depth > hashTable[index].data.depth) || score == illegal)
  {
    unsigned int splitIndex =
```

```
      static_cast<unsigned int>
      (hashTable[index].hashKey % (hashTableSize - hashTableSplit))
      + hashTableSplit;
    hashTable[splitIndex] = hashTable[index];
  }
  else
    index = int(pos.hashKey() % (hashTableSize - hashTableSplit))
      + hashTableSplit;

  hashTable[index].data.age = hashID;
  hashTable[index].data.move = bestMove.representation();
  hashTable[index].data.type = type;
  hashTable[index].data.depth = depth;
  if (score == illegal)
  {
    hashTable[index].data.score = score + maxScore;
    hashTable[index].hashKey = pos.hashKey() ^ hashTable[index].dataul64;
    return;
  }

  Score hashScore;
  switch(type)
  {
    case exact:
      if (score > mate - 300)
        hashScore = Score(score + ply);
      else
        if (score < -mate + 300)
          hashScore = Score(score - ply);
        else
          hashScore = score;
      break;

    case upperBound:
      hashScore = max(score,Score(-mate + 300));
      break;

    case lowerBound:
      hashScore = min(score,Score(mate - 300));
      break;

    default:
      hashScore = illegal;
      cout << "HashStore: type is illegal." << endl;
  }
  hashTable[index].data.score = hashScore + maxScore;

  hashTable[index].hashKey = pos.hashKey()  ^ hashTable[index].dataul64;
}
```

```
void TransRefTable::clear()
{
  for (int i = 0; i < hashTableSize; ++i)
  {
    hashTableWhite[i].data.age = 0;
    hashTableWhite[i].data.depth = 0;
    hashTableWhite[i].hashKey = 0;
    hashTableWhite[i].data.move = 0;
    hashTableWhite[i].data.score = 0;
    hashTableWhite[i].data.type = 0;

    hashTableBlack[i].data.age = 0;
    hashTableBlack[i].data.depth = 0;
    hashTableBlack[i].hashKey = 0;
    hashTableBlack[i].data.move = 0;
    hashTableBlack[i].data.score = 0;
    hashTableBlack[i].data.type = 0;
  }
}

HashKey hashPiece(Color c, Piece p, Square s)
{
  return hashPiece_[c][p][s];
}

HashKey hashEpSquare(Square s)
{
  return hashEpSquare_[s];
}

HashKey hashCastling(Color c, CastlingRight cr)
{
  return hashCastling_[c][cr];
}

HashKey calcHashKey(const Position& pos)
{
  HashKey result = 0;

  forall (Square,s)
    result ^= hashPiece(pos[s].color(),pos[s].piece(),s);

  result ^= hashEpSquare(pos.epSquare());

  result ^= hashCastling(white,pos.castlingRight(white));

  result ^= hashCastling(black,pos.castlingRight(black));
```

```
  return result;
}
```

B.4 main.cpp

```
// $Id: main.cpp,v 1.27 2004/06/18 11:54:25 s958547 Exp $

#include <pthread.h>

#include <iostream>
#include <vector>
#include <sstream>
#include <fstream>

#include "moves.h"
#include "position.h"
#include "evaluate.h"
#include "search.h"
#include "hash.h"
#include "parallel.h"

using namespace std;

bool done;
Color computer = black;
bool force = true;

vector<string> tokenize(const string& source, const string& delimeters)
{
  vector<string> tokens;

  string::size_type start = 0;
  string::size_type end = 0;

  while ((start = source.find_first_not_of(delimeters, start)) !=
         string::npos)
  {
    end = source.find_first_of(delimeters, start);
    tokens.push_back(source.substr(start, end - start));
    start = end;
  }

  return tokens;
}

void processInput(Position& pos)
{
  cout << (pos.turn() == white ? "white: " : "black: ");
```

```
string line;
getline(cin,line);
vector<string> tokens = tokenize(line," \n\t");
if (tokens.empty())
  return;
if (tokens[0] == "!")
  cout << pos << newline;
else if (tokens[0] == "quit")
  done = true;
else if (tokens[0] == "perft")
{
  if (tokens.size() > 1)
  {
    int d;
    stringstream ss;
    ss.str(tokens[1]);
    ss >> d;
    perft(pos,d);
  }
}
else if (tokens[0] == "moves")
{
  Moves mvs;
  generateMoves(pos,mvs);
  mvs.removeIllegal(pos);
  cout << "Generated " << mvs.size() << " moves:" << newline;
  for (int i = 0; i < mvs.size(); ++i)
  {
    cout << i+1 << ": " << mvs[i] << " ";
    if (((i + 1) % 8) == 0)
      cout << newline;

  }
  cout << newline;
}
else if (tokens[0] == "setboard")
{
  string FEN;
  for (int i = 1; i < tokens.size(); ++i)
    FEN += tokens[i]+" ";
  pos.setupFEN(FEN);
}
else if (tokens[0] == "openfen")
{
  if (tokens.size() > 1)
  {
    ifstream in;
    in.open(tokens[1].c_str());
    if (!in)
```

```
    {
      cout << "Couldn't open '" << tokens[1] << "'" << newline;
    }
    else
    {
      string fen;
      getline(in,fen);
      pos.setupFEN(fen);
    }
    in.close();
  }
}
else if (tokens[0] == "go")
{
  computer = pos.turn();
  force = false;
}
else if (tokens[0] == "force")
{
  force = true;
}
else if (tokens[0] == "eval")
{
  cout << "eval: " << evaluate(pos) << endl;
}
else if (tokens[0] == "see")
{
  if (tokens.size() > 1)
  {
    stringstream ss(tokens[1]);
    int no = -1;
    ss >> no;
    if (no != -1)
    {
      Moves mvs;
      generateMoves(pos,mvs);
      mvs.removeIllegal(pos);

      if (no > 0 && no <= mvs.size())
      {
        int i = no - 1;
        cout << mvs[i] << newline;
        cout << SEE(pos,mvs[i]) << endl;
        return;
      }
    }
  }
}
else // Move...
```

```
  {
    Moves mvs;
    generateMoves(pos,mvs);
    mvs.removeIllegal(pos);

    stringstream ss(tokens[0]);
    int no = -1;
    ss >> no;
    if (no != -1)
    {
      if (no > 0 && no <= mvs.size())
      {
        int i = no - 1;
        cout << mvs[i] << newline;
        pos.makeMove(mvs[i]);
        ++pos.rootPly_;
        return;
      }
    }

    File f1 = File(tokens[0][0] - 'a');
    Rank r1 = Rank(tokens[0][1] - '1');
    File f2 = File(tokens[0][2] - 'a');
    Rank r2 = Rank(tokens[0][3] - '1');
    Square from = toSquare(f1,r1);
    Square to = toSquare(f2,r2);

    bool foundMove = false;
    for (int i = 0; i < mvs.size(); ++i)
      if (mvs[i].from() == from && mvs[i].to() == to)
      {
        foundMove = true;
        cout << mvs[i] << newline;
        pos.makeMove(mvs[i]);
        ++pos.rootPly_;
        break;
      }
    if (!foundMove)
      cout << "Illegal move or command..." << newline;
  }
}

int main(int argc, char* argv[])
{
  system("uname -a");

#if !defined NDEBUG
  cout << "!!! DEBUG !!!" << newline;
#endif
```

```
string fen;
bool batchMode = false;
if (argc > 1)
{
  if (argc != 3)
  {
    cout << "Wrong number of arguments..." << endl;
    exit(0);
  }

  ifstream in;
  in.open(argv[1]);
  if (!in)
    cout << "Couldn't open '" << argv[1] << endl;
  else
  {
    cout << "Reading '" << argv[1] << "'" << endl;
    getline(in,fen);
    force = false;
    batchMode = true;
  }
  in.close();

  stringstream ss(argv[2]);
  int t;
  ss >> t;
  if (t < 1 || t > maxThreads)
  {
    cout << "Illegal number of threads..." << endl;
    exit(0);
  }
  nThreads = t;
  maxThreadsPerNode = t;
}

initBitBoard();
initMoves();
initHash();
initParallel();

Node& rootNode = *rNode;
Position& pos = rootNode.pos;

if (batchMode)
{
  cout << "Setting up FEN: " << fen << endl;
  pos.setupFEN(fen);
  computer = pos.turn();
```

```
  }

  done = false;
  while (!done)
  {
    if (pos.turn() == computer && !force)
      think(rootNode);
    else
      processInput(pos);
    if (batchMode)
      break;
  }
}
```

B.5 moves.cpp

```
// $Id: moves.cpp,v 1.17 2004/07/26 14:26:10 s958547 Exp $

#include <iostream>

#include "base.h"
#include "moves.h"

using namespace std;

const Square unrotate45Left_[consts::squares] =
{
  a1,
  a2,b1,
  a3,b2,c1,
  a4,b3,c2,d1,
  a5,b4,c3,d2,e1,
  a6,b5,c4,d3,e2,f1,
  a7,b6,c5,d4,e3,f2,g1,
  a8,b7,c6,d5,e4,f3,g2,h1,
  b8,c7,d6,e5,f4,g3,h2,
  c8,d7,e6,f5,g4,h3,
  d8,e7,f6,g5,h4,
  e8,f7,g6,h5,
  f8,g7,h6,
  g8,h7,
  h8
};

const Square unrotate45Right_[consts::squares] =
{
  h1,
  g1,h2,
```

```
  f1,g2,h3,
  e1,f2,g3,h4,
  d1,e2,f3,g4,h5,
  c1,d2,e3,f4,g5,h6,
  b1,c2,d3,e4,f5,g6,h7,
  a1,b2,c3,d4,e5,f6,g7,h8,
  a2,b3,c4,d5,e6,f7,g8,
  a3,b4,c5,d6,e7,f8,
  a4,b5,c6,d7,e8,
  a5,b6,c7,d8,
  a6,b7,c8,
  a7,b8,
  a8
};

const Square unrotate90Left_[consts::squares] =
{
  a8,a7,a6,a5,a4,a3,a2,a1,
  b8,b7,b6,b5,b4,b3,b2,b1,
  c8,c7,c6,c5,c4,c3,c2,c1,
  d8,d7,d6,d5,d4,d3,d2,d1,
  e8,e7,e6,e5,e4,e3,e2,e1,
  f8,f7,f6,f5,f4,f3,f2,f1,
  g8,g7,g6,g5,g4,g3,g2,g1,
  h8,h7,h6,h5,h4,h3,h2,h1
};

const int rankShift_[consts::squares] =
{
   0, 0, 0, 0, 0, 0, 0, 0,
   8, 8, 8, 8, 8, 8, 8, 8,
  16,16,16,16,16,16,16,16,
  24,24,24,24,24,24,24,24,
  32,32,32,32,32,32,32,32,
  40,40,40,40,40,40,40,40,
  48,48,48,48,48,48,48,48,
  56,56,56,56,56,56,56,56
};

const int fileShift_[consts::squares] =
{
  0, 8,16,24,32,40,48,56,
  0, 8,16,24,32,40,48,56,
  0, 8,16,24,32,40,48,56,
  0, 8,16,24,32,40,48,56,
  0, 8,16,24,32,40,48,56,
  0, 8,16,24,32,40,48,56,
  0, 8,16,24,32,40,48,56,
  0, 8,16,24,32,40,48,56
```

```
};

const int diagonalShift_a1h8_[consts::squares] =
{
  28,21,15,10, 6, 3, 1, 0,
  36,28,21,15,10, 6, 3, 1,
  43,36,28,21,15,10, 6, 3,
  49,43,36,28,21,15,10, 6,
  54,49,43,36,28,21,15,10,
  58,54,49,43,36,28,21,15,
  61,58,54,49,43,36,28,21,
  63,61,58,54,49,43,36,28
};

const int diagonalShift_h1a8_[consts::squares] =
{
   0, 1, 3, 6,10,15,21,28,
   1, 3, 6,10,15,21,28,36,
   3, 6,10,15,21,28,36,43,
   6,10,15,21,28,36,43,49,
  10,15,21,28,36,43,49,54,
  15,21,28,36,43,49,54,58,
  21,28,36,43,49,54,58,61,
  28,36,43,49,54,58,61,63
};

BitBoard pawnAttack_[consts::squares][Colorend];
BitBoard knightAttack_[consts::squares];
BitBoard kingAttack_[consts::squares];
BitBoard diagonalAttack_a1h8_[consts::squares][256];
BitBoard diagonalAttack_h1a8_[consts::squares][256];
BitBoard rankAttack_[consts::squares][256];
BitBoard fileAttack_[consts::squares][256];

BitBoard minus9Direction_[consts::squares];
BitBoard plus9Direction_[consts::squares];
BitBoard minus7Direction_[consts::squares];
BitBoard plus7Direction_[consts::squares];
BitBoard minus1Direction_[consts::squares];
BitBoard plus1Direction_[consts::squares];
BitBoard minus8Direction_[consts::squares];
BitBoard plus8Direction_[consts::squares];

int directions_[consts::squares][consts::squares];

Square rotate45Left_[consts::squares];
Square rotate45Right_[consts::squares];
Square rotate90Left_[consts::squares];
```

```
BitBoard fileMask_[consts::files];
BitBoard rankMask_[consts::ranks];

int rankShift(Square s)
{
  return rankShift_[s];
}

int fileShift(Square s)
{
  return fileShift_[s];
}

int diagonalShift_a1h8(Square s)
{
  return diagonalShift_a1h8_[s];
}

int diagonalShift_h1a8(Square s)
{
  return diagonalShift_h1a8_[s];
}

Square unrotate45Left(Square s)
{
  return unrotate45Left_[s];
}

Square unrotate45Right(Square s)
{
  return unrotate45Right_[s];
}

Square unrotate90Left(Square s)
{
  return unrotate90Left_[s];
}

ostream& operator<<(ostream& os, const Move m)
{
  if (m == nullMove)
  {
    os << "[null]";
    return os;
  }

  if (m.piece() == king && fileDist(m.from(),m.to()) == 2)
  {
    if (getFile(m.to()) == fileG)
```

```
      os << "O-O";
    else
      os << "O-O-O";
    return os;
  }
  if (m.piece() != pawn)
    os << m.piece();
  else if (m.capture())
    os << getFile(m.from());
  if (m.capture() != none)
    os << "x";
  os << m.to();
  if (m.promote())
    os << "=" << m.promote();
  return os;
}

void initMoves()
{
  // Generate the rotation maps for squares
  forall(Square,s)
  {
    rotate45Left_[unrotate45Left_[s]] = s;
    rotate45Right_[unrotate45Right_[s]] = s;
    rotate90Left_[unrotate90Left_[s]] = s;
  }

  // Pawn Attacks
  forall (Square,s)
  {
    pawnAttack_[s][white] = empty;
    pawnAttack_[s][black] = empty;

    if (getFile(s) > fileA)
    {
      if (getRank(s) != rank8)
        pawnAttack_[s][white] |= mask(Square(s + 7));
      if (getRank(s) != rank1)
        pawnAttack_[s][black] |= mask(Square(s - 9));
    }

    if (getFile(s) < fileH)
    {
      if (getRank(s) != rank8)
        pawnAttack_[s][white] |= mask(Square(s + 9));
      if (getRank(s) != rank1)
        pawnAttack_[s][black] |= mask(Square(s - 7));
    }
  }
```

