Schach-Computer
Algorithmen und Architekturen

Hochschule Bremen
Fach: RST-Labor
Wintersemester 06/07

Dozent: Prof. Dr. Thomas Risse

Torben Reimers (125458) Semester: I7I
Tobias Tietjen (126688) Semester: I7I
Christian Wehr (126340) Semester: I7I

1. Inhaltsverzeichnis

Table of Contents

2. Abbildungsverzeichnis

3. Geschichte der Schachcomputer

Für die Abschnitte „Geschichte der Schachcomputer“, „Der Schachtürke“ und „Moderne Schachrechner“ wurden hauptsächlich die Quellen [15] und [16] herangezogen.

Die Ursprünge des Schachs werden in Indien sowie in China vermutet. So genau können es die Historiker nicht sagen. In Indien wird vermutet, dass Schach dem Spiel Chaturanga entstammt, welches vier Figuren beinhaltet. Chaturanga hatte die Figuren Streitwagen (Turm), Pferd (Springer), Elefant (Läufer) und den Soldaten (Bauer), welche die Armee der damaligen Zeit widerspiegelten. Die Spielzüge wurden mittels Würfel bestimmt.

Auf der chinesischen Seite wird das Spiel Xiangqi genannt, welches auch noch heute sehr beliebt ist und regelmäßig gespielt wird. Die hier verwendeten Spielsteine sind der Feldherr (König), die Leibwächter (Dame), die Elefanten / Minister (Läufer), die Pferde (Springer), die Wagen (Türme), die Kanonen und die Soldaten (Bauern). Das Schachspiel, in etwas anderer Form als das, was wir in der Neuzeit spielen, kam um die erste Jahrtausendwende zu uns und hielt im 13. Jahrhundert Einzug in die Herrschaftshäuser. Gegen Ende des 15. Jahrhunderts wurde dann schließlich das Schachspiel, so wir es kennen, geboren.

3.1 Der Schachtürke

Mit dem Begriff „Der Schachtürke“ wird der Schachrechner des ungarischen Mechanikers Wolfgang von Kempelen bezeichnet, der im Jahre 1770 gebaut wurde. Dieser Schachrechner bestand aus einem Kasten, an den eine mechanische Puppe montiert war. Dieser Apparat sollte nun die Menschen der damaligen Zeit im Schach besiegen oder zumindestens selbstständig gegen sie spielen. Natürlich konnte so eine Maschine nicht von selbst spielen, aber für den Zuschauer entstand der Eindruck. Im Kasten lag damals ein russischer Schachgroßmeister, der mit Seilen, die an der Puppe angebracht waren, die Figuren auf dem Spielbrett bewegte. Der Schwindel flog auf, als Friedrich der Große, nach seiner Niederlage, dem Erbauer eine große Menge an Geld gab, um das Geheimnis zu erfahren.

Abbildung 1: Der Schachtürke (Quelle: http://www-i1.informatik.rwth-aachen.de)

Bekannte Gegner des Schachtürken waren unter anderem Kaiser Joseph, Großfürst Paul von Russland sowie Napoléon Bonaparte. In der Partie gegen Napoléon benutzt dieser eine neue Eröffnungsstrategie, welche später unter dem Namen Napoléons Angriff oder Napoléons Eröffnung berühmt wurde (er verlor dennoch das Spiel).

3.2 Moderne Schachrechner

Der erste Rechner, der wirklich von selbst spielen konnte, wurde 1890 gebaut. Dieser konnte das Endspiel von König und Turm gegen König spielen. Danach passierte in dieser Hinsicht erstmal nichts, bis schließlich 1956 (dank der Einführung der Digitalcomputer) die Entwicklung rasant weiter ging. 1956 besiegte ein Rechner zum ersten Mal einen Menschen im Schach. Das hierfür verwendete Spielfeld bestand aus einem 6x6 Feld und der Gegner hatte erst eine Woche zuvor angefangen Schach zu spielen.

1979 wurde dann die Rechenmaschine Belle vorgestellt, welche bis zu neun Halbzuge im voraus berechnen konnte. Belle beherrschte die Szene bis 1983 und wurde von Cray Blitz abgelöst. In der heutigen Zeit werden die Schachsysteme im Wesentlichen nur noch auf Softwarebasis, ohne spezielle Hardware, realisiert.

4. Die Elo-Zahl

Die Quellen für dieses Kapitel sind hauptsächlich [2] und [7]. Mit der Elo-Zahl, bzw. dem Elo-System kann man die Spielstärke von Go- und Schachspielern objektiv und näherungsweise bewerten. Es wurde in den sechziger Jahren von Arpad Elo, einem aus Ungarn stammenden Professor für Theoretische Physik, entwickelt. Dieses System ist heute international gebräuchlich und beruht auf Methoden der Statistik und der Wahrscheinlichkeitstheorie. Die Elo-Zahl wird durch den Vergleich von Schachspielern und der Auswertung der Spielpartien berechnet. Grundlage ist die Annahme, dass die Verteilung der Spielstärke in der Gesamtheit der Spieler der Normalverteilung entspricht. Umso mehr Spiele für die Berechnung der Elo-Zahl herangezogen werden, umso genauer konvergiert die Elo-Zahl gegen die tatsächliche Spielstärke. Es lässt sich beim Vergleich der Elo-Zahlen zweier Spieler dann voraussagen, mit welcher Wahrscheinlichkeit ein Spieler gewinnen oder verlieren wird. Folgende Tabelle zeigt die Elo-Zahlen der unterschiedlichen Klassen, in denen menschliche Spieler eingeordnet werden. Es gibt auch eine Rangliste für PC-Schachprogramme (SSDF [1]). Da die Werte allerdings nur aus Computerpartien ermittelt wurden, kann man sie nicht einfach mit den Werten der Rangliste menschlicher Spieler vergleichen

Kategorie / Person	**Elo–Zahl**
Super-Großmeister	>2700
Großmeister	2500-2699
Nationaler Meister	2200-2299
Amateure - Klasse A-D	1200-1999
Anfänger	<1000

Tabelle 1: Die unterschiedlichen Klassen mit den Elo-Zahlen

4.1 Berechnung der Elo-Zahl

Die Formeln wurden aus der Quelle [2] bezogen. Dort findet sich auch eine Tabelle, aus der man mit der Differenz der beiden Elo-Zahlen den so genannten Erwartungswert ermitteln kann. Eine Formel zur Berechnung der Erwartungswerte ist ebenfalls vorhanden. Der Erwartungswert berechnet sich aus den bisherigen Elo-Zahlen der beiden Spieler.

"Der Erwartungswert [...] ist jener Wert, der sich [...] bei oftmaligem Wiederholen des zugrunde liegenden Experiments als Mittelwert der Ergebnisse ergibt. [...] Er ist vergleichbar mit dem empirischen arithmetischen Mittel einer Häufigkeitsverteilung [...]." [21]

Es wird davon ausgegangen, dass die beiden Spieler bereits Elo-Zahlen haben. Für Spieler ohne Elo-Zahlen muss erst einmal eine Elo-Zahl ermittelt werden, worauf wir nicht weiter eingehen.

Zur Berechnung werden folgende Daten benötigt:

Elo_Spieler	zuletzt veröffentlichte Elo-Zahl
Elo_Gegner	zuletzt veröffentlichte Elo-Zahl
Resultat	1 für gewonnen, 0,5 für Remis und 0 für verloren

<u>Berechnung des Einfluss auf die Elo-Zahl:</u>

Faktor = $(3400 – Elo\_Spieler)^2 / 100000$
Differenz = Elo_Spieler – Elo_Gegner
Erwartung = laut Tabelle von [2] oder Formel von [2]
Änderung = (Resultat - Erwartung) * Faktor

Zu beachten ist, dass der Faktor konstant bei 10 bleibt, falls die Spieler eine höhere Elo-Zahl als 2400 haben und dass es ab einer gewissen Anzahl von Spielpartien eine so genannte Faktorreduktion gibt. Bei einer "kritischen Partienanzahl" wird der Faktor für alle Partien reduziert. Das soll eine ungerechtfertigt starke Änderung der Elo-Zahl bei Spielern mit hoher Partienanzahl verhindern.