```
// Knight Attacks
const int knightOffset[] = {-17,-15,-10,-6,6,10,15,17};
forall(Square,from)
{
  knightAttack_[from] = empty;
  for (int i = 0; i != 8; ++i)
  {
    Square to = Square(from + knightOffset[i]);
    if (a1 <= to && to <= h8 && dist(from,to) == 2)
      knightAttack_[from] |= mask(to);
  }
}

// King Attacks
const int kingOffset[] = {-9,-8,-7,-1,1,7,8,9};
forall(Square,from)
{
  kingAttack_[from] = empty;
  for (int i = 0; i != 8; ++i)
  {
    Square to = Square(from + kingOffset[i]);
    if (a1 <= to && to <= h8 && dist(from,to) == 1)
      kingAttack_[from] |= mask(to);
  }
}

forall (Square,from)
  forall (Square,to)
    directions_[from][to] = 0;

// Diagonal Attacks (a1->h8 direction)
forall(Square,from)
{
  minus9Direction_[from] = empty;
  plus9Direction_[from] = empty;
  for (int state = 0; state <= 255; ++state)
  {
    BitBoard tmp = empty;
    BitBoard shiftState = BitBoard(state) << diagonalShift_a1h8(from);
    for (Square to = Square(from - 9); getFile(to) != fileH && to >= a1;
         to = Square(to - 9))
    {
      tmp |= mask(to);
      if (mask(rotate45Right(to)) & shiftState)
        break;
      if (state == 0)
      {
        minus9Direction_[from] |= mask(to);
```

```
        directions_[from][to] = -9;
      }
    }
    for (Square to = Square(from + 9); getFile(to) != fileA && to <= h8;
         to = Square(to + 9))
    {
      tmp |= mask(to);
      if (mask(rotate45Right(to)) & shiftState)
        break;
      if (state == 0)
      {
        plus9Direction_[from] |= mask(to);
        directions_[from][to] = 9;
      }
    }
    diagonalAttack_a1h8_[from][int(state)] = tmp;
  }
}

// Diagonal Attacks (h1->a8 direction)
forall(Square,from)
{
  minus7Direction_[from] = empty;
  plus7Direction_[from] = empty;
  for (int state = 0; state <= 255; ++state)
  {
    BitBoard tmp = empty;
    BitBoard shiftState = BitBoard(state) << diagonalShift_h1a8(from);
    for (Square to = Square(from - 7); getFile(to) != fileA && to >= a1;
         to = Square(to - 7))
    {
      tmp |= mask(to);
      if (mask(rotate45Left(to)) & shiftState)
        break;
      if (state == 0)
      {
        minus7Direction_[from] |= mask(to);
        directions_[from][to] = -7;
      }
    }
    for (Square to = Square(from + 7); getFile(to) != fileH && to <= h8;
         to = Square(to + 7))
    {
      tmp |= mask(to);
      if (mask(rotate45Left(to)) & shiftState)
        break;
      if (state == 0)
      {
        plus7Direction_[from] |= mask(to);
```

```
        directions_[from][to] = 7;
      }
    }
    diagonalAttack_h1a8_[from][int(state)] = tmp;
  }
}

// Generate the attack masks for attacks on a rank
forall(Square,from)
{
  minus1Direction_[from] = empty;
  plus1Direction_[from] = empty;
  for (int state = 0; state <= 255; ++state)
  {
    BitBoard tmp = empty;
    BitBoard shiftState = BitBoard(state) << rankShift(from);
    for (Square to = Square(from - 1); getFile(to) != fileH; --to)
    {
      tmp |= mask(to);
      if (mask(to) & shiftState)
        break;
        if (state == 0)
      {
        minus1Direction_[from] |= mask(to);
        directions_[from][to] = -1;
      }
    }
    for (Square to = Square(from + 1); getFile(to) != fileA; ++to)
    {
      tmp |= mask(to);
      if (mask(to) & shiftState)
        break;
      if (state == 0)
      {
        plus1Direction_[from] |= mask(to);
        directions_[from][to] = 1;
      }
    }
    rankAttack_[from][int(state)] = tmp;
  }
}

// Generate the attack masks for attacks on a file
forall(Square,from)
{
  minus8Direction_[from] = empty;
  plus8Direction_[from] = empty;
  for (int state = 0; state <= 255; ++state)
  {
```

```
      BitBoard tmp = empty;
      BitBoard shiftState = BitBoard(state) << fileShift(from);
      for (Square to = Square(from - 8); to >= a1;
           to = Square(to - 8))
      {
        tmp |= mask(to);
        if (mask(rotate90Left(to)) & shiftState)
          break;
        if (state == 0)
        {
          minus8Direction_[from] |= mask(to);
          directions_[from][to] = -8;
        }
      }
      for (Square to = Square(from + 8); to <= h8; to = Square(to + 8))
      {
        tmp |= mask(to);
        if (mask(rotate90Left(to)) & shiftState)
          break;
        if (state == 0)
        {
          plus8Direction_[from] |= mask(to);
          directions_[from][to] = 8;
        }
      }
      fileAttack_[from][int(state)] = tmp;
    }
  }

  // Masks for files and ranks
  forall (File,f)
    fileMask_[f] = empty;

  forall (Rank,r)
    rankMask_[r] = empty;

  forall (Square,s)
  {
    rankMask_[getRank(s)] |= mask(s);
    fileMask_[getFile(s)] |= mask(s);
  }
}


Square rotate45Left(Square s)
{
  return rotate45Left_[s];
}
```

```
Square rotate45Right(Square s)
{
  return rotate45Right_[s];
}

Square rotate90Left(Square s)
{
  return rotate90Left_[s];
}


BitBoard rotate45Left(const BitBoard& bitboard)
{
  BitBoard tmp = empty;
  forall(Square,s)
  {
    if (bitboard & mask(s))
      tmp |= mask(rotate45Left(s));
  }
  return tmp;
}

BitBoard unrotate45Left(const BitBoard& bitboard)
{
  BitBoard tmp = empty;
  forall(Square,s)
  {
    if (bitboard & mask(s))
      tmp |= mask(unrotate45Left(s));
  }
  return tmp;
}

BitBoard rotate45Right(const BitBoard& bitboard)
{
  BitBoard tmp = empty;
  forall(Square,s)
  {
    if (bitboard & mask(s))
      tmp |= mask(rotate45Right(s));
  }
  return tmp;
}

BitBoard unrotate45Right(const BitBoard& bitboard)
{
  BitBoard tmp = empty;
  forall(Square,s)
  {
```

```
    if (bitboard & mask(s))
      tmp |= mask(unrotate45Right(s));
  }
  return tmp;
}

BitBoard rotate90Left(const BitBoard& bitboard)
{
  BitBoard tmp = empty;
  forall(Square,s)
  {
    if (bitboard & mask(s))
      tmp |= mask(rotate90Left(s));
  }
  return tmp;
}

BitBoard unrotate90Left(const BitBoard& bitboard)
{
  BitBoard tmp = empty;
  forall(Square,s)
  {
    if (bitboard & mask(s))
      tmp |= mask(unrotate90Left(s));
  }
  return tmp;
}

/*
inline BitBoard pawnAttack(Square s, Color c)
{
  BitBoard result = empty;

  if (getFile(s) != fileA)
    result |= mask(Square(c == white ? s + 7 : s - 9));

  if (getFile(s) != fileH)
    result |= mask(Square(c == white ? s + 9 : s - 7));

  return result;
}
*/

inline BitBoard pawnAttack(Square s, Color c)
{
  return pawnAttack_[s][c];
}

inline BitBoard knightAttack(Square s)
```

```
{
  return knightAttack_[s];
}

inline BitBoard kingAttack(Square s)
{
  return kingAttack_[s];
}

inline BitBoard diagonalAttack_a1h8(Square s, const Position& pos)
{
  return diagonalAttack_a1h8_
    [s][(pos.occupiedRotate45Right() >> diagonalShift_a1h8(s)) & 0xff];
}

inline BitBoard diagonalAttack_h1a8(Square s, const Position& pos)
{
  return diagonalAttack_h1a8_
    [s][(pos.occupiedRotate45Left() >> diagonalShift_h1a8(s)) & 0xff];
}

inline BitBoard rankAttack(Square s, const Position& pos)
{
  return rankAttack_
    [s][(pos.occupied() >> rankShift(s)) & 0xff];
}

inline BitBoard fileAttack(Square s, const Position& pos)
{
  return fileAttack_
    [s][(pos.occupiedRotate90Left() >> fileShift(s)) & 0xff];
}

inline BitBoard bishopAttack(Square s, const Position& pos)
{
  return diagonalAttack_a1h8(s,pos) | diagonalAttack_h1a8(s,pos);
}

inline BitBoard rookAttack(Square s, const Position& pos)
{
  return rankAttack(s,pos) | fileAttack(s,pos);
}

inline BitBoard queenAttack(Square s, const Position& pos)
{
  return rookAttack(s,pos) | bishopAttack(s,pos);
}

BitBoard fileMask(File f)
```

```
{
  return fileMask_[f];
}

BitBoard rankMask(Rank r)
{
  return rankMask_[r];
}

BitBoard attacksTo(Square s, const Position& pos)
{
  return(
    (pawnAttack(s,white) & pos.bitboard(black,pawn)) |
    (pawnAttack(s,black) & pos.bitboard(white,pawn)) |
    (knightAttack(s) & (pos.bitboard(white,knight)
                         | pos.bitboard(black,knight))) |
    (bishopAttack(s,pos) &
     (pos.bitboard(white,bishop)
      | pos.bitboard(white,queen)
      | pos.bitboard(black,bishop)
      | pos.bitboard(black,queen))) |
    (rookAttack(s,pos) &
     (pos.bitboard(white,rook)
      | pos.bitboard(white,queen)
      | pos.bitboard(black,rook)
      | pos.bitboard(black,queen))) |
    (kingAttack(s) &
     (pos.bitboard(white,king) | pos.bitboard(black,king)))
    );
}

BitBoard xrayAttacks(const Position &pos, BitBoard attacks,
                     Square from, int direction)
{
  BitBoard pieces;
  switch (direction)
  {
    case 1:
      pieces = pos.bitboard(white,rook) | pos.bitboard(white,queen)
        | pos.bitboard(black,rook) | pos.bitboard(black,queen);
      return (attacks |
       (rankAttack(from,pos) & pieces & plus1Direction_[from]));
    case 7:
      pieces = pos.bitboard(white,bishop) | pos.bitboard(white,queen)
        | pos.bitboard(black,bishop) | pos.bitboard(black,queen);
      return(attacks |
       (diagonalAttack_h1a8(from,pos) & pieces & plus7Direction_[from]));
    case 8:
      pieces = pos.bitboard(white,rook) | pos.bitboard(white,queen)
```

```
        | pos.bitboard(black,rook) | pos.bitboard(black,queen);
      return(attacks |
       (fileAttack(from,pos) & pieces & plus8Direction_[from]));
    case 9:
      pieces = pos.bitboard(white,bishop) | pos.bitboard(white,queen)
        | pos.bitboard(black,bishop) | pos.bitboard(black,queen);
      return(attacks |
       (diagonalAttack_a1h8(from,pos) & pieces & plus9Direction_[from]));
    case -1:
      pieces = pos.bitboard(white,rook) | pos.bitboard(white,queen)
        | pos.bitboard(black,rook) | pos.bitboard(black,queen);
      return(attacks |
       (rankAttack(from,pos) & pieces & minus1Direction_[from]));
    case -7:
      pieces = pos.bitboard(white,bishop) | pos.bitboard(white,queen)
        | pos.bitboard(black,bishop) | pos.bitboard(black,queen);
      return(attacks |
       (diagonalAttack_h1a8(from,pos) & pieces & minus7Direction_[from]));
    case -8:
      pieces = pos.bitboard(white,rook) | pos.bitboard(white,queen)
        | pos.bitboard(black,rook) | pos.bitboard(black,queen);
      return(attacks |
       (fileAttack(from,pos) & pieces & minus8Direction_[from]));
    case -9:
      pieces = pos.bitboard(white,bishop) | pos.bitboard(white,queen)
        | pos.bitboard(black,bishop) | pos.bitboard(black,queen);
      return(attacks |
       (diagonalAttack_a1h8(from,pos) & pieces & minus9Direction_[from]));
  }
  return(attacks);
}

Score SEE(const Position &pos, Move move)
{
  // This function was basically "stolen" from Crafty

  Square source = move.from();
  Square target = move.to();

  const Score pieceValueSEE[Pieceend] =
    {
      Score(0),Score(1000),Score(3000),Score(3000),
      Score(5000),Score(9000),Score(150000)
    };

  int nc = 1;
  Score swap_list[32]; // No more than 32 pieces on the board

  BitBoard attacks = attacksTo(target,pos);
```

```
Score attacked_piece = pieceValueSEE[pos[target].piece()];

swap_list[0] = attacked_piece;
attacked_piece = pieceValueSEE[pos[source].piece()];
clear(attacks,source);
int direction = directions_[target][source];
if (direction)
  attacks = xrayAttacks(pos,attacks,source,direction);
Color color = opposite(pos.turn());

Square square;
while (attacks)
{
  if (color == white)
  {
    if (pos.bitboard(white,pawn) & attacks)
      square = firstBit(pos.bitboard(white,pawn) & attacks);
    else if (pos.bitboard(white,knight) & attacks)
      square = firstBit(pos.bitboard(white,knight) & attacks);
    else if (pos.bitboard(white,bishop) & attacks)
      square = firstBit(pos.bitboard(white,bishop) & attacks);
    else if (pos.bitboard(white,rook) & attacks)
      square = firstBit(pos.bitboard(white,rook) & attacks);
    else if (pos.bitboard(white,queen) & attacks)
      square = firstBit(pos.bitboard(white,queen) & attacks);
    else if (pos.bitboard(white,king) & attacks)
      square = firstBit(pos.bitboard(white,king) & attacks);
    else break;
  }
  else
  {
    if (pos.bitboard(black,pawn) & attacks)
      square = firstBit(pos.bitboard(black,pawn) & attacks);
    else if (pos.bitboard(black,knight) & attacks)
      square = firstBit(pos.bitboard(black,knight) & attacks);
    else if (pos.bitboard(black,bishop) & attacks)
      square = firstBit(pos.bitboard(black,bishop) & attacks);
    else if (pos.bitboard(black,rook) & attacks)
      square = firstBit(pos.bitboard(black,rook) & attacks);
    else if (pos.bitboard(black,queen) & attacks)
      square = firstBit(pos.bitboard(black,queen) & attacks);
    else if (pos.bitboard(black,king) & attacks)
      square = firstBit(pos.bitboard(black,king) & attacks);
    else break;
  }

  swap_list[nc] = Score(-swap_list[nc - 1] + attacked_piece);
  attacked_piece = pieceValueSEE[pos[square].piece()];
  clear(attacks,square);
```