<u>Faktorreduktion Berechnung:</u>

kritische_anzahl_partien = 850 / Faktor

Wird diese kritische Partienanzahl erreicht, berechnet sich der Faktor folgendermaßen:

Faktor = 850 / anzahl_gespielter_partien

Beispiel für die Änderungen der Elo-Zahlen:

Elo_Spieler = 1830
Elo_Gegner = 1980
Resultat = 0,5 (Remis)

Faktor = (3400 - 1830)² / 100000 = 24,6
Differenz = 1830 – 1980 = -150
Erwartung laut Tabelle = 0,30
Änderung = (0,5 – 0,30) * 24,6 = +5

Elo_Spieler beträgt nach dem Remis 1835.

5. Schachprogramme und ihre Funktionsweise

Das gesamte Kapitel 5 bezieht sich auf die Quellen [3], [11] und [15]. Es konnte auch auf früheres Wissen zurückgegriffen werden, da wir die Minimax-Suche in einem früheren Projekt in Java für das Spiel „Vier gewinnt“ implementiert hatten.

Die grundlegenden Komponenten eines Schachprogramms sind Zuggenerator und Bewertungsfunktion. Mit Hilfe des Zuggenerators wird der Spielbaum generiert. Die Bewertungsfunktion bewertet die Spielstellungen an den Blättern des Baumes. Durch die Minimax-Suche (siehe unten) wird dann der beste Zug ermittelt.

Das hier vorgestellte Basisdesign stammt von Claude Shannon. Er hat in den 70er Jahren ein Programm namens „Chess 4.5“ herausgebracht, welches als erstes den hier vorgestellten Ansatz implementiert hatte.

5.1 Zuggenerator

Der Zuggenerator ist erstmal eine einfache Funktion, die ein Spielbrett übergeben bekommt und die dann alle legalen Züge für eine bestimmte Farbe ermittelt und diese möglichen Figurbewegungen - die potentiellen Züge - ausgibt. Indem sich der Zuggenerator selbst rekursiv aufruft, kann er einen Spielbaum bis zu einer bestimmten Tiefe erzeugen. Das rekursive Aufrufen geschieht wechselnd. Wurde der Zuggenerator für weiß aufgerufen, ruft er sich in der nächsten Instanz für schwarz auf und umgekehrt. Die Tiefe des Spielbaums wird mit der Anzahl der "Halbzüge" (plies) angegeben. Wenn das Schachprogramm seine eigenen möglichen Züge mit den möglichen Zügen des Gegners durchrechnet, ergibt das zwei Halbzüge. Abbildung 1 zeigt den rekursiven Aufruf des Zuggenerators für zwei Halbzüge. Die Graphik ist vereinfacht. Jeder Spieler hat nur zwei Möglichkeiten. Im Schach gibt es meistens weit mehr Zugoptionen. Die Ausgangssituation im Schach bringt 20 Zugmöglichkeiten. Bei 2 Halbzügen ergibt das schon 400 Möglichkeiten. Die Anzahl der möglichen Halbzüge steigt mit der Suchtiefe exponentiell an ("kombinatorische Explosion"). An den Blättern des Spielbaums wird das resultierende Spielbrett der Bewertungsfunktion übergeben. Die Bewertungsfunktion gibt an, ob die Stellung (das Spielbrett) für eine bestimmte Partei "gut" oder "schlecht" ist. Die Bewertungsfunktion gibt für eine Spielsituation einen Wert aus. Ist er positiv, ist weiß im Vorteil, ist er negativ, ist schwarz im Vorteil. Mit den Werten der Bewertungsfunktion kann man mit der so genannten Minimax-Suche einen Zug berechnen.

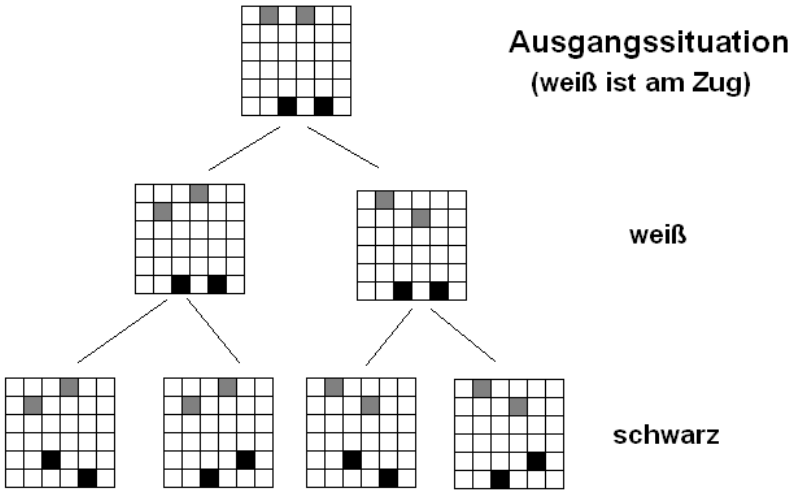

Abbildung 2: Rekursiver Aufruf des Zuggenerators [22]

5.2 Minimax-Suche

Die Minimax-Suche beruht auf dem Prinzip, dass jeder Spieler seinen „bestmöglichen“ Zug spielen würde. Für den weißen Spieler bedeutet „bester Zug“, dass von den zur Option stehenden Zügen, der Zug ausgewählt wird, der maximal positiv ist. Für den schwarzen Spieler bedeutet es, dass der maximal negative Zug gewählt wird (siehe Bewertungsfunktion). Abbildung 3 zeigt die Minimax-Suche. Es werden hier wieder zwei Halbzüge des Spielbaums verarbeitet. Weiß ist am Zug und hat am Anfang die Auswahl unter drei möglichen Zügen. Rekursiv wird dann der Spielbaum bis zur Suchtiefe zwei durchgearbeitet. An den Blattknoten werden die resultierenden Stellungen bewertet. Es wird jetzt immer abwechselnd pro Rekursionstiefe minimiert oder maximiert. Das bedeutet, dass die Elternknoten den niedrigsten bzw. höchsten Wert der Kindknoten übernehmen. In der Abbildung werden die Blattknoten in der untersten Ebene minimiert und in der nächsthöheren Ebene maximiert. Somit wurde der bestmögliche Spielzug von den drei zur Auswahl stehenden Zügen für den weißen Spieler ermittelt. Es ist der mittlere Spielzug, den das Programm dann auch spielen würde. Hätte man einen Zug für den schwarzen Spieler berechnen wollen, wäre das gleiche Prinzip zur Anwendung gekommen, mit dem Unterschied, dass am Anfang (oberste Ebene) minimiert statt maximiert werden würde.

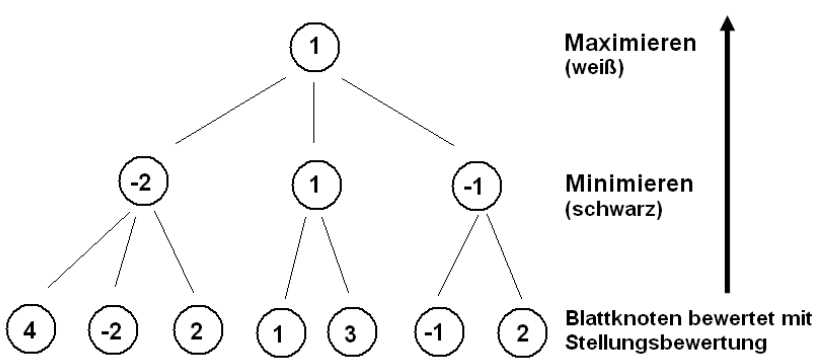

Abbildung 3: Minimax-Suche [22]

5.3 Bewertungsfunktion

Wie oben erwähnt ist die Bewertungsfunktion eine Funktion, die ein Spielbrett (mit einer beliebigen Spielsituation) übergeben bekommt und eine Bewertungszahl zurückliefert.

Die Grundidee der Bewertungsfunktion ist die, dass sie einen von drei Werten zurückgibt. „1“ für „Weiß gewinnt“, „-1“ für „Schwarz gewinnt“ und „0“ für unentschieden. Das setzt aber voraus, dass die Bewertungsfunktion nur Stellungen am Ende des Spielbaums bewertet, dass sie also nur Situationen zur Bewertung bekommt, an denen das Spiel zu Ende ist. Dies lässt sich zwar in endlicher Zeit berechnen, aber die Zeit die man dafür bräuchte ist astronomisch groß. Die Bewertungsfunktion muss also auch Stellungen bewerten können, bei denen das Spiel noch nicht zu Ende ist.

Das kann man nur mit Heuristiken[1] erreichen. So wird versucht zu ermitteln, mit welcher ungefähren Wahrscheinlichkeit die Stellung für einen Spieler besser oder schlechter ist, bzw. mit welcher Wahrscheinlichkeit ein Spieler gewinnt.

Die einfachste heuristische Bewertungsfunktion für Schachprogramme besteht im Aufsummieren der Figuren, die sich auf dem Spielfeld befinden, also die reinen Materialwerte. Die Figuren haben unterschiedliche Werte. Die Dame hat natürlich einen höheren Wert als ein Bauer. Die schwarzen Figuren erhalten negative Werte, die weißen Figuren erhalten positive Werte. Wenn schwarz und weiß die gleichen Figuren in der gleichen Anzahl haben, ergibt die Summe null. Nach der Bewertungsfunktion bedeutet das, dass diese Spielsituation ausgewogen, also unentschieden ist. Die Bewertungsfunktion wird einen positiven Wert übergeben, wenn weiß im materiellen Vorteil ist, wobei die Zahl aussagt, wie stark das der Fall ist. Negative Werte bedeuten einen Vorteil für schwarz. Die Bewertungsfunktion erkennt natürlich auch, ob ein König im Schach steht, wenn das Spiel unentschieden ist oder ein König matt ist. Für das Schachsetzen gibt es einen Bonus bzw. eine Strafe. Für das Mattsetzen gibt es einen sehr hohen Bonus bzw. eine sehr hohe Strafe. Der Bonus bzw. die Strafe sollte so hoch sein, dass der Materialwert dagegen unbedeutend ist, weil Mattzüge natürlich in jedem Fall gesetzt werden sollen (aus der Sicht des Mattsetzers), bzw. in jedem Fall verhindert werden sollen (aus der Sicht des Mattgesetzten).

Man erkennt gleich, wie einfach dieses Verfahren ist, weil die Positionen auf dem Spielfeld und der Figuren zueinander unberücksichtigt bleiben. Hier setzen dann komplexere Bewertungsfunktionen an, die eine Vielzahl von Heuristiken enthalten, wann eine Situation für eine Partei von Vorteil ist. Diese Situationen gehen mit "Bonuspunkten" in die Bewertung mit ein. Die Programmierer der Schachprogramme lassen sich dabei unter anderem von Schachgroßmeistern beraten. Trotzdem könnte ein Schachprogramm mit der beschriebenen einfachen Stellungsbewertung, vorausgesetzt man hätte die entsprechende Rechenleistung, ein perfektes Spiel

[1] "Heuristik: Algorithmen zur Lösung komplexer Probleme verbessert man häufig durch Strategien, die oft auf Hypothesen und Vermutungen aufbauen und die mit höherer Wahrscheinlichkeit (jedoch ohne Garantie) das Auffinden einer Lösung beschleunigen sollen. Solche Strategien heißen Heuristiken. Faustregeln, bereits früher beobachtete Eigenschaften von Lösungen oder die Nachbildung des menschlichen Problemlösungsprozesses sind typische Heuristiken." Definition aus dem Informatikduden

spielen. Zum Durchrechnen des gesamten Spielbaums ist die Komplexität des Spiels aber zu hoch, zumindest in der Anfangsphase des Spiels.

5.4 Erweiterungen und Optimierungen

Es gibt viele Möglichkeiten das oben beschriebene Konzept zu optimieren und die Spielstärke zu verbessern.

5.4.1 Alpha-beta-Suche und Vorsortierung der Züge

Informelle Beschreibung

Mit der Alpha-Beta-Suche muss der Spielbaum unter bestimmten Bedingungen nicht komplett durcharbeiten werden, was zu einer erheblichen Geschwindigkeitssteigerung führt, ohne dass ein schlechterer Zug gespielt wird. Es können so ganze Äste „abgeschnitten“ werden, die nicht weiter verfolgt werden müssen, weil durch die Minimax-Suche sowieso nur noch schlechtere Züge ermittelt werden würden, als die schon berechneten. Wegen des „Abschneidens“ von Ästen wird die Alpha-Beta-Suche auch Alpha-Beta-Cutoff genannt.

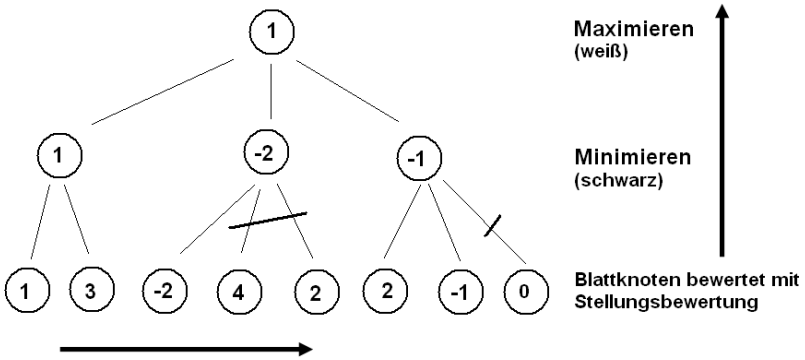

Abbildung 4: Alpha-Beta-Suche [22]

Abbildung 4 zeigt das Prinzip. Der Spielbaum wird von links nach rechts bearbeitet. Der erste Knoten muss erst einmal komplett berechnet werden. Er lieferte den Wert 1. Dieser Wert muss überboten werden. Im zweiten Knoten liefert die Stellungsbewertung den Wert -2. In dieser unteren Ebene wird aber minimiert. Das bedeutet, dass von den nächsten Werten nur noch schlechtere genommen werden. Der Wert 1 aus dem ersten Knoten kann nicht mehr übertroffen werden, also können die restlichen Berechnungen übersprungen werden und es kann zum nächsten Knoten (auf der Minimieren-Ebene) gewechselt werden. Im dritten Knoten wird als erstes der Wert 2 ermittelt. Dieser Wert ist größer als 1, also muss weiter gerechnet werden. Der nächste Wert ist -1. Der Wert ist kleiner als 1, also kann die Berechnung wieder beendet werden.

Beschreibung wie Alpha-Beta-Suche implementiert wird

Die Frage ist nun, was es mit den Alpha- und Betawerten auf sich hat. In der Implementierung wird die normale Minimax-Suche mit zwei Werten, Alpha und Beta erweitert. Diese Werte beschreiben das so genannte Alpha-Beta-Fenster. Mit Hilfe dieser Werte wird entschieden, ob der Suchvorgang abgebrochen werden kann oder

nicht. Die Werte werden am Anfang mit –unendlich für Alpha und unendlich für Beta initialisiert. Für unendlich wird natürlich MAX-Integer bzw. MIN-Integer verwendet. Die Werte werden sozusagen an die Kindknoten vererbt. Die Werte werden horizontal im Spielbaum angepasst, allerdings gilt das nur für nebenliegende Knoten, die den gleichen Elternknoten haben. In einer Minimierstufe wird der Alpha-Wert des nebenliegenden Knotens eventuell nach oben angepasst, wenn der berechnete Wert (von den Kindknoten oder den Blattknoten) größer als der aktuelle Alpha-Wert ist. In einer Maximierstufe wird der Beta-Wert des nebenliegenden Knotens nach unten angepasst, wenn der berechnete Wert kleiner als der aktuelle Beta-Wert ist. In Abbildung 5 sind zu jeden Knoten drei Werte angegeben. Links der Alpha-Wert, in der Mitte der Knotenwert und rechts der Beta-Wert.