```
    direction = directions_[target][square];
    if (direction)
      attacks = xrayAttacks(pos,attacks,square,direction);
    ++nc;
    color = opposite(color);
  }

  while(--nc)
    if(swap_list[nc] > -swap_list[nc - 1])
      swap_list[nc - 1] = Score(-swap_list[nc]);

  return (swap_list[0]);
}

bool isAttacking(Color color, Square s, const Position& pos)
{
  if (pawnAttack(s,opposite(color)) & pos.bitboard(color,pawn))
    return true;

  if (knightAttack(s) & pos.bitboard(color,knight))
    return true;

  if (bishopAttack(s,pos) &
      (pos.bitboard(color,bishop) | pos.bitboard(color,queen)))
    return true;

  if (rookAttack(s,pos) &
      (pos.bitboard(color,rook) | pos.bitboard(color,queen)))
    return true;

  if (kingAttack(s) & pos.bitboard(color,king))
    return true;

  return false;
}

BitBoard attack(Piece p, Square from, const Position& pos)
{
  switch (p)
  {
    case knight: return knightAttack(from);
    case bishop: return bishopAttack(from,pos);
    case rook: return rookAttack(from,pos);
    case queen: return queenAttack(from,pos);
    case king: return kingAttack(from);
    default: return empty;
  }
}
```

```
void generateMoves(Piece piece, const Position& pos,
                   Moves& moves, bool captures)
{
  BitBoard pieces = pos.bitboard(pos.turn(),piece);
  const BitBoard targets =
    captures ? pos.occupied(opposite(pos.turn())) : pos.nonOccupied();
  while (pieces)
  {
    Square from = firstBit(pieces);
    BitBoard attacks = attack(piece,from,pos) & targets;
    while (attacks)
    {
      Square to = firstBit(attacks);
      moves.push_back(Move(from,to,piece,pos[to].piece(),none));
      clear(attacks,to);
    }
    clear(pieces,from);
  }
}

void generateNonCapturePromotions(const Position& pos,Moves& moves)
{
  Color turn = pos.turn();

  const BitBoard targets = pos.nonOccupied();
  BitBoard attacks = turn == white ?
    (pos.bitboard(white,pawn) << 8) & targets :
    (pos.bitboard(black,pawn) >> 8) & targets;

  attacks &= rankMask(turn == white ? rank8 : rank1);

  while (attacks)
  {
    Square to = firstBit(attacks);
    Square from = Square(turn == white ? to - 8 : to + 8);
    for (Piece p = queen; p >= knight; --p)
      moves.push_back(Move(from,to,pawn,none,p));
    clear(attacks,to);
  }
}

void generatePawnMoves(const Position& pos, Moves& moves, bool captures)
{
  Color turn = pos.turn();
  if (captures)
  {
    BitBoard targets = pos.occupied(opposite(turn));

    if (pos.epSquare() != noEpSquare)
```

```
    targets |= mask(pos.epSquare());

  BitBoard attacks = turn == white ?
    ((pos.bitboard(white,pawn) & (~fileMask(fileA))) << 7) & targets :
    ((pos.bitboard(black,pawn) & (~fileMask(fileA))) >> 9) & targets;

  while (attacks)
  {
    Square to = firstBit(attacks);
    Square from = Square(turn == white ? to - 7 : to + 9);
    if (getRank(to) == (turn == white ? rank8 : rank1))
    {
      for (Piece p = queen; p >= knight; --p)
        moves.push_back(Move(from,to,pawn,pos[to].piece(),p));
    }
    else
    {
      if (to == pos.epSquare())
        moves.push_back(Move(from,to,pawn,pawn,none));
      else
        moves.push_back(Move(from,to,pawn,pos[to].piece(),none));
    }
    clear(attacks,to);
  }

  attacks = turn == white ?
    ((pos.bitboard(white,pawn) & (~fileMask(fileH))) << 9) & targets :
    ((pos.bitboard(black,pawn) & (~fileMask(fileH))) >> 7) & targets;

  while (attacks)
  {
    Square to = firstBit(attacks);
    Square from = Square(turn == white ? to - 9 : to + 7);
    if (getRank(to) == (turn == white ? rank8 : rank1))
    {
      for (Piece p = queen; p >= knight; --p)
        moves.push_back(Move(from,to,pawn,pos[to].piece(),p));
    }
    else
    {
      if (to == pos.epSquare())
        moves.push_back(Move(from,to,pawn,pawn,none));
      else
        moves.push_back(Move(from,to,pawn,pos[to].piece(),none));
    }
    clear(attacks,to);
  }
}
```

```
  else // Non-captures

  {
    const BitBoard targets = pos.nonOccupied();
    BitBoard attacks = turn == white ?
      (pos.bitboard(white,pawn) << 8) & targets :
      (pos.bitboard(black,pawn) >> 8) & targets;

    BitBoard attacks2 = turn == white ?
      ((attacks & rankMask(rank3)) << 8) & targets :
      ((attacks & rankMask(rank6)) >> 8) & targets;

    while (attacks)
    {
      Square to = firstBit(attacks);
      Square from = Square(turn == white ? to - 8 : to + 8);
      if (getRank(to) == (turn == white ? rank8 : rank1))
      {
        for (Piece p = queen; p >= knight; --p)
          moves.push_back(Move(from,to,pawn,none,p));
      }
      else
        moves.push_back(Move(from,to,pawn,none,none));
      clear(attacks,to);
    }

    while (attacks2)
    {
      Square to = firstBit(attacks2);
      Square from = Square(turn == white ? to - 16: to + 16);
      moves.push_back(Move(from,to,pawn,none,none));
      clear(attacks2,to);
    }
  }
}

void generateKnightMoves(const Position& pos, Moves& moves, bool captures)
{
  BitBoard pieces = pos.bitboard(pos.turn(),knight);
  const BitBoard targets =
    captures ? pos.occupied(opposite(pos.turn())) : pos.nonOccupied();
  while (pieces)
  {
    Square from = firstBit(pieces);
    BitBoard attacks = knightAttack(from) & targets;
    while (attacks)
    {
      Square to = firstBit(attacks);
      moves.push_back(Move(from,to,knight,pos[to].piece(),none));
```

```
      clear(attacks,to);
    }
    clear(pieces,from);
  }
}

void generateBishopMoves(const Position& pos, Moves& moves, bool captures)
{
  BitBoard pieces = pos.bitboard(pos.turn(),bishop);
  const BitBoard targets =
    captures ? pos.occupied(opposite(pos.turn())) : pos.nonOccupied();
  while (pieces)
  {
    Square from = firstBit(pieces);
    BitBoard attacks = bishopAttack(from,pos) & targets;
    while (attacks)
    {
      Square to = firstBit(attacks);
      moves.push_back(Move(from,to,bishop,pos[to].piece(),none));
      clear(attacks,to);
    }
    clear(pieces,from);
  }
}

void generateRookMoves(const Position& pos, Moves& moves, bool captures)
{
  BitBoard pieces = pos.bitboard(pos.turn(),rook);
  const BitBoard targets =
    captures ? pos.occupied(opposite(pos.turn())) : pos.nonOccupied();
  while (pieces)
  {
    Square from = firstBit(pieces);
    BitBoard attacks = rookAttack(from,pos) & targets;
    while (attacks)
    {
      Square to = firstBit(attacks);
      moves.push_back(Move(from,to,rook,pos[to].piece(),none));
      clear(attacks,to);
    }
    clear(pieces,from);
  }
}

void generateQueenMoves(const Position& pos, Moves& moves, bool captures)
{
  BitBoard pieces = pos.bitboard(pos.turn(),queen);
  const BitBoard targets =
    captures ? pos.occupied(opposite(pos.turn())) : pos.nonOccupied();
```

```
  while (pieces)
  {
    Square from = firstBit(pieces);
    BitBoard attacks = queenAttack(from,pos) & targets;
    while (attacks)
    {
      Square to = firstBit(attacks);
      moves.push_back(Move(from,to,queen,pos[to].piece(),none));
      clear(attacks,to);
    }
    clear(pieces,from);
  }
}

void generateKingMoves(const Position& pos, Moves& moves, bool captures)
{
  BitBoard pieces = pos.bitboard(pos.turn(),king);
  const BitBoard targets =
    captures ? pos.occupied(opposite(pos.turn())) : pos.nonOccupied();
  while (pieces)
  {
    Square from = firstBit(pieces);
    BitBoard attacks = kingAttack(from) & targets;
    while (attacks)
    {
      Square to = firstBit(attacks);
      moves.push_back(Move(from,to,king,pos[to].piece(),none));
      clear(attacks,to);
    }
    clear(pieces,from);
  }

  if (pos.turn() == white)
  {
    if (!captures && (pos.castlingRight(white) & kingside))
    {
      if (!(pos.occupied() & (mask(f1) | mask(g1))))
        if (!isAttacking(black,e1,pos) &&
            !isAttacking(black,f1,pos) &&
            !isAttacking(black,g1,pos))
          moves.push_back(Move(e1,g1,king,none,none));
    }

    if (!captures && (pos.castlingRight(white) & queenside))
    {
      if (!(pos.occupied() & (mask(c1) | mask(d1) | mask(b1))))
        if (!isAttacking(black,e1,pos) &&
            !isAttacking(black,d1,pos) &&
            !isAttacking(black,c1,pos))
```

```
        moves.push_back(Move(e1,c1,king,none,none));
    }
  }
  else
  {
    if (!captures && (pos.castlingRight(black) & kingside))
    {
      if (!(pos.occupied() & (mask(f8) | mask(g8))))
        if (!isAttacking(white,e8,pos) &&
            !isAttacking(white,f8,pos) &&
            !isAttacking(white,g8,pos))
          moves.push_back(Move(e8,g8,king,none,none));
    }

    if (!captures && (pos.castlingRight(black) & queenside))
    {
      if (!(pos.occupied() & (mask(c8) | mask(d8) | mask(b8))))
        if (!isAttacking(white,e8,pos) &&
            !isAttacking(white,d8,pos) &&
            !isAttacking(white,c8,pos))
          moves.push_back(Move(e8,c8,king,none,none));
    }
  }

}

void generateMoves(const Position& pos, Moves& moves)
{
//  forall (Piece,p)
//    generateMoves(p,pos,moves,true);

//  forall (Piece,p)
//    generateMoves(p,pos,moves,false);

  generatePawnMoves(pos,moves,true);
  generateKnightMoves(pos,moves,true);
  generateBishopMoves(pos,moves,true);
  generateRookMoves(pos,moves,true);
  generateQueenMoves(pos,moves,true);
  generateKingMoves(pos,moves,true);

  generatePawnMoves(pos,moves,false);
  generateKnightMoves(pos,moves,false);
  generateBishopMoves(pos,moves,false);
  generateRookMoves(pos,moves,false);
  generateQueenMoves(pos,moves,false);
  generateKingMoves(pos,moves,false);

}
```

```
void generateNonQuietMoves(const Position& pos, Moves& moves)
{
  generateNonCapturePromotions(pos,moves);
  generatePawnMoves(pos,moves,true);
  generateKnightMoves(pos,moves,true);
  generateBishopMoves(pos,moves,true);
  generateRookMoves(pos,moves,true);
  generateQueenMoves(pos,moves,true);
  generateKingMoves(pos,moves,true);
}
```

B.6 parallel.cpp

```
// $Id: parallel.cpp,v 1.23 2004/07/26 14:26:10 s958547 Exp $

#include <iostream>
#include <sstream>

#include "parallel.h"
#include "position.h"
#include "search.h"

using namespace std;

int nThreads = 1;
int maxThreadsPerNode = 1;

volatile int idleThreads;

pthread_mutex_t globalMutex;
pthread_mutex_t outputMutex;

Node* volatile thread[maxThreads];
Node* splitBlock[nSplitBlocks];

Node* rNode;

void childToParent(Node& parent, Node& child)
{
  if (child.nodes && !child.stopped && child.cValue > parent.pValue)
  {
    if (child.cValue < parent.beta)
      parent.pv.copy(child.pv,parent.ply);

    parent.pValue = child.cValue;
    parent.bestMove = child.bestMove;
  }
```

```
  parent.nodes += child.nodes;
  parent.qNodes += child.qNodes;

  parent.failHighs += child.failHighs;
  parent.failHighFirsts += child.failHighFirsts;

  parent.transRefProbes += child.transRefProbes;
  parent.transRefHits += child.transRefHits;

  child.used = 0;
}

Node* parentToChild(Node& parent, int childThread)
{

  int from = childThread*splitBlocksPerThread;
  int to = (childThread+1)*splitBlocksPerThread;

  if (from == 0)
    from = 1;

  int free;
  for (free = from; free < to; ++free)
    if (!splitBlock[free]->used)
      break;

  if (free >= to)
  {
    for (free = 1; free < nSplitBlocks; ++free)
      if (!splitBlock[free]->used)
break;
  }

  if (free >= nSplitBlocks)
  {
    cout << "Out of split blocks..." << endl;
    return 0;
  }

  Node& child = *splitBlock[free];
  child.used = true;
  child.stopped = false;

  for (int t = 0; t < nThreads; ++t)
    child.children[t] = 0;

  // copy stuff
  child.pos = parent.pos;
```

```
  child.nodes = 0;
  child.qNodes = 0;
  child.failHighs = 0;
  child.failHighFirsts = 0;
  child.transRefProbes = 0;
  child.transRefHits = 0;

  child.alpha = parent.alpha;
  child.beta = parent.beta;
  child.ply = parent.ply;
  child.depth = parent.depth;
  child.totalExtension = parent.totalExtension;

  child.cValue = Score(-illegal);

  return &child;
}

void threadStop(Node* node)
{
  pthread_mutex_lock(&node->mutex);
  node->stopped = true;
  for (int t = 0; t < nThreads; ++t)
    if (node->children[t])
      threadStop(node->children[t]);
  pthread_mutex_unlock(&node->mutex);
}

void searchParallel(Node& node, Score alpha, Score beta,
                    Depth depth, int ply, Depth totalExtension)
{
  node.bestMove = nullMove;

  Position& pos = node.pos;

  // The main search loop
  Moves& moves = node.parent->moves[ply];
  int& i = node.parent->nextMove[ply];

  while (true)
  {
    pthread_mutex_lock(&node.parent->mutex);

      if (i >= moves.size())
      {
        pthread_mutex_unlock(&node.parent->mutex);
        break;
      }
```

```
  moves.sortFrom(i);
  Move move = moves[i];
  ++i;

pthread_mutex_unlock(&node.parent->mutex);

pos.makeMove(move);

if (inCheck(opposite(pos.turn()),pos))
{
  pos.unMakeMove(move);
  continue;
}

Score value;
value = Score(-search(node, Score(-alpha - 1), Score(-alpha),
                      Depth(depth - onePly), ply + 1, true,
                      totalExtension));
if (node.stopped)
{
  pos.unMakeMove(move);
  break;
}

if (value > alpha && value < beta)
{
  value = (Score)(-search(node, Score(-beta), Score(-alpha),
                          Depth(depth - onePly), ply + 1, true,
                          totalExtension));
  if (node.stopped)
  {
    pos.unMakeMove(move);
    break;
  }
}

pos.unMakeMove(move);

if (value > alpha)
{
  node.bestMove = move;
  node.cValue = value;

  if (value >= beta)
  {
    pthread_mutex_lock(&globalMutex);
      pthread_mutex_lock(&node.parent->mutex);
      if (!node.stopped)
```