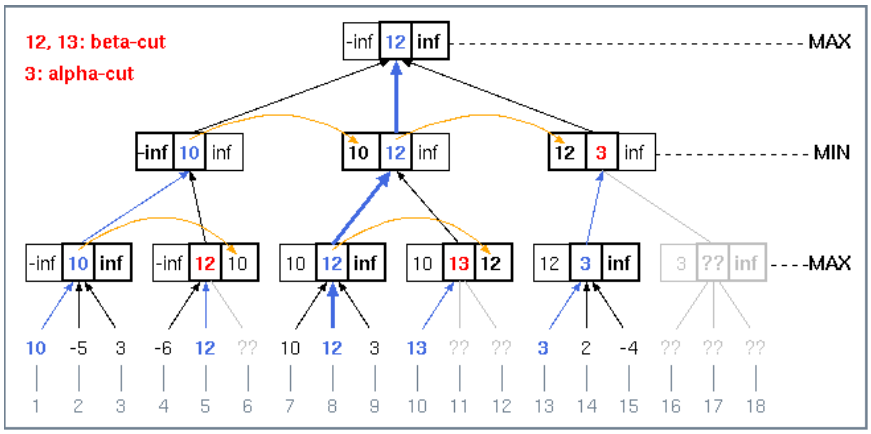

Abbildung 5: Implementierung der Alpha-Beta-Suche [3]

Der Erfolg dieser Optimierung hängt stark davon ab, in welcher Reihenfolge die Züge berechnet werden. Werden zuerst „schwächere“ Züge berechnet und am Ende der „beste“ Zug, können keine oder nur wenige Äste abgeschnitten werden. Deshalb werden die Züge in den Schachprogrammen durch ein schnelles Verfahren vorsortiert, was die offensichtlich unsinnigen Zugmöglichkeiten an das Ende der Liste setzt. Die Züge werden z.B. erst mit der Suchtiefe 2 berechnet. Die Ergebnisse werden dann benutzt um die Züge für die nächst höhere Suchtiefe vorzusortieren. Das Ganze wird auch als „Iterative Deepening“ oder Breitensuche bezeichnet. Der Vorteil ist, dass man bei zeitkritischen Spielen immer schon komplett berechnete Züge hat, die dann nach und nach tiefer analysiert werden. Das ist besser als nur ein paar tief analysierte Züge zu haben, während die restlichen Züge noch gar nicht bearbeitet wurden. Letztendlich entscheidet aber die tiefste Berechnung, welche Züge gut oder schlecht sind. Es kann also sein, dass ein Zug, der von der Vorsortierung als schlecht interpretiert wird, von der tiefer gehenden Suche, als sehr gut eingestuft wird. Dieser Zug ist dann in kleiner Instanz zwar von Nachteil, aber in der tieferen Berechnung erweist er sich als vorteilhaft. Das könnte beispielsweise ein Bauernopfer sein. Menschliche Spieler führen intuitiv eine Art Alpha-Beta-Suche durch. Sie verfolgen offensichtlich unsinnige Züge nicht weiter. Das müssen sie auch, denn selbst Großmeister können nur wenige Halbzüge - im Gegensatz zu schnellen Schachcomputern - in die Zukunft blicken.

5.4.2 Beschleunigung des Suchvorgangs mit Alpha-Beta-Suche [3]

Die Beschleunigung, die mit Alpha-Beta-Suche erreicht werden kann, ist sehr hoch. Es wurden folgende Beschleunigungsfaktoren experimentell ermittelt. Sie stammen von der Quelle [3]. Leider sind dort keine weiteren Quellen angegeben, aber diese Beschleunigungswerte können wir bestätigen, da wir in unserer „Vier gewinnt“-Implementierung – beim Vergleich der Minimax-Suche mit und ohne Alpha-Beta-Cutoff - ähnliche Werte erhalten haben.

Beschleunigung von Minimax mit Alpha-Beta-Suche gegenüber einfachen Minimax: *13,65*

Beschleunigung von Minimax mit Alpha-Beta-Suche und einfacher Vorsortierung gegenüber einfachen Minimax: *136,23*

Die Werte stammen aus einer Beispielberechnung einer Schachstellung bei konstanter Suchtiefe von vier Halbzügen.

5.4.3 Pseudocode für die Minimax-Suche mit Alpha-Beta-Cutoff [3]

```
int Max(int tiefe, int alpha, int beta)
 {
     if (tiefe == 0)
         return Bewerten();
     GeneriereMoeglicheZuege();
     while (ZuegeUebrig())
     {
         FuehreNaechstenZugAus();
         wert = Min(tiefe-1, alpha, beta);
         MacheZugRueckgaengig();
         if (wert >= beta)
             return beta;
         if (wert > alpha)
             alpha = wert;
     }
     return alpha;
 }
```

```
int Min(int tiefe, int alpha, int beta)
 {
     if (tiefe == 0)
         return Bewerten();
     GeneriereMoeglicheZuege();
     while (ZuegeUebrig())
     {
         FuehreNaechstenZugAus();
         wert = Max(tiefe-1, alpha, beta);
         MacheZugRueckgaengig();
         if (wert <= alpha)
             return alpha;
         if (wert < beta)
             beta = wert;
     }
     return beta;
 }
```

5.4.4 Eröffnungsbibliotheken

Viele Schachprogramme stützen sich auf Eröffnungsbibliotheken. Sie sind zuvor sehr zeitaufwändig berechnet worden und liefern für den Anfang der Spielpartie sehr gute Züge. Bei den Eröffnungsbibliotheken wird auch auf das Wissen von Großmeistern zurückgegriffen (zum Beispiel von aufgezeichneten Spielpartien). Die Eröffnungsbibliotheken liefern bis zu einer bestimmten Spieltiefe die Züge und danach beginnt erst die Berechnung. Allerdings gibt es auch Endspielbücher, die von Endspielsituationen ausgehend, durchgerechnete Spiele enthalten.

5.4.5 Hash-Tables

Es werden sogenannte Hash-Tables eingesetzt, um der Situation aus dem Weg zu gehen, zwei identische Spielbretter zweimal bewerten zu müssen. Aus einer Spielsituationen wird mit einer Hash-Funktion ein Index berechnet. An dieser Stelle (z.B. in einem Array) wird die Bewertung gespeichert. Stößt die Berechnung wieder auf dieselbe Spielsituation, kann auf die gespeicherte Bewertung mit der Komplexität O(1) zurückgegriffen werden, ohne die zeitaufwändige Stellungsbewertung noch einmal ausführen zu müssen. Das bringt wieder einen Geschwindigkeitsvorteil, der zum tieferen Durchrechnen des Spielbaums angewendet werden kann. Hash-Tables werden in diesem Zusammenhang hauptsächlich wegen der schnellen Zugriffsgeschwindigkeit auf die Werte eingesetzt. Es kann aber auch zu Kollisionen kommen und die Hashtabelle müsste mehrere Werte an einer Stelle aufnehmen. Dafür gibt es dann spezielle Kollisionsauflösungsstrategien.

5.4.6 Ruhesuche

Ruhesuche heißt, dass nicht nur bis zur vorgegebenen Suchtiefe gerechnet wird. Der Spielbaum wird dabei unter bestimmten Umständen von einem Blatt aus weiter verfolgt. Dies geschieht zum Beispiel, wenn eine Figur gerade geschlagen wurde. Das kann eventuell einen Vorteil oder einen Nachteil bedeuten, der nicht gerechtfertigt ist, weil in der nächsten Runde ein „Gegenschlag“ die Situation wieder ausgleichen könnte.

Zum Beispiel schlägt die Dame in einem Blatt des Spielbaums einen Läufer. Das bringt erst einmal einen Vorteil. Doch in der nächsten Runde könnte diese Dame durch einen Bauern geschlagen werden, was sich dann in einen gravierenden Nachteil verwandelt würde. Deshalb wird die Berechnung an dieser Stelle soweit vertieft, bis sich die Spielsituation „beruhigt“ hat und es zu keinem Schlagabtausch mehr kommt. Der Zuggenerator kann dabei auch unterstützen, indem er die Schlagzüge mit einer Zusatzinformation über den Zug markiert und die Bewertungsfunktion dies nicht selbst „herausfinden“ muss.

Generell wird das Spiel auch in verschiedene Phasen eingeteilt. Im Endspiel befinden sich weniger Figuren auf dem Spielfeld und die Berechnung ist weniger komplex. Deshalb kann in gleicher Zeit tiefer gerechnet werden, was dann auch gemacht wird. Allgemein wird die Suchtiefe dynamisch variiert, um sich den verschiedenen Zuständen des Spiels anzupassen.

5.5 Zusammenfassung

Das interessante an diesem Algorithmus ist, dass ein Schachprogramm nicht viel mehr, als die Spielregeln können muss, um gut bis perfekt spielen zu können. Mit perfekt ist dabei gemeint, dass das Spiel bis zum Spielende durchgerechnet wird und immer der „beste“ Zug gespielt wird. Was der "beste" Zug ist, entscheidet letztendlich die Bewertungsfunktion. Somit kann ein Schachprogramm – mit entsprechender Rechenkapazität - besser spielen als seine Programmierer. Der Titel einer Präsentation von Feng-hsiung Hsu, einem der Hauptentwickler von Deep Blue, lautet: „Designing a Single Chip Chess Grandmaster While Knowing Nothing About Chess (Well, I did know close to nothing about chess...)“.

Dieses Prinzip mit Zuggenerator, Bewertungsfunktion und Minimax-Suche lässt sich prinzipiell auf alle rundenbasierten Nullsummen-Spiele[2] mit perfekter Information anwenden wie z.B. Vier Gewinnt, Tic Tac Toe, Dame, Go, Halma, Mühle usw. Im Grunde muss nur der Zuggenerator und die Bewertungsfunktion an das jeweilige Spiel angepasst werden.

6. Optimierungen der Programme für vorhandene Rechnerstrukturen

Dieses Kapitel gibt einen kurzen Überblick, wie die Schachprogramme die Gegebenheiten der modernen Computersysteme ausnutzen können. Gemeint sind damit hauptsächlich Multi-Core Systeme und 64-Bit-Prozessoren.

Das gesamte Kapitel 6 bezieht sich auf die Dokumentation [4] des Open Source Schachprogramms Crafty von Prof. Robert Hyatt. Crafty ist der Nachfolger von Cray Blitz, einem Schachprogramm für die Cray-Supercomputer.

6.1 Parallele Berechnung des Schachbaums

Der Alpha-Beta Algorithmus stellt die Parallelverarbeitung aufgrund von Datenabhängigkeiten vor Probleme. Trotzdem will man nicht auf die Alpha-Beta-Beschleunigung verzichten.

Das Prinzip ist, dass der Suchbaum auf mehrere Prozessoren aufgeteilt wird. Wichtig ist dabei das loadbalancing, also die optimale Auslastung der Prozessoren. Die vorgestellten Verfahren sind für Mehrprozessorsysteme mit Shared Memory definiert.

[2] „Nullsummenspiele oder allgemeiner Spiele mit konstanter Summe beschreiben in der Spieltheorie Situationen, also Spiele im verallgemeinerten Sinne, bei denen dem Gewinn einer Partei der Verlust einer anderen Partei gegenübersteht. […] Spieltheoretisch lassen sich Nullsummenspiele mit vollständiger Information und zwei Gegnern am einfachsten erfassen. Für diese Spiele existiert immer eine berechenbare Gewinnstrategie, wenngleich sie bisweilen so komplex ist, dass sie noch nicht gefunden wurde, wie bei Schach oder Go.“ Auszug aus http://de.wikipedia.org/wiki/Nullsummenspiel

6.1.1 Principal Variation Splitting (PVS)

Dieses Unterkapitel bezieht sich zusätzlich zur Quelle [4] auch auf die Quellen [23] und [24].

Die wichtigste Regel bei PVS ist, dass immer die erste Verzweigung an einem Knoten durchsucht werden muss, bevor die restlichen Verzweigungen parallel durchsucht werden können. Abbildung 6 zeigt wie die Arbeit auf die vorhandenen Prozessoren aufgeteilt wird. In diesem Beispiel sind es zwei Prozessoren, die parallel den Spielbaum durchsuchen sollen, um den besten Spielzug zu ermitteln. Zuerst arbeitet nur ein Prozessor, doch im weiteren Verlauf nimmt der Parallelisierungsgrad schnell zu.

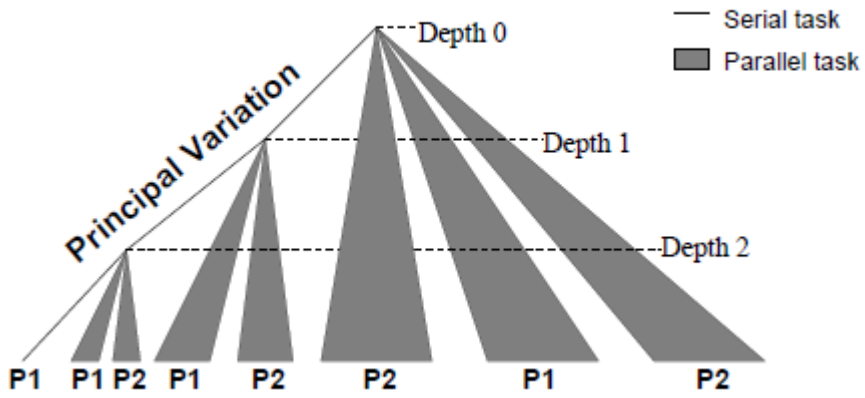

Abbildung 6: PVS mit 2 Prozessoren [24]

Ein wichtiger Aspekt, der in der ersten Version der Ausarbeitung zu kurz gekommen ist, ist der, dass die Prozessoren während des Suchens über die Shared Memory Architektur miteinander kommunizieren. Sie lesen den aktuellen Knotenwert und sorgen dafür, dass dieser Wert gemäß Minimax beim Elternknoten angepasst wird. Das Aktualisieren des Alpha-Beta-Fensters für die Schwesternknoten erfolgt ebenfalls über den Shared Memory.

Suchen mehrere Prozessoren also parallel, aktualisieren sie die Knoten- und Alpha/Beta-Werte fortlaufend. Andererseits lesen sie beim Durchsuchen diese Informationen auch aus, um eventuell einen Cutoff gemäß der Alpha/Beta-Suche durchführen zu können.

Nachteile

Die Anzahl der Prozessoren, die eingesetzt werden können ist abhängig von der Anzahl der Züge. Wenn die Anzahl der möglichen Züge klein oder die Anzahl der Prozessoren groß ist, haben einige Prozessoren nichts zu tun.

Der zweite Nachteil ist, dass die Unterbäume von einer gegebenen Position nicht immer gleich groß sind (zum Beispiel wegen den Sucherweiterungen). Das bedeutet, dass man auf einen Unterbaum eventuell warten muss, während der andere schon durchgerechnet wurde. Dies führt wieder eventuell zum Prozessor-Leerlauf.