```
          {
            for (int t = 0; t < nThreads; ++t)
              if (node.parent->children[t] && t != node.threadNumber)
                threadStop(node.parent->children[t]);
          }
          pthread_mutex_unlock(&node.parent->mutex);
        pthread_mutex_unlock(&globalMutex);
        break;
      }
      alpha = value;
      node.pv.update(ply,node.bestMove);
    }
  }
}

int threadLoop(int threadNumber, Node* waiting)
{
  while (true)
  {
    pthread_mutex_lock(&globalMutex);
      ++idleThreads;
    pthread_mutex_unlock(&globalMutex);

    while (!thread[threadNumber] && (!waiting || waiting->workers))
      /* spin */;

    pthread_mutex_lock(&globalMutex);
      if (!thread[threadNumber])
        thread[threadNumber] = waiting;

      --idleThreads;
    pthread_mutex_unlock(&globalMutex);

    if (thread[threadNumber] == waiting)
      return 0;

    Node& node = *thread[threadNumber];
    searchParallel(node, node.alpha, node.beta,
                   node.depth, node.ply, node.totalExtension);

    pthread_mutex_lock(&globalMutex);
      pthread_mutex_lock(&thread[threadNumber]->parent->mutex);

        childToParent(*thread[threadNumber]->parent,
                      *thread[threadNumber]);
        thread[threadNumber]->parent->workers--;
        thread[threadNumber]->parent->children[threadNumber] = 0;

      pthread_mutex_unlock(&thread[threadNumber]->parent->mutex);
```

```
      thread[threadNumber] = 0;
    pthread_mutex_unlock(&globalMutex);
  }
}

void* threadInit(void* tp)
{
  int threadNumber = (int)tp;

  int from = threadNumber*splitBlocksPerThread;
  int to = (threadNumber + 1)*splitBlocksPerThread;
  for (int i = from; i < to; ++i)
  {
    splitBlock[i] = new Node;
    splitBlock[i]->used = false;
    pthread_mutex_init(&splitBlock[i]->mutex,0);
  }

  threadLoop(threadNumber,0);
  return 0;
}

void initParallel()
{
  idleThreads = 0;

  for (int i = 0; i < splitBlocksPerThread; ++i)
  {
    splitBlock[i] = new Node;
    splitBlock[i]->used = false;
    pthread_mutex_init(&splitBlock[i]->mutex,0);
  }

  splitBlock[0]->parent = 0;
  splitBlock[0]->used = true;
  splitBlock[0]->stopped = 0;
  splitBlock[0]->workers = 0;
  splitBlock[0]->threadNumber = 0;

  pthread_mutex_init(&globalMutex,0);
  pthread_mutex_init(&outputMutex,0);

  pthread_attr_t pthread_attr;
  pthread_attr_init(&pthread_attr);
  pthread_attr_setdetachstate(&pthread_attr, PTHREAD_CREATE_DETACHED);
  pthread_attr_setscope(&pthread_attr, PTHREAD_SCOPE_SYSTEM);

  cout << "Thread 0 created (main thread)" << endl;
```

```
  for (int t = 1; t < nThreads; ++t)
  {
    cout << "Creating worker thread " << t << endl;
    thread[t] = 0;
    pthread_t pthread;
    pthread_create(&pthread,&pthread_attr,threadInit,(void*) t);
  }

  while (idleThreads < nThreads-1);
  cout << "Threads started: " << nThreads << endl;

  thread[0] = splitBlock[0];
  rNode = splitBlock[0];
}

bool split(Node* node)
{
  pthread_mutex_lock(&globalMutex);
    int t;
    for (t = 0; t < nThreads; ++t)
      if (thread[t] == 0)
        break;
    if (t == nThreads || node->stopped)
    {
      pthread_mutex_unlock(&globalMutex);
      return false;
    }

    thread[node->threadNumber] = 0;
    node->workers = 0;

    int nblocks = 0;
    for (t = 0; t < nThreads && nblocks < maxThreadsPerNode; ++t)
    {
      node->children[t] = 0;
      if (thread[t] == 0)
      {
        Node* block = parentToChild(*node,t);
        if (!block) // No available blocks
          continue;
        node->children[t] = block;
        block->threadNumber = t;
        block->pos.threadNumber = t;
        block->parent = node;
        ++node->workers;
        ++nblocks;
      }
    }
```

```
    node->pValue = node->alpha;

    if (!nblocks)
    {
      thread[node->threadNumber] = node;
      pthread_mutex_unlock(&globalMutex);
      return false;
    }

    for (t = 0; t < nThreads; ++t)
      if (node->children[t])
        thread[t] = node->children[t];

  pthread_mutex_unlock(&globalMutex);

  threadLoop(node->threadNumber, node);

  return true;
}
```

B.7 position.cpp

```
// $Id: position.cpp,v 1.27 2004/07/26 14:26:10 s958547 Exp $

#include <iostream>
#include <sstream>
#include <cctype>
#include <ctime>
#include <sstream>
#include <string>

using namespace std;

#include "position.h"
#include "moves.h"
#include "evaluate.h"
#include "hash.h"

void checkIntegrity(const Position& pos, Move m, const string& desc)
{
  stringstream ss;

  BitBoard bitboard_[Colorend][Pieceend];
  BitBoard occupied_[Colorend];
  BitBoard occupiedRotate45Left_;
  BitBoard occupiedRotate45Right_;
  BitBoard occupiedRotate90Left_;
```

```
forall (Color,c)
{
  occupied_[c] = empty;
  forall (Piece,p)
    bitboard_[c][p] = empty;
}

occupiedRotate45Left_ = empty;
occupiedRotate45Right_ = empty;
occupiedRotate90Left_ = empty;

forall(Square,s)
{
  if (pos[s].piece() != none)
  {
    bitboard_[pos[s].color()][pos[s].piece()] |= mask(s);
    occupied_[pos[s].color()] |= mask(s);
    occupiedRotate45Left_ |= mask(rotate45Left(s));
    occupiedRotate45Right_ |= mask(rotate45Right(s));
    occupiedRotate90Left_ |= mask(rotate90Left(s));
  }
}

forall (Color,c)
  forall (Piece,p)
  if (pos.bitboard(c,p) != bitboard_[c][p])
    ss << "bitboard(c,p) mismatch: " << c << "," << p << endl;

forall (Color,c)
  if (pos.occupied(c) != occupied_[c])
  {
    ss << "occupied(c) mismatch: " << c << endl;
    ss << pos.occupied(c) << endl;
    ss << occupied_[c] << endl;
  }

if (pos.occupiedRotate45Left() != occupiedRotate45Left_)
    ss << "occupied45Left(c) mismatch: " << endl;

if (pos.occupiedRotate45Right() != occupiedRotate45Right_)
    ss << "occupied45Right(c) mismatch: " << endl;

if (pos.occupiedRotate90Left() != occupiedRotate90Left_)
    ss << "occupied90Left(c) mismatch: " << endl;

int mat[Colorend];
forall (Color,c)
  mat[c] = 0;
```

```
  forall (Square,s)
    mat[pos[s].color()] += pieceValue(pos[s].piece());

  forall (Color,c)
  {
    if (pos.material(c) != Score(mat[c]))
    {
      ss << "material[" << c << "] mismatch" << endl;
      ss << "material: " << pos.material(c) << endl;
      ss << "calc    : " << Score(mat[c]) << endl;
    }
  }

  HashKey hashKey = calcHashKey(pos);
  if (hashKey != pos.hashKey())
  {
    ss << "hashKey mismatch" << endl;
    ss << "hashKey: " << pos.hashKey() << endl;
    ss << "calc   : " << hashKey << endl;
  }

  string s = ss.str();
  if (!s.empty())
  {
    cout << desc << endl << "threadID: " << pos.threadNumber << endl;
    cout << pos << endl;
    cout << "Move: " << m << endl;
    cout << s << endl;
    exit(EXIT_FAILURE);
  }

}

void Position::makeNullMove()
{
  gameRecord_[gamePly_].epSquare = epSquare_;
  gameRecord_[gamePly_].halfmoveClock = halfmoveClock_;
  gameRecord_[gamePly_].move = nullMove;
  gameRecord_[gamePly_].hashKey = hashKey_;
  ++gamePly_;
  ++halfmoveClock_;

  hashKey_ ^= hashEpSquare(epSquare_);
  epSquare_ = noEpSquare;
  hashKey_ ^= hashEpSquare(epSquare_);

  turn_ = opposite(turn_);

#if !defined NDEBUG
```

```
  checkIntegrity(*this,nullMove,"makeNullMove");
#endif
}

void Position::unMakeNullMove()
{
  turn_ = opposite(turn_);

  --halfmoveClock_;
  --gamePly_;

#if !defined NDEBUG
  if (gamePly_ < 0)
  {
    cout << endl << "gamePly_ < 0!" << endl;
  }
#endif

  epSquare_ = gameRecord_[gamePly_].epSquare;
  halfmoveClock_ = gameRecord_[gamePly_].halfmoveClock;
  hashKey_ = gameRecord_[gamePly_].hashKey;

#if !defined NDEBUG
  if (gameRecord_[gamePly_].move != nullMove)
    cout << "makeMove/unMakeMove mismatch!" << endl;
#endif

#if !defined NDEBUG
  checkIntegrity(*this,nullMove,"unMakeNullMove");
#endif
}

void Position::makeMove(Move m)
{
  const Square from = m.from();
  const Square to = m.to();
  const Piece piece = m.piece();
  const Piece capture = m.capture();
  const Piece promote = m.promote();

  const Color color = turn_;
  const Color oppColor = opposite(color);

  gameRecord_[gamePly_].epSquare = epSquare_;
  gameRecord_[gamePly_].castlingRight[white] = castlingRight_[white];
  gameRecord_[gamePly_].castlingRight[black] = castlingRight_[black];
  gameRecord_[gamePly_].halfmoveClock = halfmoveClock_;
  gameRecord_[gamePly_].move = m;
  gameRecord_[gamePly_].hashKey = hashKey_;
```

```
++gamePly_;

if (piece == pawn || capture != none)
  halfmoveClock_ = 0;
else
  ++halfmoveClock_;

// Remove the piece from source square
board_[from] = emptySquare;
clear(bitboard_[color][piece],from);
clear(occupied_[color],from);
clear(occupiedRotate45Left_,rotate45Left(from));
clear(occupiedRotate45Right_,rotate45Right(from));
clear(occupiedRotate90Left_,rotate90Left(from));
hashKey_ ^= hashPiece(color,piece,from);

switch (piece)
{
  case pawn:
    if (promote)
    { // Promotion
      // Put the piece on destination square
      board_[to] = ColoredPiece(color,promote);
      set(bitboard_[color][promote],to);
      set(occupied_[color],to);
      hashKey_ ^= hashPiece(color,promote,to);

      material_[color] += Score(pieceValue(promote) - pieceValue(pawn));

      if (!capture)
      {
        set(occupiedRotate45Left_,rotate45Left(to));
        set(occupiedRotate45Right_,rotate45Right(to));
        set(occupiedRotate90Left_,rotate90Left(to));
      }
      else
      { // Remove the captured piece from the destination square
        clear(bitboard_[oppColor][capture],to);
        clear(occupied_[oppColor],to);

        material_[oppColor] -= pieceValue(capture);

        hashKey_ ^= hashPiece(oppColor,capture,to);
      }
      break;
    }
    else if (to == epSquare_)
    { // En Passant
      // Put the piece on destination square
```

```
      board_[to] = ColoredPiece(color,piece);
      set(bitboard_[color][piece],to);
      set(occupied_[color],to);
      set(occupiedRotate45Left_,rotate45Left(to));
      set(occupiedRotate45Right_,rotate45Right(to));
      set(occupiedRotate90Left_,rotate90Left(to));
      hashKey_ ^= hashPiece(color,piece,to);

      // Remove the captured piece from it's square
      Square capSquare = Square(color == white ? to - 8 : to + 8);
      board_[capSquare] = emptySquare;
      clear(bitboard_[oppColor][capture],capSquare);
      clear(occupied_[oppColor],capSquare);
      clear(occupiedRotate45Left_,rotate45Left(capSquare));
      clear(occupiedRotate45Right_,rotate45Right(capSquare));
      clear(occupiedRotate90Left_,rotate90Left(capSquare));

      material_[oppColor] -= pieceValue(capture);

      hashKey_ ^= hashPiece(oppColor,capture,capSquare);

      break;
    }
    // Normal pawn move:
  case knight:
  case bishop:
  case rook:
  case queen:
  case king:
    // Put the piece on destination square
    board_[to] = ColoredPiece(color,piece);
    set(bitboard_[color][piece],to);
    set(occupied_[color],to);

    hashKey_ ^= hashPiece(color,piece,to);

    if (!capture)
    {
      set(occupiedRotate45Left_,rotate45Left(to));
      set(occupiedRotate45Right_,rotate45Right(to));
      set(occupiedRotate90Left_,rotate90Left(to));
    }
    else
    { // Remove the captured piece from the destination square
      clear(bitboard_[oppColor][capture],to);
      clear(occupied_[oppColor],to);

      material_[oppColor] -= pieceValue(capture);
```

```
      hashKey_ ^= hashPiece(oppColor,capture,to);
    }

    if (piece == king && fileDist(from,to) == 2)
    { // Castling
      Square fromR,toR;
      if (getFile(to) == fileG)
      {
        fromR = Square(to + 1);
        toR = Square(to - 1);
      }
      else
      {
        fromR = Square(to - 2);
        toR = Square(to + 1);
      }

      // Remove the rook from source square
      board_[fromR] = emptySquare;
      clear(bitboard_[color][rook],fromR);
      clear(occupied_[color],fromR);
      clear(occupiedRotate45Left_,rotate45Left(fromR));
      clear(occupiedRotate45Right_,rotate45Right(fromR));
      clear(occupiedRotate90Left_,rotate90Left(fromR));
      hashKey_ ^= hashPiece(color,rook,fromR);

      // Put the rook on the destination square
      board_[toR] = ColoredPiece(color,rook);
      set(bitboard_[color][rook],toR);
      set(occupied_[color],toR);
      set(occupiedRotate45Left_,rotate45Left(toR));
      set(occupiedRotate45Right_,rotate45Right(toR));
      set(occupiedRotate90Left_,rotate90Left(toR));
      hashKey_ ^= hashPiece(color,rook,toR);
    }
    break;
}

hashKey_ ^= hashEpSquare(epSquare_);
epSquare_ = noEpSquare;
if (piece == pawn && rankDist(from,to) == 2)
{
  // maybe only if opponents pawn present? probably, for hashing
  epSquare_ = Square(color == white ? from + 8 : from - 8);
}
hashKey_ ^= hashEpSquare(epSquare_);

hashKey_ ^= hashCastling(white,castlingRight_[white]);
if (castlingRight_[white])
```

```
  {
    if ((mask(from) | mask(to)) & (mask(a1) | mask(e1)))
      castlingRight_[white] =
        CastlingRight(castlingRight_[white] & kingside);

    if ((mask(from) | mask(to)) & (mask(h1) | mask(e1)))
      castlingRight_[white] =
        CastlingRight(castlingRight_[white] & queenside);
  }
  hashKey_ ^= hashCastling(white,castlingRight_[white]);

  hashKey_ ^= hashCastling(black,castlingRight_[black]);
  if (castlingRight_[black])
  {
    if ((mask(from) | mask(to)) & (mask(a8) | mask(e8)))
      castlingRight_[black] =
        CastlingRight(castlingRight_[black] & kingside);

    if ((mask(from) | mask(to)) & (mask(h8) | mask(e8)))
      castlingRight_[black] =
        CastlingRight(castlingRight_[black] & queenside);
  }
  hashKey_ ^= hashCastling(black,castlingRight_[black]);

  turn_ = oppColor;

#if !defined NDEBUG
  checkIntegrity(*this,m,"makeMove");
#endif
}

void Position::unMakeMove(Move m)
{
  const Square from = m.from();
  const Square to = m.to();
  const Piece piece = m.piece();
  const Piece capture = m.capture();
  const Piece promote = m.promote();

  const Color oppColor = turn_;
  const Color color = opposite(turn_);

  turn_ = color;

  --gamePly_;
#if !defined NDEBUG
  if (gamePly_ < 0)
  {
    cout << endl << "gamePly_ < 0!" << endl;
```

```
  }
#endif

  epSquare_ = gameRecord_[gamePly_].epSquare;
  castlingRight_[white] = gameRecord_[gamePly_].castlingRight[white];
  castlingRight_[black] = gameRecord_[gamePly_].castlingRight[black];
  halfmoveClock_ = gameRecord_[gamePly_].halfmoveClock;
  hashKey_ = gameRecord_[gamePly_].hashKey;

#if !defined NDEBUG
  if (m != gameRecord_[gamePly_].move)
    cout << "makeMove/unMakeMove mismatch!" << endl;
#endif

  // Put the piece back on source square
  board_[from] = ColoredPiece(color,piece);
  set(bitboard_[color][piece],from);
  set(occupied_[color],from);
  set(occupiedRotate45Left_,rotate45Left(from));
  set(occupiedRotate45Right_,rotate45Right(from));
  set(occupiedRotate90Left_,rotate90Left(from));

  switch (piece)
  {
    case pawn:
      if (promote)
      { // Promotion
        // Remove the piece from destination square
        clear(bitboard_[color][promote],to);
        clear(occupied_[color],to);

        material_[color] -= Score(pieceValue(promote) - pieceValue(pawn));

        if (!capture)
        {
          board_[to] = emptySquare;
          clear(occupiedRotate45Left_,rotate45Left(to));
          clear(occupiedRotate45Right_,rotate45Right(to));
          clear(occupiedRotate90Left_,rotate90Left(to));
        }
        else
        { // Put the captured piece back on the destination square
          board_[to] = ColoredPiece(oppColor,capture);
          set(bitboard_[oppColor][capture],to);
          set(occupied_[oppColor],to);

          material_[oppColor] += pieceValue(capture);

        }
```

```
    break;
  }
  else if (to == epSquare_)
  { // En Passant
    // Remove the piece from destination square
    board_[to] = emptySquare;
    clear(bitboard_[color][piece],to);
    clear(occupied_[color],to);
    clear(occupiedRotate45Left_,rotate45Left(to));
    clear(occupiedRotate45Right_,rotate45Right(to));
    clear(occupiedRotate90Left_,rotate90Left(to));

    // Put the captured piece back on it's square
    Square capSquare = Square(color == white ? to - 8 : to + 8);
    board_[capSquare] = ColoredPiece(oppColor,capture);
    set(bitboard_[oppColor][capture],capSquare);
    set(occupied_[oppColor],capSquare);
    set(occupiedRotate45Left_,rotate45Left(capSquare));
    set(occupiedRotate45Right_,rotate45Right(capSquare));
    set(occupiedRotate90Left_,rotate90Left(capSquare));

    material_[oppColor] += pieceValue(capture);

    break;
  }
  // Normal pawn move:
case knight:
case bishop:
case rook:
case queen:
case king:
  // Remove the piece from destination square
  clear(bitboard_[color][piece],to);
  clear(occupied_[color],to);
  if (!capture)
  {
    board_[to] = emptySquare;
    clear(occupiedRotate45Left_,rotate45Left(to));
    clear(occupiedRotate45Right_,rotate45Right(to));
    clear(occupiedRotate90Left_,rotate90Left(to));
  }
  else
  { // Put the captured piece back on the destination square
    board_[to] = ColoredPiece(oppColor,capture);
    set(bitboard_[oppColor][capture],to);
    set(occupied_[oppColor],to);

    material_[oppColor] += pieceValue(capture);
```

```
      }

      if (piece == king && fileDist(from,to) == 2)
      { // Castling
        Square fromR,toR;
        if (getFile(to) == fileG)
        {
          fromR = Square(to + 1);
          toR = Square(to - 1);
        }
        else
        {
          fromR = Square(to - 2);
          toR = Square(to + 1);
        }

        // Remove the rook from the destination square
        board_[toR] = emptySquare;
        clear(bitboard_[color][rook],toR);
        clear(occupied_[color],toR);
        clear(occupiedRotate45Left_,rotate45Left(toR));
        clear(occupiedRotate45Right_,rotate45Right(toR));
        clear(occupiedRotate90Left_,rotate90Left(toR));

        // Put the rook back on source square
        board_[fromR] = ColoredPiece(color,rook);
        set(bitboard_[color][rook],fromR);
        set(occupied_[color],fromR);
        set(occupiedRotate45Left_,rotate45Left(fromR));
        set(occupiedRotate45Right_,rotate45Right(fromR));
        set(occupiedRotate90Left_,rotate90Left(fromR));
      }
      break;
  }

#if !defined NDEBUG
  checkIntegrity(*this,m,"unMakeMove");
#endif
}

ostream& operator<<(ostream& os, const Position& pos)
{
  os << "   +---+---+---+---+---+---+---+---+" << newline;
  for (Rank r = rank8; r >= rank1; --r)
  {
    os << " " << int(r+1) << " |";
    for (File f = fileA; f <= fileH; ++f)
      os << pos[toSquare(f,r)] << " |";
    os << newline;
```

```
    os << "   +---+---+---+---+---+---+---+---+" << newline;
  }
  os << "     a   b   c   d   e   f   g   h" << newline;
  os << "\nFEN: " << pos.getFEN() << newline;
  return os;
}

string Position::getFEN() const
{
  stringstream FEN;
  for (Rank r = rank8; r >= rank1; --r)
  {
    int nones = 0;
    forall (File,f)
    {
      ColoredPiece cp = board_[toSquare(f,r)];
      if (cp.piece() == none)
        ++nones;
      else
      {
        if (nones)
          FEN << nones;
        nones = 0;
        if (cp.color() == white)
          FEN << letter(cp.piece());
        else
          FEN << char(tolower(letter(cp.piece())));
      }
    }
    if (nones)
      FEN << nones;

    if (r != rank1)
      FEN << "/";
  }

  FEN << " ";

  if (turn_ == white)
    FEN << "w ";
  else
    FEN << "b ";

  if (castlingRight_[white] & kingside) FEN << "K";
  if (castlingRight_[white] & queenside) FEN << "Q";
  if (castlingRight_[black] & kingside) FEN << "k";
  if (castlingRight_[black] & queenside) FEN << "q";
  if (!castlingRight_[white] && !castlingRight_[black]) FEN << "-";
```

```
  FEN << " ";

  if (epSquare_ == noEpSquare)
    FEN << "- ";
  else
    FEN << epSquare_ << " ";

  FEN << halfmoveClock_ << " ";

  FEN << "1";

  return FEN.str();
}

void Position::updateBitBoards()
{
  forall(Color,c)
  {
    occupied_[c] = empty;
    forall (Piece,p)
      bitboard_[c][p] = empty;
  }

  occupiedRotate45Left_ = empty;
  occupiedRotate45Right_ = empty;
  occupiedRotate90Left_ = empty;

  forall(Square,s)
  {
    if (board_[s].piece() != none)
    {
      bitboard_[board_[s].color()][board_[s].piece()] |= mask(s);
      occupied_[board_[s].color()] |= mask(s);
      occupiedRotate45Left_ |= mask(rotate45Left(s));
      occupiedRotate45Right_ |= mask(rotate45Right(s));
      occupiedRotate90Left_ |= mask(rotate90Left(s));
    }
  }
}

class ParseError
{
  public:
    ParseError(string ss) : s(ss) {}
    string error() { return s; }
  private:
    string s;
};
```

```
void parseRank(stringstream& ss, ColoredPiece rank[])
{
  char c;
  File f = fileA;
  while (f <= fileH)
  {
    ss >> c;
    if (c >= '1' && c <= '8')
    {
      int count = c - '0';
      if (f + count - 1 > fileH)
        throw ParseError("Too many squares on rank");
      for (int i = 0; i < count; ++i)
        rank[f++] = emptySquare;
      continue;
    }

    switch (c)
    {
      case 'P' : rank[f++] = ColoredPiece(white,pawn); break;
      case 'p' : rank[f++] = ColoredPiece(black,pawn); break;
      case 'N' : rank[f++] = ColoredPiece(white,knight); break;
      case 'n' : rank[f++] = ColoredPiece(black,knight); break;
      case 'B' : rank[f++] = ColoredPiece(white,bishop); break;
      case 'b' : rank[f++] = ColoredPiece(black,bishop); break;
      case 'R' : rank[f++] = ColoredPiece(white,rook); break;
      case 'r' : rank[f++] = ColoredPiece(black,rook); break;
      case 'Q' : rank[f++] = ColoredPiece(white,queen); break;
      case 'q' : rank[f++] = ColoredPiece(black,queen); break;
      case 'K' : rank[f++] = ColoredPiece(white,king); break;
      case 'k' : rank[f++] = ColoredPiece(black,king); break;

      default: throw ParseError("Unknown piece on rank"+c);
    }
  }
}

void parseChar(char ch, stringstream& ss)
{
  char c;
  ss >> c;
  if (c != ch)
    throw ParseError("Syntax Error");
}

void parseColor(stringstream& ss, Color& color)
{
  char c;
  ss >> c;
```

```
  switch (c)
  {
    case 'w': color = white; break;
    case 'b': color = black; break;
    default: throw ParseError("Unknown color");
  }
}

void parseCastling(stringstream& ss,
                   CastlingRight& crw, CastlingRight& crb)
{
  crw = noCastling;
  crb = noCastling;

  char c;
  ss >> c;
  if (c == '-')
    return;
  if (c == 'K') { crw = CastlingRight(crw | kingside); ss >> c; }
  if (c == 'Q') { crw = CastlingRight(crw | queenside); ss >> c; }
  if (c == 'k') { crb = CastlingRight(crb | kingside); ss >> c; }
  if (c == 'q') { crb = CastlingRight(crb | queenside); ss >> c; }

  if (crw == noCastling && crb == noCastling)
    throw ParseError("Unknown castle rights");

  ss.putback(c);
}

void parseEpSquare(stringstream& ss, Square& s)
{
  File f; Rank r;
  char c;

  ss >> c;
  if (c == '-')
  {
    s = noEpSquare;
    return;
  }

  if (c >= 'a' && c <= 'h') // ASCII
    f = File(c - 'a');
  else
    throw ParseError("Unknown square");

  ss >> c;
  if (c >= '1' && c <= '8')
    r = Rank(c - '1');
```

```
  else
    throw ParseError("Unknown square");
  s = toSquare(f,r);
}

void parseHalfmoveClock(stringstream& ss, int& i)
{
  ss >> i;
  if (ss)
    return;
  else
    i = 0;
  //    throw ParseError("Halfmove clock unreadable");
}

void parseFullmoveClock(stringstream& ss)
{
  int i; // dummy
  ss >> i;
  if (ss)
    return;
  else
    i = 0;
  //    throw ParseError("Fullmove clock unreadable");
}

void Position::setupFEN(string FEN)
{
  Position tmpPos(0);
  stringstream ss(FEN);

  try
  {
    for (Rank r = rank8; r >= rank1; --r)
    {
      ColoredPiece rank[Fileend];
      parseRank(ss,rank);

      forall (File,f)
        tmpPos.board_[toSquare(f,r)] = rank[f];

      if (r > rank1)
        parseChar('/',ss);
    }

    parseColor(ss,tmpPos.turn_);

    parseCastling
      (ss,tmpPos.castlingRight_[white],tmpPos.castlingRight_[black]);
```

```
    parseEpSquare(ss,tmpPos.epSquare_);

    parseHalfmoveClock(ss,tmpPos.halfmoveClock_);

    parseFullmoveClock(ss);
  }
  catch (ParseError p)
  {
    cout << "Error: " << p.error() << endl;
    return;
  }

  forall (Square,s)
    board_[s] = tmpPos.board_[s];

  turn_ = tmpPos.turn_;
  epSquare_ = tmpPos.epSquare_;

  forall (Color,c)
    castlingRight_[c] = tmpPos.castlingRight_[c];

  halfmoveClock_ = tmpPos.halfmoveClock_;

  updateBitBoards();

  int m[Colorend];
  forall (Color,c)
    m[c] = 0;

  forall (Square,s)
    m[board_[s].color()] += pieceValue(board_[s].piece());

  forall (Color,c)
    material_[c] = Score(m[c]);

  gamePly_ = 0;

  rootPly_ = 0;

  hashKey_ = calcHashKey(*this);

  checkIntegrity(*this,nullMove,"setupFEN");
}

int totalMoves;

bool inCheck(Color color, const Position& pos)
{
```

```
  return
    (isAttacking(opposite(color),firstBit(pos.bitboard(color,king)),pos));
}

void generateSubtree(Position& pos, int depth, int ply)
{
  Moves moves;
  generateMoves(pos,moves);

  for (int i = 0; i < moves.size(); ++i)
    if (moves[i].capture() == king)
      return;

  for (int i = 0; i < moves.size(); ++i)
  {
    int mcount;
    if (ply == 0)
    {
      cout << moves[i] << ": ";
      mcount = totalMoves;
    }
    pos.makeMove(moves[i]);

    if (depth > 1)
      generateSubtree(pos,depth - 1, ply + 1);
    else
      if (!inCheck(opposite(pos.turn()),pos))
        ++totalMoves;

    pos.unMakeMove(moves[i]);
    if (ply == 0)
    {
      mcount = totalMoves - mcount;
      cout << mcount << endl;
    }
  }
}

void perft(Position& pos, const int depth)
{
  clock_t before = clock();
  while (clock() == before);
  before = clock();

  totalMoves = 0;
  generateSubtree(pos,depth,0);
  clock_t after = clock();

  double timeUsed =
```

```
    (after - before) / static_cast<double>(CLOCKS_PER_SEC);
  cout << "total moves = " << totalMoves
       << "  time=" << timeUsed << endl;
}
```

B.8 random.cpp

```
// $Id: random.cpp,v 1.1 2004/04/22 12:57:11 s958547 Exp $

// The MT19937 PRNG

#include "random.h"

/* Period parameters */
#define N 624
#define M 397
#define MATRIX_A 0x9908b0df   /* constant vector a */
#define UPPER_MASK 0x80000000 /* most significant w-r bits */
#define LOWER_MASK 0x7fffffff /* least significant r bits */

/* Tempering parameters */
#define TEMPERING_MASK_B 0x9d2c5680
#define TEMPERING_MASK_C 0xefc60000
#define TEMPERING_SHIFT_U(y)  (y >> 11)
#define TEMPERING_SHIFT_S(y)  (y << 7)
#define TEMPERING_SHIFT_T(y)  (y << 15)
#define TEMPERING_SHIFT_L(y)  (y >> 18)

static unsigned long mt[N]; /* the array for the state vector  */
static int mti=N+1; /* mti==N+1 means mt[N] is not initialized */