Dies sind die Gründe warum Cray Blitz auf der Cray YMP mit 8 Prozessoren kaum besser war als auf der Cray XMP mit 4 Prozessoren (wenn man die höhere Taktrate nicht berücksichtigt).

# processors	1	2	4	8	16
speedup	1.0	1.8	3.0	4.1	4.6

Tabelle 2: Geschwindigkeitssteigerung bei PVS

Die Performance-Ergebnisse zeigen, dass die Geschwindigkeitssteigerung mit diesem Verfahren gegen 5 konvertiert. Egal wie viele Prozessoren eingesetzt werden, man erreicht einfach keine höhere Beschleunigung, weil die Arbeit nicht effizient auf die Prozessoren aufgeteilt werden kann. Der Leerlaufanteil der Prozessoren ist einfach zu hoch und wenn dann mehr Prozessoren einsetzt werden, müssen nur mehr Prozessoren warten. Ein wichtiger Grund dafür wurde oben bei den Nachteilen auch genannt. Die Anzahl der Prozessoren die effizient beschäftigt werden können ist abhängig von der Anzahl der möglichen Spielzüge in einer bestimmten Position. Diese Anzahl ist endlich und nimmt gegen Ende der Partie auch noch ab.

6.1.2 Enhanced Principal Variation Splitting (EPVS)

EPVS funktioniert im Grunde wie PVS, mit dem Unterschied, dass ein Prozessor ein Leerlaufsignal auslöst, wenn er nichts mehr zu tun hat. Wenn ein Prozessor ein Leerlaufsignal ausgelöst hat und m Prozessoren im System sind, gibt es noch mindestens m-1 aktive Prozessoren. EPVS stoppt dann alle Prozessoren (welche am Halbzug P, der aktuellen Position arbeiten) und folgt der ersten verbleibenden Halbzug=P Verzweigung zwei Halbzüge. Dort wird der Baum aufgeteilt mit einer Parallelsuche, die exakt wie bei PVS abläuft. Der Gewinn ist jetzt, dass alle Prozessoren, wenn sie an einer einzelnen Verzweigung arbeiten, an der es wenig Arbeit gibt, zwei Halbzüge im Baum „runterspringen“, zu einem Knoten an dem es mehr Arbeit gibt.

Dies funktioniert nur, weil die Teilergebnisse der Suche die gestoppt wurde, in einer Transposition Table gesichert wurden. Der Grund warum der Baum genau zwei Halbzüge weiterverfolgt wird, liegt in der Alpha-Beta-Suche begründet.

# processors	1	2	4	8	16
speedup	1.0	1.9	3.4	5.4	6.0

Tabelle 3: Geschwindigkeitssteigerung bei EPVS

Doch die Performance dieser Erweiterung ist auch nicht viel höher als die vom PVS. Dies war die Motivation für Dynamic Tree Splitting.

6.1.3 Dynamic Tree Splitting (DTS)

Am Anfang der Suche verhält sich DTS wie PVS. Ein Prozessor geht an der linken Seite des Spielbaums runter von Halbzug 1 bis zum Halbzug n. Bei Halbzug n steigen dann alle Prozessoren mit ein und suchen die Zugliste bei Halbzug n wie bei PVS/EPVS parallel durch. Wird ein Prozessor mit seiner Arbeit fertig setzt DTS ein.

Der Prozessor im Leerlauf meldet im Shared Memory, dass er nichts mehr zu tun hat. Der Prozessor versucht nun einen anderen Prozessor zu unterstützen. Die arbeitenden Prozessoren geben ihren Baumstatus (Tree State) während der Arbeit ständig an, ebenfalls im Shared Memory. Der Prozessor im Leerlauf analysiert diese Daten und entscheidet dann, welchem Prozessor er helfen kann. Der arbeitende Prozessor muss dazu gerade an einem Baum arbeiten, der kompliziert genug ist. Ist ein Prozessor gefunden, informiert der Prozessor im Leerlauf den arbeitenden Prozessor und die Zusammenarbeit kann beginnen.

Der Prozessor im Leerlauf wählt einen Halbzug s und informiert den arbeitenden Prozessor, welcher Halbzug (Split Point Ply) gewählt wurde. Der arbeitende Prozessor, dem der Unterbaum "gehört", kopiert dann den kompletten Baumstatus in ein Gebiet im Shared Memory, das "Split Block" genannt wird. Beide Prozessoren können nun mit Hilfe dieser Daten Züge extrahieren und sie parallel bearbeiten. Wenn ein Prozessor dann wieder nichts zu tun hat, wird dieser Prozess wiederholt. Entweder hilft dieser Prozessor dem Prozessor, mit dem er vorher gearbeitet hat, wieder an einer anderen Stelle oder er hilft einem komplett anderen Prozessor, weil sich eine weitere Zusammenarbeit nicht mehr lohnen würde. Ein Problem kann allerdings auftreten, wenn die Prozessoren genau so lange für die Verwaltung des Splittings brauchen, wie für das eigentliche Suchen.

Im Grunde starten die Prozessoren zwar alle an einem Einzelknoten, aber sie wandern dann schnell in Gruppen auseinander und entkoppeln sich so von den Einzelknoten und rechnen immer überall dort, wo Arbeit ist. Man muss aber aufpassen, dass das Berechnen der Split Points und das Suchen gleich lange dauern.

# processors	1	2	4	8	16
speedup	1.0	2.0	3.7	6.6	11.1

Tabelle 4: Geschwindigkeitssteigerung bei DTS

Die Performance-Ergebnisse zeigen, dass der Beschleunigungsfaktor mit höherer Anzahl der Prozessoren auch steigt und nicht schon bei 16 Prozessoren stagniert.

6.2 Programminterne Representation des Schachbretts

Die Performance von Schachprogrammen hängt auch von der programminternen Repräsentation des Schachbretts ab. Es gibt unterschiedliche Ansätze. Die Beschreibung der unterschiedlichen Ansätze bezieht sich auf die Dokumentation des Crafty-Schachprogramms [4].

6.2.1 Interne Brettdarstellung mit einem zweidimensionales Array

Der intuitivste Ansatz, das Schachbrett darzustellen, ist es, ein zweidimensionales Array zu verwenden, z.B. board[8][8]. Zweidimensionale Arrays werden auch als lineare Liste von Elementen abgebildet. Will man jetzt auf Element (x,y) zugreifen, wird der Index des Elements mit x*8+y berechnet. Dabei hat man bei jedem Zugriff eine Multiplikation und eine Addition auszuführen. Im Idealfall erzeugt der Compiler aus der Multiplikation einen Linksshift um 3 Bit, aber das ist immer noch nicht optimal.

Zweiter Nachteil: Bei jedem Zug werden neue x und y Koordinaten errechnet. Diese müssen allerdings noch auf Legalität überprüft werden, eine Figur darf sich ja auch nicht außerhalb des Spielbretts bewegen. Also muss jeweils für die x und y Koordinate geprüft werden, ob die Werte >= 0 und <= 7 sind. Dies ist ein Nachteil, weil dieser Test sehr oft wiederholt werden muss.

6.2.2 Interne Brettdarstellung mit einem eindimensionalen Array

Eine Optimierung, die bei Ansätzen mit zweidimensionalen Array dann oft angewendet wird, ist es, aus dem zweidimensionalen Array ein eindimensionales Array zu machen, z.B. board[64]. Vorteilhaft ist, dass jetzt die Multiplikation/Addition beim Zugriff auf die Elemente wegfällt. Will man sich beispielsweise diagonal bewegen muss man einfach die Konstante 9 addieren. Dieser Entwurf löst das eben erwähnte Problem, aber es bleibt noch das Problem des Testens, ob sich die berechnete Position noch auf dem Brett, bzw. innerhalb der Array-Grenzen befindet. Man spart dabei also die Multiplikation/Addition, aber man muss den Bedingungstest weiterhin tolerieren. Auf modernen RISC Computern sind die Bedingungstest langsamer, als die Multiplikations- und Additionsbefehle. Ein Bedingungstest ist komplexer als eine einfache Addition. Wenn man z.B. ermitteln will, ob 8 > 3 ist, wird die 8 von der 3 abgezogen (was ja auch eine Addition ist) und anschließend muss noch geprüft werden, ob das Ergebnis negativ ist. Also spart man mit dieser Brettdarstellung nicht besonders viel Rechenzeit und man muss versuchen, die Bedingungstests effizienter zu gestalten.

6.2.3 Interne Brettdarstellung mit einem eindimensionalen Array und Umrahmung

Um die Bedingungstests einfacher zu machen gibt es eine weitere Brettdarstellung. Dabei wird dem Spielbrett eine Außenfläche gegeben, die zwei Felder breit ist. So kommt man auf eine Elementanzahl von 144. Stellt man sich dieses Array zweidimensional mit 12 * 12 Feldern vor, so liegt das Spielbrett in der Mitte. Die Umrahmung muss zwei Felder breit sein, weil der Springer bei seinen Zügen ein Feld überspringen kann. Die „illegalen“ Felder in der Umrahmung werden mit einem speziellen Wert vorbelegt, z.B. 99. So kann man mit einem einzigen Bedingungstest prüfen, ob sich die Spielfigur noch auf dem Brett befindet. Man muss nur prüfen ob Position ungleich 99 ist, anstatt wie beim Entwurf im vorherigen Kapitel prüfen zu müssen, ob jede Koordinate >=0 und <=63.

6.2.4 Interne Brettdarstellung mit Bitboards

Die Bitboards verfolgen ein anderes Konzept. Mit einem 64-Bit Wort lässt sich die Position eines bestimmten Typs von Stein darstellen. Dabei repräsentiert das 0. Bit das Spielfeld A1 und das 63. Bit Spielfeld H8. Für alle sechs Spielsteine (König, Königin, Turm, Läufer, Springer und Bauer) und für jeweils beide Farben gibt es eine Repräsentation in Form eines 64-Bit Wortes.

Diese Form der Brettdarstellung ist nicht sonderlich verbreitet, da sie den Nachteil hat, durch die vielen Bitoperationen relativ unübersichtlich zu sein. Das Programm Crafty verwendet aber solche Bitboards. Der Autor gibt selbst sogar an, daß er mit dieser Art von Darstellung anfänglich (etwas über ein Jahr) Probleme hatte.

Beispiel für "bit-parallele" Operationen

Wenn z.B. überprüft werden soll, ob ein schwarzer Bauer auf dem Feld d5, d6, d7, e5, e7, f5, f6 oder f7 steht, lässt sich dies mit einer einfachen And-Operation überpüfen. Um dies zu bewerkstelligen, muss ein Bitboard erstellt werden, das die entsprechenden Felder auf 1 setzt. Führt man nun eine AND Verknüpfung mit dem Bitboard der schwarzen Bauern aus, kann man das Ergebnis direkt auswerten. Wenn es ungleich null ist, muss auf einem der 8 Felder mindestens ein Bauer stehen. Im Prinzip wurden also mit einer Operation 8 Anfragen parallel bearbeitet. Die Position der Bauern lässt sich bei Bedarf mit Hilfe von Shift-Operationen bestimmen.

Bei einer der Array-Brettdarstellungen müsste man dies mit Hilfe von mehreren Operationen für jeden Stein separat überprüfen.

Fazit

Bitboards bringen nur auf 64-Bit Maschinen wirkliche Geschwindigkeitsvorteile, da die Bitoperationen dort effizient umgesetzt werden können. Führt man Programme, die eine andere Form der Brettdarstellung gewählt haben, auf so einer Maschine aus, lässt sich dieser Vorteil nicht ausnutzen.

7. Deep Blue

Falls nicht anderes angegeben war die Hauptinformationsquelle bei der Recherche über Deep Blue [5]. Deep Blue ist der Nachfolger von Deep Thought. Der Initiator des Deep Thought Projekts, welches im Jahr 1985 an der Carnegie Mellon University startete, war Feng-hsiung Hsu. Das Projekt bestand im wesentlichen aus einem auf einen Chip integrierten Zuggenerator. Hsu wechselte 1989 zu IBM und aus seiner dortigen Forschungsarbeit an Problemen der Parallelrechnung enstand Deep Blue. Die Motivation bei Deep Blue war es, wie auch bei Deep Thought, Großmeister im Turniermodus besiegen zu können. Es gab zwei separate Versionen von Deep Blue. Die erste Version (Deep Blue I) verlor 1996 gegen Garry Kasparov und die zweite Version (Deep Blue II) besiegte ihn im Jahr 1997. Die zweite Version von DeepBlue erreichte eine Elo-Zahl von 2768. Es war das erste mal, dass ein Schachcomputer einen amtierenden Weltmeister unter regulären Zeitkontrollen besiegte.

7.1 Designphilosophie von Deep Blue

Ein Ziel von Deep Blue (II) war das Erreichen einer hohen Geschwindigkeit. Dies sollte mit einem parallelen System und speziellen Schachchips umgesetzt werden. Auf Grund der „kombinatorischen Explosion“ wird für tiefere Suchen eine immer höhere Rechenleistung benötigt. Doch obwohl die Geschwindigkeit eine große Rolle spielt, ist sie tatsächlich eher zweitrangig. Wenn Deep Blue gegen Großmeister bestehen soll, darf das System keine groben Fehler enthalten, wenn überhaupt dann nur Fehler, die sich schwer ausnutzen lassen. Großmeister sind in der Regel in der Lage, solche Fehler konsequent auszunutzen, um den Rechner zu besiegen. Für Schachcomputer im Allgemeinen ist es daher nicht nur wichtig, viele Halbzüge zu berücksichtigen. Die verschiedenen Stellungen müssen vor allem richtig bewertet werden. Für eine schnelle Bewertung spielt auch eine gute Vorsortierung (siehe Alpha-Beta-Suche) eine Rolle. Daher sollte der Suchalgorithmus so optimiert sein, aus der Vielzahl von möglichen Halbzügen möglichst die stärksten für die Bewertung zu berücksichtigen.

7.2 Systemübersicht

Deep Blue wurde als massiv paralleles System für die Suche in Schachbäumen konzipiert.
Das System basiert auf einer IBM RS/6000 SP bestehend aus:

- 30 Knoten
- Knoten kommunizieren über einen High-Speed Switch
- Knoten besteht aus
    - IBM RS/6000
    - 16 Schachchips
    - MicroChannel®bus für die Anbindung der Schachchips
    - 1 GB RAM
    - 4 GB Festplattenkapazität

7.3 Verteilung der Aufgaben im System

Die Suche in den Schachbäumen ist in drei Schichten aufgeteilt. Jede Schicht durchsucht den Spielbaum für eine bestimmte Anzahl von Halbzügen und bewertet die Stellungen mit Hilfe der Schachchips.

Die erste Schicht der Suche wird von der Masterworkstation übernommen. Die ersten Halbzüge können problemlos von einer Workstation bewerten werden, da die Anzahl der zu bewertenden Halbzüge noch vergleichsweise gering ist. Eine Parallelisierung der Suche findet an dieser Stelle noch nicht statt und würde auch kaum Geschwindigkeitsvorteile bringen.

Die zweite Schicht der Suche wird von den verbleibenden 29 Workstations übernommen, welche weitere Halbzüge bewerten.

Die dritte Schicht der Suche wird von den 29 Workstations an die Schachchips delegiert. Dazu gehört auch die Ruhesuche. Die Schachchips übernehmen mit den letzten Halbzügen die meiste Arbeit. Durch den exponentiellen Anstieg an möglichen Schachzügen pro Halbzug werden über 99% der durchsuchten Halbzüge von den Schachchips übernommen.