/* initializing the array with a NONZERO seed */
void sgenrand(unsigned long seed)
{
  /* setting initial seeds to mt[N] using         */
  /* the generator Line 25 of Table 1 in          */
  /* [KNUTH 1981, The Art of Computer Programming */
  /*    Vol. 2 (2nd Ed.), pp102]                  */
  mt[0]= seed & 0xffffffff;
  for (mti=1; mti<N; mti++)
    mt[mti] = (69069 * mt[mti-1]) & 0xffffffff;
}

unsigned long genrand()
{
  unsigned long y;
  static unsigned long mag01[2]={0x0, MATRIX_A};
  /* mag01[x] = x * MATRIX_A  for x=0,1 */
```

```
  if (mti >= N) { /* generate N words at one time */
    int kk;

    if (mti == N+1)   /* if sgenrand() has not been called, */
      sgenrand(4357); /* a default initial seed is used   */

    for (kk=0;kk<N-M;kk++) {
      y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
      mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1];
    }
    for (;kk<N-1;kk++) {
      y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
      mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0x1];
    }
    y = (mt[N-1]&UPPER_MASK)|(mt[0]&LOWER_MASK);
    mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1];

    mti = 0;
  }

  y = mt[mti++];
  y ^= TEMPERING_SHIFT_U(y);
  y ^= TEMPERING_SHIFT_S(y) & TEMPERING_MASK_B;
  y ^= TEMPERING_SHIFT_T(y) & TEMPERING_MASK_C;
  y ^= TEMPERING_SHIFT_L(y);

  return y;
}

void initRandom(unsigned long int seed)
{
  sgenrand(seed); // any nonzero integer can be used as a seed (4357)
}

unsigned long random32()
{
  return genrand();
}

uint64 random64()
{
  return ((uint64(random32()) << 32) | random32());
}
```

B.9 search.cpp

```
// $Id: search.cpp,v 1.45 2004/07/26 14:26:10 s958547 Exp $
```

```
#include <iostream>

#include <sys/time.h>

#include "search.h"
#include "moves.h"
#include "evaluate.h"
#include "hash.h"
#include "parallel.h"

using namespace std;

//#define TRACE
#define TDEPTH 2

const Depth checkExtension = Depth(onePly*0.75);
const Depth oneReplyExtension = Depth(onePly*0.75);
const Depth pawnExtension = Depth(onePly*0.75);
const Depth recaptureExtension = Depth(onePly*0.75);

enum Time {oneSecond = 100};

namespace shared
{
  int maxDepth, maxQDepth;
  int historyScore[Colorend][consts::squares][consts::squares];
  Depth iterationDepth;
}

inline std::ostream& operator<<(std::ostream& os, const Time& t)
{
  int time = t;
  int hours = time / 360000;
  time -= hours * 360000;
  int minutes = time / 6000;
  time -= minutes * 6000;
  int seconds = time / 100;
  time -= seconds * 100;
  int hundreths = time;

  os << setfill('0');
  os << setw(2) << hours << ":";
  os << setw(2) << minutes << ":";
  os << setw(2) << seconds << ".";
  os << setw(2) << hundreths;
  os << setfill(' ');
  return os;
}
```

```
class Clock
{
  public:
    Clock()
    {
      timeval tv;
      gettimeofday(&tv,0);
      baseline = tv.tv_sec;
      start = get();
    }

    Time get() const
    {
      timeval tv;
      gettimeofday(&tv,0);
      return Time((tv.tv_sec - baseline)*100 + tv.tv_usec/10000);
    }

    void reset()
    {
      start = get();
    }

    Time elapsed() const
    {
      return Time(get() - start);
    }

  private:
    Time start;
    time_t baseline;
};

Clock clck;

bool drawCheck(const Position &pos, int ply)
{
  if (pos.halfmoveClock() > 0)
  {
    if (pos.halfmoveClock() >= 100)
      return true;

    int treeCount = 0;
    int gameCount = 0;

    for (int i = pos.gamePly_ - 2;
         i >= pos.gamePly_ - pos.halfmoveClock();
         i -= 2)
```

```
    {
      if (pos.gameRecord_[i].hashKey == pos.hashKey())
      {
        if (i > pos.rootPly_)
          ++treeCount; // internal tree repetition
        else
          ++gameCount; // repetitions of a game history positions
      }
    }

    if (gameCount == 0) // internal repetition only
    {
      if (treeCount > 0)
      {
        return true;
      }
    }
    else // repetitions of a game history position
    {
      if ((gameCount + treeCount) >= 2)
      {
        return true;
      }
    }
  }
  return false;
}

void scoreQuiescenceMoves(const Position& pos, Moves& moves)
{
  for (int i = 0; i < moves.size(); ++i)
  {
    int score;
    if (moves[i].promote())
      score = pieceValue(moves[i].promote());
    else
      score = SEE(pos,moves[i]);

    moves.score(i,score);
  }
}

void updateHistoryAndKillers(Node& node, Move m,
                             Color c, Depth d, int ply)
{
  if (m.capture() || m.promote())
    return;

  shared::historyScore[c][m.from()][m.to()] += (d/onePly)*(d/onePly);
```

```
  if (node.killer1[ply] != m)
  {
    node.killer2[ply] = node.killer1[ply];
    node.killer1[ply] = m;
  }
}

void scoreMoves(const Node& node, Moves& moves, int ply, Move hashMove)
{
  const Position& pos = node.pos;

  for (int i = 0; i < moves.size(); ++i)
  {
    if (moves[i] == hashMove)
    {
      moves.score(i,100000000);
      continue;
    }
    if (moves[i].capture() || moves[i].promote())
    {
      int score;
      if (moves[i].promote())
        score = pieceValue(moves[i].promote());
      else
        score = SEE(pos,moves[i]);

      if (score >= 0)
        score += 1000000;
      else
        score -= 1000000;
      moves.score(i,score);
    }
    else
    {
      if (moves[i] == node.killer1[ply])
        moves.score(i,100000);
      else
        if (moves[i] == node.killer2[ply])
          moves.score(i,50000);
      else

      moves.score
        (i,shared::historyScore[pos.turn()]
         [moves[i].from()][moves[i].to()]);
    }
  }
}

#if defined TRACE
```

```
void searchTrace(int ply,Score alpha, Score beta, Move move, string text)
{
  if (ply < TDEPTH)
  {
    for (int t = 0; t < ply; ++t)
      cout << "  ";
    cout << "(" << alpha << "," << beta << ") " << move << ": "
         << text << endl;
  }
}
#else
#define searchTrace(ply,alpha,beta,move,text)
#endif

Score quiescenceSearch(Node &node, Score alpha, Score beta, int ply)
{
  Position& pos = node.pos;

  Variation& pv = node.pv;
  pv.init(ply);

  if (ply > shared::maxQDepth)
    shared::maxQDepth = ply;

  Score eval = evaluate(pos);

  searchTrace(ply,eval,eval,nullMove,"Eval");

  if (eval > alpha)
  {
    if (eval >= beta)
    {
      searchTrace(ply,alpha,beta,nullMove,"Beta cut");

      return eval;
    }
    alpha = eval;
  }

  Moves& moves = node.moves[ply];
  moves.reset();
  int& i = node.nextMove[ply];
  generateNonQuietMoves(pos,moves);
  scoreQuiescenceMoves(pos,moves);

  int legalMoves = 0;

  for (i = 0; i < moves.size(); ++i)
  {
```

```
moves.sortFrom(i);
if (moves.getScore(i) < 0)
{
  break;
}

const Score margin = pieceValue(pawn);

Score optimisticExpectation = Score(
  (pos.material(pos.turn()) - pos.material(opposite(pos.turn())))
  + moves.getScore(i) + margin);

if (optimisticExpectation <= alpha)
{
  break;
}

++node.nodes;
++node.qNodes;

searchTrace(ply,alpha,beta,moves[i],"Qmove");

pos.makeMove(moves[i]);

if (inCheck(opposite(pos.turn()),pos))
{
  pos.unMakeMove(moves[i]);
  continue;
}

++legalMoves;

Score value =
  (Score)(-quiescenceSearch(node, Score(-beta), Score(-alpha),
                            ply + 1));

pos.unMakeMove(moves[i]);

if (value > alpha)
{
  if (value >= beta)
  {
    return value;
  }
  alpha = value;
  pv.update(ply,moves[i]);
}
if (node.stopped)
  return draw;
```

```
  }

  searchTrace(ply,alpha,beta,nullMove,"Qreturn");

  return alpha;
}

TransRefTable transRef;

Score search(Node &node, Score alpha, Score beta, Depth depth,
             int ply, bool nullMoveIsOK, Depth totalExtension)
{
  Position& pos = node.pos;

  ++node.nodes;

  Variation& pv = node.pv;
  pv.init(ply);

  if (drawCheck(pos,ply))
  {
    searchTrace(ply,alpha,beta,nullMove,"Repetition");

    return draw;
  }

  if (ply > shared::maxDepth)
    shared::maxDepth = ply;

  bool inChk = inCheck(pos.turn(),pos);

  //
  // Extensions
  //

  Depth extension = Depth(0);

  if (inChk)
  {
    extension += checkExtension;

    Moves evasions;
    generateMoves(pos,evasions);
    evasions.removeIllegal(pos);
    if (evasions.size() == 1)
      extension += oneReplyExtension;
  }

  const Move lastMove = pos.gameRecord_[pos.gamePly_ - 1].move;
```

```
const Move previousMove = pos.gameRecord_[pos.gamePly_ - 2].move;
if (lastMove.piece() == pawn)
{
  if (pos.turn() == white)
  {
    if (getRank(lastMove.to()) >= rank7)
      extension += pawnExtension;
  }
  else
  {
    if (getRank(lastMove.to()) <= rank2)
      extension += pawnExtension;
  }
}

if (ply > 1
    && (lastMove.to() == previousMove.to())
    && (pieceValue(lastMove.capture())
        == pieceValue(previousMove.capture()))
  )
{
  extension += recaptureExtension;
}

Depth maxExtension = Depth(ply*onePly - totalExtension);
if (extension > maxExtension)
  extension = maxExtension;

if (ply*onePly > 2*shared::iterationDepth)
  extension = Depth(extension/2);

depth += extension;
totalExtension += extension;

if (depth < onePly)
  return quiescenceSearch(node,alpha,beta,ply);

//
// Probe the hashtable
//

Move hashMove = nullMove;

switch(transRef.probe(node,depth,ply,alpha,beta,hashMove))
{
  case exact:
    return alpha;
    break;
  case lowerBound:
```

```
    return beta;
    break;
  case upperBound:
    return alpha;
    break;
}

//
// Null move pruning
//

Score pieceMaterial = Score(pos.material(pos.turn())
  - pieceValue(pawn)*popCount(pos.bitboard(pos.turn(),pawn)));

if (nullMoveIsOK && !inChk && pieceMaterial >= 6*pieceValue(pawn))
{
  Depth R;
  if (depth >= 7*onePly)
    R = Depth(3*onePly);
  else
    R = Depth(2*onePly);

  pos.makeNullMove();

  Score value =
    Score(-search(node, Score(-beta), Score(-beta + 1),
                  Depth(depth - onePly - R),  ply + 1, false,
                  totalExtension));

  pos.unMakeNullMove();

  if (node.stopped)
    return draw;

  if (value >= beta)
  {
    transRef.store(pos,depth,ply,lowerBound,value,nullMove);
    return value;
  }
}

// The main search loop

Moves& moves = node.moves[ply];
moves.reset();
int& i = node.nextMove[ply];

generateMoves(pos,moves);
scoreMoves(node,moves,ply,hashMove);
```

```
int legalMoves = 0;
bool isExact = false;
Move bestMove = nullMove;
for (i = 0; i < moves.size(); ++i)
{

  moves.sortFrom(i);

  searchTrace(ply,alpha,beta,moves[i],"Move");

  pos.makeMove(moves[i]);

  if (inCheck(opposite(pos.turn()),pos))
  {
    pos.unMakeMove(moves[i]);
    continue;
  }

  ++legalMoves;

  Score value;
  if (i == 0)
  {
    value = (Score)(-search(node, Score(-beta), Score(-alpha),
                            Depth(depth - onePly), ply + 1, true,
                            totalExtension));
  }
  else
  {
    value = Score(-search(node, Score(-alpha - 1), Score(-alpha),
                          Depth(depth - onePly), ply + 1, true,
                          totalExtension));

    if (node.stopped)
    {
      pos.unMakeMove(moves[i]);
      return draw;
    }

    if (value > alpha && value < beta)
    {
      value = (Score)(-search(node, Score(-beta), Score(-alpha),
                              Depth(depth - onePly), ply + 1, true,
                              totalExtension));
    }
  }

  pos.unMakeMove(moves[i]);
```

```
if (node.stopped)
  return draw;

if (value > alpha)
{
  if (value >= beta)
  {
    ++node.failHighs;
    if (legalMoves == 1)
      ++node.failHighFirsts;

    updateHistoryAndKillers(node,moves[i],pos.turn(),depth,ply);

    transRef.store(pos,depth,ply,lowerBound,value,moves[i]);

    searchTrace(ply,alpha,beta,moves[i],"Beta cut");

    return value;
  }
  alpha = value;
  pv.update(ply,moves[i]);
  bestMove = moves[i];
  isExact = true;
}

const Depth minSplitDepth = Depth(3*onePly);

if (idleThreads && legalMoves && depth >= minSplitDepth)
{
  node.alpha = alpha;
  node.beta = beta;
  node.ply = ply;
  node.depth = depth;
  node.totalExtension = totalExtension;

  bool success = split(&node);

  if (success)
  {
    if (node.stopped)
      return draw;

    value = node.pValue;
    if (value > alpha)
    {
      bestMove = node.bestMove;

      if (value >= beta)
```

```
          {
            ++node.failHighs;

            updateHistoryAndKillers(node,bestMove,pos.turn(),depth,ply);

            transRef.store(pos,depth,ply,lowerBound,value,bestMove);

            searchTrace(ply,alpha,beta,bestMove,"Beta cut");

            return value;
          }

          alpha = value;
          isExact = true;
        }
        break;
      }
    }
  }

  if (legalMoves == 0)
  {
    if (inChk)
      alpha = Score(-mate + ply);
    else
      alpha = draw;
  }

  if (isExact)
    transRef.store(pos,depth,ply,exact,alpha,bestMove);
  else
    transRef.store(pos,depth,ply,upperBound,alpha,nullMove);

  return alpha;
}

Score rootSearch(Node &node, Score alpha, Score beta, Depth depth,
 Moves& moves, Move& bestmove)
{
  Position& pos = node.pos;

  const int ply = 0;
  shared::maxDepth = 0;
  shared::maxQDepth = 0;

  Variation& pv = node.pv;
  pv.reset();

  bool isExact = false;
```

```
Move bestMove = nullMove;
for (int i = 0; i < moves.size(); ++i)
{
  moves.sortFrom(i);

  searchTrace(ply,alpha,beta,moves[i],"Rmove");

  unsigned long long nodesBefore = node.nodes + node.qNodes;

  pos.makeMove(moves[i]);

  Score value;
  if (i == 0)
    value = (Score)(-search(node, Score(-beta), Score(-alpha),
                            Depth(depth - onePly), ply + 1, true,
                            Depth(0)));
  else
  {
    value = Score(-search(node, Score(-alpha - 1), Score(-alpha),
                          Depth(depth - onePly), ply + 1, true,
                          Depth(0)));

    if (value > alpha && value < beta)
    {
      value = (Score)(-search(node, Score(-beta), Score(-alpha),
                              Depth(depth - onePly), ply + 1, true,
                              Depth(0)));
    }
  }

  pos.unMakeMove(moves[i]);

  moves.score(i,node.nodes + node.qNodes - nodesBefore);

  if (value > alpha)
  {
    moves.score(i,node.nodes + node.qNodes);
    if (value >= beta)
    {
      ++node.failHighs;
      if (i == 0)
        ++node.failHighFirsts;

      updateHistoryAndKillers(node,moves[i],pos.turn(),depth,ply);

      transRef.store(pos,depth,ply,lowerBound,value,moves[i]);

      return value;
    }
```

```
      alpha = value;
      pv.update(ply,moves[i]);
      isExact = true;
      bestMove = moves[i];
      cout << setw(11) << clck.elapsed()  << setw(14) << node.nodes << " "
           << setfill('0')
           << setw(2) << depth/onePly << "|"
           << setw(2) << shared::maxDepth << "|"
           << setw(2) << shared::maxQDepth
           << " " << alpha
           << " "  << pv << newline;
    }
  }

  bestmove = pv.bestMove();

  if (isExact)
    transRef.store(pos,depth,ply,exact,alpha,bestMove);
  else
    transRef.store(pos,depth,ply,upperBound,alpha,nullMove);

  return alpha;
}

void storePV(Position& pos,const Variation& pv)
{
  for (int ply = 0; ply < pv.length(); ++ply)
  {
    transRef.store(pos,Depth(0),0,exact,illegal,pv[ply]);
    pos.makeMove(pv[ply]);
  }

  for (int ply = pv.length() - 1; ply >= 0; ply--)
  {
    pos.unMakeMove(pv[ply]);
  }
}

void think(Node& rootNode)
{

  Position& pos = rootNode.pos;

  rootNode.nodes = 0;
  rootNode.qNodes = 0;
  rootNode.failHighs = 0;
  rootNode.failHighFirsts = 0;

  clck.reset();
```

```
forall (Color,c)
  forall (Square,from)
    forall (Square,to)
      shared::historyScore[c][from][to] = 0;

for (int ply = 0; ply < maxPly; ++ply)
  rootNode.killer1[ply] = rootNode.killer2[ply] = nullMove;

rootNode.transRefProbes = 0;
rootNode.transRefHits = 0;

transRef.nextGeneration();

Moves moves;
generateMoves(pos,moves);
moves.removeIllegal(pos);
for (int i = 0; i < moves.size(); ++i)
  moves.score(i,0);

cout << "time         nodes    depth    score pv" << newline;

Move bestMove;
Score value =
  Score(pos.material(pos.turn())
        - pos.material(opposite(pos.turn())));

for (shared::iterationDepth = Depth(1*onePly);
     shared::iterationDepth <= 11*onePly;
     shared::iterationDepth += onePly)
{
  Score alpha = Score(value - pieceValue(pawn)/3);
  Score beta = Score(value + pieceValue(pawn)/3);

  value = rootSearch
    (rootNode,alpha,beta,shared::iterationDepth,moves,bestMove);

  if (value <= alpha || value >= beta)
  {
    if (value >= beta)
      cout << "                     ++" << endl;
    else
      cout << "                     --" << endl;
    value = rootSearch(rootNode,Score(-mate),mate,
                       shared::iterationDepth,moves,bestMove);
  }

  cout << setw(11) << clck.elapsed() << setw(14) << rootNode.nodes
       << " ---------------------------------------------------"
```

```
        << newline;

//    storePV(pos,pv);
  }

  Time elapsed = clck.elapsed();

  cout << newline;
  cout << "   time used: " << elapsed << newline;
  cout << "   nps: " << int(rootNode.nodes / (elapsed/double(100)));
  cout << "   nodes: " << rootNode.nodes;
  cout << "   Q-nodes: " << rootNode.qNodes;
  cout << " (" << int(rootNode.qNodes/(double(rootNode.nodes)/100)) <<
    "%)" << newline;

  cout << "   move order: "
       << int(100*(rootNode.failHighFirsts/double(rootNode.failHighs)))
       << "%" << newline;

  cout << "   hash probes: " << rootNode.transRefProbes
       << "   hash hits: " << rootNode.transRefHits
       << " (" << int(rootNode.transRefHits/
                      (double(rootNode.transRefProbes+1)/100))
       << "%)" << newline;

  cout << "   hash usage: "
       << int(transRef.getUsage()/(double(2*transRef.getSize()+1)/100))
       << "%" << newline;

  cout << newline << "My move: " << bestMove << newline << newline;
  pos.makeMove(bestMove);
}
```

B.10 base.h

```
// $Id: base.h,v 1.15 2004/05/15 00:30:53 s958547 Exp $

#if !defined BASE_H__
#define BASE_H__

#include <iostream>
#include <cstdlib>
#include <iomanip>
#include <algorithm>

#include "enums.h"

namespace consts
```

```
{
  const int files = 8;
  const int ranks = 8;
  const int squares = files*ranks;
  const int max_moves = 300;
}

const int maxPly = 99;

typedef unsigned long long int uint64;

enum Color {white,black,
            Colorbegin = white,Colorend = black + 1};
DeclareEnumTricks(Color)

inline Color opposite(Color c)
{
  return c == white ? black : white;
}

inline std::ostream& operator<<(std::ostream& os, const Color c)
{
  os << (c == white ? "white" : "black");
  return os;
}

enum Piece {none,pawn,knight,bishop,rook,queen,king,
            Piecebegin = none, Pieceend = king + 1};
DeclareEnumTricks(Piece)

inline char letter(Piece p)
{
  static const char letters[] = {" PNBRQK"};
  return letters[p];
}

inline std::ostream& operator<<(std::ostream& os, const Piece p)
{
  os << letter(p);
  return os;
}

enum Square
{
  a1,b1,c1,d1,e1,f1,g1,h1,
  a2,b2,c2,d2,e2,f2,g2,h2,
  a3,b3,c3,d3,e3,f3,g3,h3,
  a4,b4,c4,d4,e4,f4,g4,h4,
  a5,b5,c5,d5,e5,f5,g5,h5,
```

```
  a6,b6,c6,d6,e6,f6,g6,h6,
  a7,b7,c7,d7,e7,f7,g7,h7,
  a8,b8,c8,d8,e8,f8,g8,h8,
  Squarebegin = a1, Squareend = h8 + 1
};
DeclareEnumTricks(Square)

enum File {fileA,fileB,fileC,fileD,fileE,fileF,fileG,fileH,
           Filebegin = fileA, Fileend = fileH + 1};
DeclareEnumTricks(File)

enum Rank {rank1,rank2,rank3,rank4,rank5,rank6,rank7,rank8,
           Rankbegin = rank1, Rankend = rank8 + 1};
DeclareEnumTricks(Rank)

inline Square toSquare(File f, Rank r)
{
  return Square(r*consts::files + f);
}

inline File getFile(Square s)
{
  return File(s & 7);
}

inline Rank getRank(Square s)
{
  return Rank(s >> 3);
}

inline std::ostream& operator<<(std::ostream& os, const File f)
{
  os << char('a'+f);
  return os;
}

inline std::ostream& operator<<(std::ostream& os, const Rank r)
{
  os << char('1'+r);
  return os;
}

inline std::ostream& operator<<(std::ostream& os, const Square s)
{
  os << getFile(s) << getRank(s);
  return os;
}

class ColoredPiece
```

```
{
  public:
    ColoredPiece() {}
    ColoredPiece(Color c, Piece p) : color_(c), piece_(p) {}
    Color color() const { return color_; }
    Piece piece() const { return piece_; }
  private:
    Color color_;
    Piece piece_;
};

const ColoredPiece emptySquare = ColoredPiece(white,none);

inline std::ostream& operator<<(std::ostream& os, const ColoredPiece& cp)
{
  if (cp.color() == white)
    os << " ";
  else
    os << "*";
  os << cp.piece();
  return os;
}

//inline int max(int i, int j)
//{
//  return (i > j ? i : j);
//}

inline int fileDist(Square s1, Square s2)
{
  return abs(getFile(s1) - getFile(s2));
}

inline int rankDist(Square s1, Square s2)
{
  return abs(getRank(s1) - getRank(s2));
}

inline int dist(Square s1, Square s2)
{
  return std::max(fileDist(s1,s2), rankDist(s1,s2));
}

enum Score
{
  minScore = -131072,
  draw = 0,
  pawnValue = 1000,
  mate = 130000,
```

```
  illegal = 130100,
  maxScore = +131072
};

inline Score& operator +=(Score& lhs, Score rhs )
{
    lhs = static_cast<Score>( lhs + rhs );
    return lhs;
}

inline Score& operator -=(Score& lhs, Score rhs )
{
    lhs = static_cast<Score>( lhs - rhs );
    return lhs;
}

inline std::ostream& operator<<(std::ostream& os, const Score& score)
{
  using namespace std;
  if (abs(score) < mate - 300)
  {
    os.setf(ios::fixed, ios::floatfield);
    os.setf(ios::right, ios::adjustfield);
    os << setiosflags(ios::showpos);
    os << setw(4) << setprecision(3)
         << double(score) / 1000;
    os.unsetf(ios::showpos);
  }
  else
  {
    if (score > 0)
      os << "+";
    else
      os << "-";
    os << "MAT" << (mate - (abs(score)));
    if (mate - (abs(score)) < 10)
      os << " ";
  }
  return os;
}

enum Depth {onePly = 100};

inline Depth& operator +=(Depth& lhs, Depth rhs )
{
    lhs = static_cast<Depth>( lhs + rhs );
    return lhs;
}
```

```
inline std::ostream& newline(std::ostream& os)
{
  return os << '\n';
}

typedef uint64 HashKey;

#endif
```

B.11 bitboard.h

```
// $Id: bitboard.h,v 1.15 2004/07/26 14:26:10 s958547 Exp $

#if !defined BITBOARD_H__
#define BITBOARD_H__

#include <iostream>

#include "base.h"

#if defined NDEBUG

typedef uint64 BitBoard;

#else

class BitBoard
{
  public:
    BitBoard() { undefined = true; }
    explicit BitBoard(uint64 ulli) :
      bb(ulli) { undefined = false; }

    operator uint64() { return bb; }

    BitBoard& operator &=(const BitBoard& rhs)
      { c(); bb &= rhs.bb; return *this; }
    BitBoard& operator |=(const BitBoard& rhs)
      { c(); bb |= rhs.bb; return *this; }
    BitBoard& operator ^=(const BitBoard& rhs)
      { c(); bb ^= rhs.bb; return *this; }

    BitBoard operator &(const BitBoard& rhs) const
    { c(); return BitBoard(bb & rhs.bb); }
    BitBoard operator |(const BitBoard& rhs) const
    { c(); return BitBoard(bb | rhs.bb); }
    BitBoard operator ^(const BitBoard& rhs) const
    { c(); return BitBoard(bb ^ rhs.bb); }
```

```
    BitBoard operator ~() const
    { c(); return BitBoard(~bb); }
    BitBoard operator <<(int i) const
    { c(); return BitBoard(bb << i); }
    BitBoard operator >>(int i) const
    { c(); return BitBoard(bb >> i); }
    bool operator ==(const BitBoard& rhs) const
    { return (bb == rhs.bb); }

  private:
    void c() const
    {
      if (undefined)
        std::cout << "Uninitialized BitBoard read from:"
                  << bb << std::endl;
    }
    bool undefined;
    uint64 bb;
};

#endif

const BitBoard empty = BitBoard(0);

//#define CALCMASK
#if defined CALCMASK
inline BitBoard mask(Square s)
{
  return BitBoard(1ULL << s);
}
#else
extern BitBoard mask_[Squareend];

inline BitBoard mask(Square s)
{
  return mask_[s];
}
#endif

extern int firstBit_[65536];

inline Square firstBit(BitBoard b)
{
  if (int(b) & 0xffff)
    return Square(firstBit_[int(b) & 0xffff]);
  else if (int(b >> 16) & 0xffff)
    return Square(firstBit_[int(b >> 16) & 0xffff] + 16);
  else if (int(b >> 32) & 0xffff)
    return Square(firstBit_[int(b >> 32) & 0xffff] + 32);
```

```
  else if (int(b >> 48) & 0xffff)
    return Square(firstBit_[int(b >> 48) & 0xffff] + 48);

  return Square_end();
}

inline int popCount(BitBoard bitboard)
{
  int count = 0;
  while(bitboard)
  {
    ++count;
    bitboard &= BitBoard(bitboard - 1);
  }
  return count;
}

inline void clear(BitBoard& b, Square s)
{
  b &= ~mask(s);
}

inline void set(BitBoard& b, Square s)
{
  b |= mask(s);
}

#if !defined NDEBUG
std::ostream& operator<<(std::ostream& os, const BitBoard b);
#endif

void initBitBoard();

#endif
```

B.12 enums.h

```
// $Id: enums.h,v 1.2 2004/02/15 11:20:55 s958547 Exp $

#if !defined ENUMS_H__
#define ENUMS_H__

#define DeclareEnumTricks(T) \
inline T& operator++(T& e) \
{ \
    e = T(e+1); \
    return e; \
} \
```

```
\
inline T operator++(T& e, int)  \
{ \
    T old = e; \
    e = T(e+1); \
    return old;\
} \
\
inline T& operator--(T& e) \
{ \
    e = T(e-1); \
    return e; \
} \
\
inline T operator--(T& e, int)  \
{ \
    T old = e; \
    e = T(e-1); \
    return old; \
} \
\
inline T T##_end() \
{ \
    return T##end; \
} \
\
inline T T##_begin() \
{ \
    return T##begin; \
}

#define forall(Enum,e) for (Enum e = Enum##begin; e != Enum##end; ++e)

#endif
```

B.13 evaluate.h

```
// $Id: evaluate.h,v 1.2 2004/02/24 12:09:20 s958547 Exp $

#if !defined EVALUATE_H__
#define EVALUATE_H__

#include "position.h"

inline Score pieceValue(Piece p)
{

  const Score pieceValue_[Pieceend] =
```

```
    {
      Score(0),
      pawnValue,
      Score(pawnValue*3),
      Score(pawnValue*3),
      Score(pawnValue*5),
      Score(pawnValue*9),
      Score(0)
    };

  return pieceValue_[p];
}

Score evaluate(const Position& pos);

#endif
```

B.14 hash.h

```
// $Id: hash.h,v 1.7 2004/06/16 11:27:44 s958547 Exp $

#if !defined HASH_H__
#define HASH_H__

#include <iostream>

#include "base.h"
#include "move.h"
#include "position.h"
#include "parallel.h"

enum HashEntryType {noHash,exact,upperBound,lowerBound};

class TransRefTable
{
  public:
    TransRefTable();

    HashEntryType probe(Node& node, Depth depth, int ply,
                        Score& alpha, Score& beta,
                        Move& bestMove);

    void store(const Position &pos, Depth depth, int ply,
               HashEntryType type, Score score, Move bestMove);

    void nextGeneration()
    {
      hashID = (hashID + 1) & 7;
```

```
      if (hashID == 0)
        ++hashID;
    }

    void clear();

    int getUsage()
    {
      int count = 0;
      for (int i = 0; i < hashTableSize; ++i)
      {
        if (hashTableWhite[i].data.age == hashID)
          ++count;
        if (hashTableBlack[i].data.age == hashID)
          ++count;
      }
      return count;
    }

    int getSize() { return hashTableSize; }

  private:
    // Asumption: sizeof(unsigned long) == 8 (64 bits)
    struct HashEntry
    {
        union
        {
            struct
            {
                unsigned long score : 18;
                unsigned long depth : 14;

                unsigned long move : 21;
                unsigned long age : 3;
                unsigned long type : 2;
            } data;
            unsigned long dataul64;
        };

        HashKey hashKey;
    };

    bool hashEnabled;
    int hashTableSize;
    int hashTableSplit;
    HashEntry* hashTableWhite;
    HashEntry* hashTableBlack;
    unsigned int hashID;
};
```

```
void initHash();
HashKey calcHashKey(const Position& pos);
HashKey hashPiece(Color c, Piece p, Square s);
HashKey hashEpSquare(Square s);
HashKey hashCastling(Color c, CastlingRight cr);

#endif
```

B.15 move.h

```
// $Id: move.h,v 1.8 2004/05/19 20:42:09 s958547 Exp $

#if !defined MOVE_H__
#define MOVE_H__

#include "base.h"

class Move
{
  public:
    static Move nullMove;
    Move() {}
    explicit Move(unsigned int m) : move(m) {}
    Move(Square from, Square to, Piece piece,
         Piece capture, Piece promote) :
      move  (((((((promote << 3) | capture) << 3)
                 | piece) << 6) | to) << 6) | from)
    {
//      move = (((((((promote << 3) | capture) << 3)
//                 | piece) << 6) | to) << 6) | from);
    }

    Square from()   const { return Square(move & 0x3F); }
    Square to()     const { return Square((move >> 6) & 0x3F); }
    Piece piece()   const { return Piece((move >> 12) & 0x07); }
    Piece capture() const { return Piece((move >> 15) & 0x07); }
    Piece promote() const { return Piece((move >> 18) & 0x07); }
    bool operator !=(const Move m) const { return move != m.move; }
    bool operator ==(const Move m) const { return move == m.move; }
    unsigned int representation() { return move; }
  private:
    unsigned int move;
};

const Move nullMove = Move(0);
const Move illegalMove = Move(1);
```

```
#endif
```

B.16 moves.h

```
// $Id: moves.h,v 1.14 2004/07/26 14:26:10 s958547 Exp $

#if !defined MOVES_H__
#define MOVES_H__

#include <iostream>
#include <algorithm>

#include "base.h"
#include "position.h"
#include "move.h"
#include "evaluate.h"

class Moves
{
  public:
    Moves() { sz = 0; }
    void push_back(Move m) { mvs[sz++].move = m; }
    int size() const { return sz; }
    Move operator[](int i) const { return mvs[i].move; }
    void score(int i, int score) { mvs[i].score = score; }
    int getScore(int i) const { return mvs[i].score; }

    void sortFrom(int from)
    {
      int best = from;
      for (int i = from+1; i < sz; ++i)
        if (mvs[i].score > mvs[best].score)
          best = i;
      if (best != from)
        std::swap(mvs[from],mvs[best]);
    }

    void removeIllegal(Position& pos)
    {
      for (int i = 0; i < sz; ++i)
      {
        pos.makeMove(mvs[i].move);
        bool illegal = inCheck(opposite(pos.turn()),pos);
        pos.unMakeMove(mvs[i].move);
        if (illegal)
        {
          mvs[i] = mvs[--sz];
          --i;
```

```
        }
      }
    }
    void reset() { sz = 0; }
    void printScores()
    {
      std::cout << sz << " moves: ";
      for (int i = 0; i < sz; ++i)
        std::cout << mvs[i].score << " ";
      std::cout << std::endl;
    }
    void checkSort(int to)
    {
      for (int i = 0; i < to && i+1 < sz; ++i)
        if (mvs[i].score < mvs[i+1].score)
          std::cout << "doh!" << std::endl;
    }
  private:
    int sz;
    struct ScoredMove
    {
        Move move;
        int score;
    };
    ScoredMove mvs[consts::max_moves];
};

std::ostream& operator<<(std::ostream& os, const Move m);

class Variation
{
  public:
    Variation() {}
    void reset() { length_[0] = 0; }
    void init(int ply) { length_[ply] = ply; }
    void update(int ply, Move m)
    {
      v_[ply][ply] = m;
      for (int j = ply + 1; j < length_[ply + 1]; ++j)
        v_[ply][j] = v_[ply + 1][j];
      length_[ply] = length_[ply + 1];
    }
    Move bestMove() { return v_[0][0]; }
    int length() const { return length_[0]; }
    Move operator[](int i) const { return v_[0][i]; }
    void copy(const Variation& pv, int ply)
    {
      for (int i = ply; i < pv.length_[ply]; ++i)
        v_[ply][i] = pv.v_[ply][i];
```

```
      length_[ply] = pv.length_[ply];

    }

    Score check(int ply, Position& pos)
    {
      if (length_[ply]-ply <= 0)
      {
        std::cout << "zero length pv!" << std::endl;
      }

      Color turn = pos.turn();
      for (int i = ply; i < length_[ply]; ++i)
        pos.makeMove(v_[ply][i]);

      Score eval = evaluate(pos);
      if (turn != pos.turn())
        eval = Score(-eval);

      for (int i = length_[ply] - 1; i >= ply; --i)
        pos.unMakeMove(v_[ply][i]);

      return eval;
    }

    void print(int ply)
    {
      std::cout << "[" << length_[ply]-ply << "] = ";
      for (int i = ply; i < length_[ply]; ++i)
        std::cout << v_[ply][i] << " ";
      std::cout << std::endl;
    }
  private:
    int length_[maxPly];
    Move v_[maxPly][maxPly];
};

void initMoves();
void generateMoves(const Position& pos, Moves& moves);
void generateNonQuietMoves(const Position& pos, Moves& moves);
Square rotate45Left(Square s);
Square rotate45Right(Square s);
Square rotate90Left(Square s);
bool isAttacking(Color color, Square s, const Position& pos);
Score SEE(const Position &pos, Move move);

inline std::ostream& operator<<(std::ostream& os, const Variation& v)
{
  for (int i = 0; i < v.length(); ++i)
```

```
    os << v[i] << " ";
  return os;
}

#endif
```

B.17 parallel.h

```
// $Id: parallel.h,v 1.18 2004/06/18 11:54:26 s958547 Exp $

#if !defined PARALLEL_H__
#define PARALLEL_H__

#include <pthread.h>
#include "position.h"
#include "base.h"
#include "moves.h"

const int maxThreads = 24;
const int splitBlocksPerThread = 64;
const int nSplitBlocks = maxThreads*splitBlocksPerThread;

extern int nThreads;
extern int maxThreadsPerNode;

struct Node
{
    Node* parent;
    Node* children[maxThreads];
    volatile bool stopped;
    volatile int workers;
    bool used;
    int threadNumber;
    pthread_mutex_t mutex;

    Position pos;
    Variation pv;
    int nodes;
    int qNodes;
    int failHighs;
    int failHighFirsts;
    int transRefProbes;
    int transRefHits;
    Move killer1[maxPly],killer2[maxPly];
    Score pValue;
    Score cValue;
    Score alpha;
    Score beta;
```

```
    Depth depth;
    int ply;
    Depth totalExtension;
    Moves moves[maxPly];
    int nextMove[maxPly];
    Move bestMove;
};

void initParallel();
bool split(Node* node);

extern volatile int idleThreads;
extern Node* rNode;

#endif
```

B.18 position.h

```
// $Id: position.h,v 1.25 2004/07/26 14:26:10 s958547 Exp $

#if !defined POSITION_H__
#define POSITION_H__

#include <iostream>
#include <string>

#include "bitboard.h"
#include "move.h"

enum CastlingRight {noCastling,kingside,queenside,bothsides,
                    CastlingRightbegin = noCastling,
                    CastlingRightend = bothsides + 1};
DeclareEnumTricks(CastlingRight)

const Square noEpSquare = a1;

class Position
{
  public:
    Position()
    {
      setupFEN
        ("rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1");
    }

    Position(std::string FEN)
    {
      setupFEN(FEN);
```

```
}

explicit Position(int) // dummy argument
{
}

void makeMove(Move m);
void unMakeMove(Move m);
void makeNullMove();
void unMakeNullMove();

void setupFEN(std::string FEN);

std::string getFEN() const;

void initial();

ColoredPiece operator[](Square s) const
{
  return board_[s];
}

BitBoard bitboard(Color c, Piece p) const
{
  return bitboard_[c][p];
}

BitBoard occupied(Color c) const
{
  return occupied_[c];
}

BitBoard occupied() const
{
  return occupied_[white] | occupied_[black];
}

BitBoard nonOccupied() const
{
  return ~(occupied_[white] | occupied_[black]);
}

BitBoard occupiedRotate45Left() const
  { return occupiedRotate45Left_; }

BitBoard occupiedRotate45Right() const
  { return occupiedRotate45Right_; }

BitBoard occupiedRotate90Left() const
```

```
      { return occupiedRotate90Left_; }

    Color turn() const { return turn_; }

    Square epSquare() const { return epSquare_; }

    CastlingRight castlingRight(Color c) const
      { return castlingRight_[c]; }

    int halfmoveClock() const { return halfmoveClock_; }

    Score material(Color c) const { return material_[c]; }

    HashKey hashKey() const { return hashKey_; }

// data:

    struct HistoryEntry
    {
        Square epSquare;
        CastlingRight castlingRight[Colorend];
        int halfmoveClock;
        Move move;
        HashKey hashKey;
    };

    int gamePly_;
    HistoryEntry gameRecord_[50]; // not enough for real play,
                                  // but enough for testing

    int rootPly_;

    int threadNumber;

  private:

    void updateBitBoards();

    ColoredPiece board_[consts::squares];
    BitBoard bitboard_[Colorend][Pieceend];
    BitBoard occupied_[Colorend];
    BitBoard occupiedRotate45Left_;
    BitBoard occupiedRotate45Right_;
    BitBoard occupiedRotate90Left_;

    Color turn_;
    Square epSquare_;
    CastlingRight castlingRight_[Colorend];
    int halfmoveClock_;
```

```
    Score material_[Colorend];

    HashKey hashKey_;
};

std::ostream& operator<<(std::ostream& os, const Position& pos);
void perft(Position& pos, const int depth);
bool inCheck(Color color, const Position& pos);

#endif
```

B.19 random.h

```
// $Id: random.h,v 1.1 2004/04/22 12:57:11 s958547 Exp $

#if !defined RANDOM_H__
#define RANDOM_H__

#include "base.h"

void initRandom(unsigned long int seed);
uint64 random64();

#endif
```

B.20 search.h

```
// $Id: search.h,v 1.6 2004/05/15 22:41:45 s958547 Exp $

#if !defined SEARCH_H__
#define SEARCH_H__

#include "position.h"
#include "parallel.h"

void think(Node& rootNode);
Score search(Node &node, Score alpha, Score beta, Depth depth,
             int ply, bool nullMoveIsOK, Depth totalExtension);

#endif
```

Bibliography

[1] Adelson-Velskiy, G. M., Arlazarov, V. L., Bitman, A. R., Zhivotovsky, A. A., and Uskov, A. V. Programming a Computer to Play Chess. *Russian Mathematical Surveys 25* (1970), 221–262.

[2] Akl, S. G., and Newborn, M. M. The principal continuation and the killer heuristic. *ACM '77 National Conference* (1977), 466–473.

[3] Beal, D. F. Experiments with the null move. In *Advances in Computer Chess 5* (1989), Elsevier Science Publishers, pp. 65–79.

[4] Beal, D. F. A generalised quiescence search algorithm. *Artificial Intelligence 43*, 1 (1990), 85–98.

[5] Berliner, H. J. Some innovations introduced by hitech. *ICCA Journal 10*, 3 (1987), 111–117.

[6] Bettadapur, P. Influence of Ordering on Capture Search. *ICCA Journal 9*, 4 (1986), 180–188.

[7] Bratko, I., and Kopec, D. The Bratko-Kopec Experiment: A Comparison of Human and Computer Performance in Chess. In *Advances in Computer Chess 3* (1982), Pergamon Press, pp. 57–72.

[8] Feldmann, R., Mysliwietz, P., and Monien, B. Game tree search on a massively parallel system. In *Advances in Computer Chess 7* (1994), University of Limburg, pp. 203–218.

[9] Hartmann, D. Notions of Evaluation Functions Tested Against Grandmaster Games. In *Advances in Computer Chess 5* (1989), Elsevier Science Publishers, pp. 91–141.

[10] Heinz, E. How DarkThought Plays Chess. *ICCA Journal 20*, 3 (1997), 166–176.

[11] Hyatt, R. M. A lockless transposition table implementation for parallel search. *ICGA Journal 25*, 1 (2002), 36–39.

[12] Hyatt, R. M., Gower, A. E., and Nelson, H. L. Cray Blitz. In *Computers, Chess, and Cognition*, T. A. Marsland and J. Schaeffer, Eds. Springer-Verlag, 1990, pp. 111–130.

[13] Knuth, D. E., and Moore, R. W. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence 6* (1975), 293–326.

[14] Levy, D. N. L., Broughton, D., and Taylor, M. The SEX algorithm in computer chess. *ICCA Journal 12*, 1 (1989), 10–21.

[15] Marsland, T. A., and Campbell, M. Parallel search of strongly ordered game trees. *ACM Computing Survey 14*, 4 (1982), 533–551.

[16] Marsland, T. A., and Popowich, F. Parallel game-tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-7*, 4 (1985), 442–452.

[17] Matsumoto, M., and Nishimura, T. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation 8*, 1 (Jan. 1998), 3–30.

[18] Newell, A., Shaw, J. C., and Simon, H. A. Chess-playing programs and the problem of complexity. *IBM Journal of Research and Development 2* (1958), 320–335.

[19] Schaeffer, J. The history heuristic. *ICCA Journal 6*, 3 (1983), 16–19.

[20] Shams, R., Kaindl, H., and Horacek, H. Using aspiration windows for minimax algorithms. In *Proceedings of the 12th IJCAI* (Sidney, Australia, 1991), pp. 192–197.

[21] Shannon, C. E. Programming a computer for playing chess. *Philosophical Magazine 41* (1950), 256–275.

[22] Slate, D. J., and Atkin, L. R. Chess 4.5 — The Northwestern University chess program. In *Chess Skill in Man and Machine*, P. W. Frey, Ed. Springer-Verlag, 1977, pp. 82–118.

[23] Turing, A. M. Digital computers applied to games. In *Faster than thought*, B. V. Bowden, Ed. Pitman, London, 1953, pp. 288–295.

[24] von Neumann, J. Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen 100* (1928), 295–320.

[25] Zobrist, A. L. A Hashing Method with Applications for Game Playing. *ICCA Journal 13*, 2 (1990), 69–73.

[26] rec.games.chess.misc FAQ. `http://www.faqs.org/faqs/games/chess/part4/`, 2002.

[27] Laws Of Chess. `http://www.fide.com/official/handbook.asp?level=EE1`, 1997.