7.3.1 Die Spielbaumsuche

Die Suche wird zum Teil von der Hardware und zum Teil von der Software übernommen. Die Software, die auf den Workstations ausgeführt wird, wurde dafür in C codiert. Die Software ist zwar flexibel und lässt sich bei Bedarf ändern, jedoch wird dies durch feste Vorgaben für die Form der Suche eingeschränkt. Das liegt daran, dass ein Teil der Suche von der Hardware übernommen wird und sich nicht mehr ändern lässt.

7.3.2 Bewertung der generierten Stellungen

Die Bewertungsfunktion ist in Hardware implementiert. Bei einer Softwarelösung gibt es immer das Problem, dass eine Bewertungsfunktion bei einer steigenden Zahl von Bewertungsaspekten immer mehr Zeit für die Ausführung benötigt. Dieses Problem fällt bei der Hardwarebewertung weg. Es handelt sich dort um eine feste Zeitkonstante. Ein Nachteil ist jedoch, dass es nachträglich nicht mehr möglich ist, die Bewertungsfunktion zu ändern bzw. neue Bewertungsaspekte hinzuzufügen.

7.3.3 Implementierung der parallelen Suche

Wenn die Suche tiefer wird, verteilt sich die Arbeit auf das gesamte System. Da die Schachchips nur direkt mit ihrem Host kommunizieren können, wurde eine länger dauernde Hardwaresuche (über 8000 Spielbaumknoten) vermieden. Eine weitere Aufteilung der Knoten ist während einer Hardwaresuche nicht mehr möglich. Deswegen wurde die Suche zum Teil auch von der Software übernommen. Anders als bei der Hardwaresuche kann die Softwaresuche jeder Zeit unterbrochen werden, um ihre erledigte Arbeit an den Master zurückzuliefern. Die Arbeit lässt sich dann weiter aufteilen bzw. parallelisieren.

Die Workstations kommunizieren nicht direkt untereinander, sondern nur über den Master, was die Implementierung vereinfacht. Der Flaschenhals in Deep Blue ist

deswegen die Master Workstation. Diese garantiert im Übrigen, dass sie immer Arbeit für die Workstations parat hat. Verschiedene Faktoren können das Timing und die Job-Verteilung auf die Prozessoren beeinflussen, was dazu führt, dass die parallele Suche nicht deterministisch ist. Das Debuggen wird dadurch erschwert, da sich die Fehlwirkungen evt. nicht reproduzieren lassen.

Die Entwickler gestehen ein, dass bei Realisierung der parallelen Suche noch eine Menge Verbesserungspotenzial besteht. Die Parallelität wird eher suboptimal genutzt. Das Hauptziel der Entwickler war eher eine ausgefeilte Bewertungsfunktion. Die Geschwindigkeit des Systems ist wie bereits erwähnt ohnehin zweitrangig. Falls die Bewertungsfunktion grobe Fehler enthält, besteht die Gefahr, dass ein Großmeister dies konsequent ausnutzt. In dem Fall würde auch eine höhere Effektivität der Parallelität, was bedeuten würde, dass das System mehr Stellungen pro Sekunde bewerten könnte, den Fehler nicht kompensieren.

7.4 Der Schachchip

Bei dem Schachchip, der bei Deep Blue mehrfach zum Einsatz kam, handelt es sich um einen ASIC. Dabei stellt jeder einzelne Schachchip eine eigenständige Schachmaschine dar.

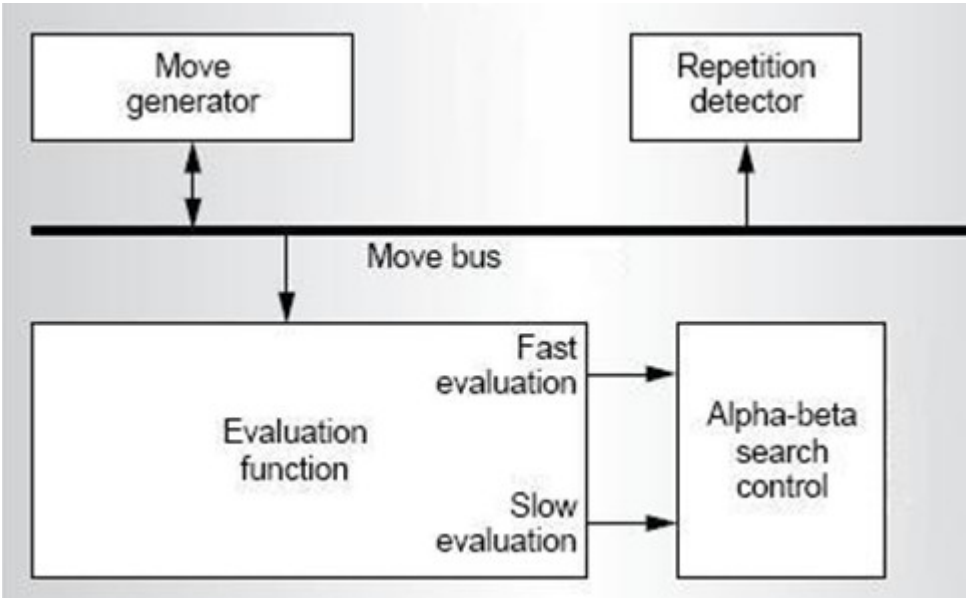

Abbildung 7: Funktionsblöcke des Schachchip [5]

Wie sich in Abbildung 7 erkennen lässt, besteht der Schachchip aus einem Zuggenerator, dem Repetitiondetector (bzw. Smartmove Stack), der Bewertungsfunktion und der Search Control.

Ein großer Vorteil des Schachchips ist, dass sich Algorithmen implementieren lassen, die als reine Softwarelösung nur eine sehr geringe Effektivität hätten. Durch die hohe Parallelität des Schachchips reduziert sich die Ausführungszeit eines solchen Algorithmus z.T. nur noch auf eine Gatterverzögerung. Dies lässt sich auch anhand des Zuggenerators von Deep Blue näher erläutern.

7.4.1 Der Zuggenerator

Der Zuggenerator verfügt über eine 8x8 Logik, die das Schachbrett repräsentiert. Jede einzelne Zelle besteht dabei aus vier Hauptkomponenten:

- Find-Victim-Transmitter
- Find-Attacker-Transmitter
- Receiver
- Arbiter
- Vier-Bit-Register für die Spielsteinrepräsentation

Die ersten drei Bit für die Spielsteine repräsentieren dabei einen von sechs Spielsteinen. Das vierte Bit steht für die Farbe (schwarz oder weiß).
Der Zuggenerator wird durch einen fest verdrahteten Zustandsautomaten kontrolliert. Dabei wird immer nur ein Zug generiert, obwohl implizit immer mehrere Züge gleichzeitig generiert werden. Die bereits generierten Züge werden ausmaskiert. Für die Zuggenerierung ist wichtig, dass möglichst die besten Züge zuerst ausgegeben werden, damit die Alpha-Beta Suche effizient ist. Es folgt ein Beispiel, welches den Zusammenhang näher erläutert.

Beispiel einer Zuggenerierungssequenz – die Schlagzüge

Die Schlagzüge werden von dem Zuggenerator zuerst generiert, da sie potenziell zu den besten Zügen gehören. Beim Generieren eines Schlagzuges soll möglichst der niederwertigste Angreifer das höherwertigste Opfer schlagen. Damit ein Schlagzug generiert werden kann, werden mehrere Zyklen durchlaufen. Dazu gehört der Finde-Opfer Zyklus und der Finde-Angreifer Zyklus. Die folgenden Bilder in diesem Kapitel wurden mit Hilfe von [25] selbst erstellt.

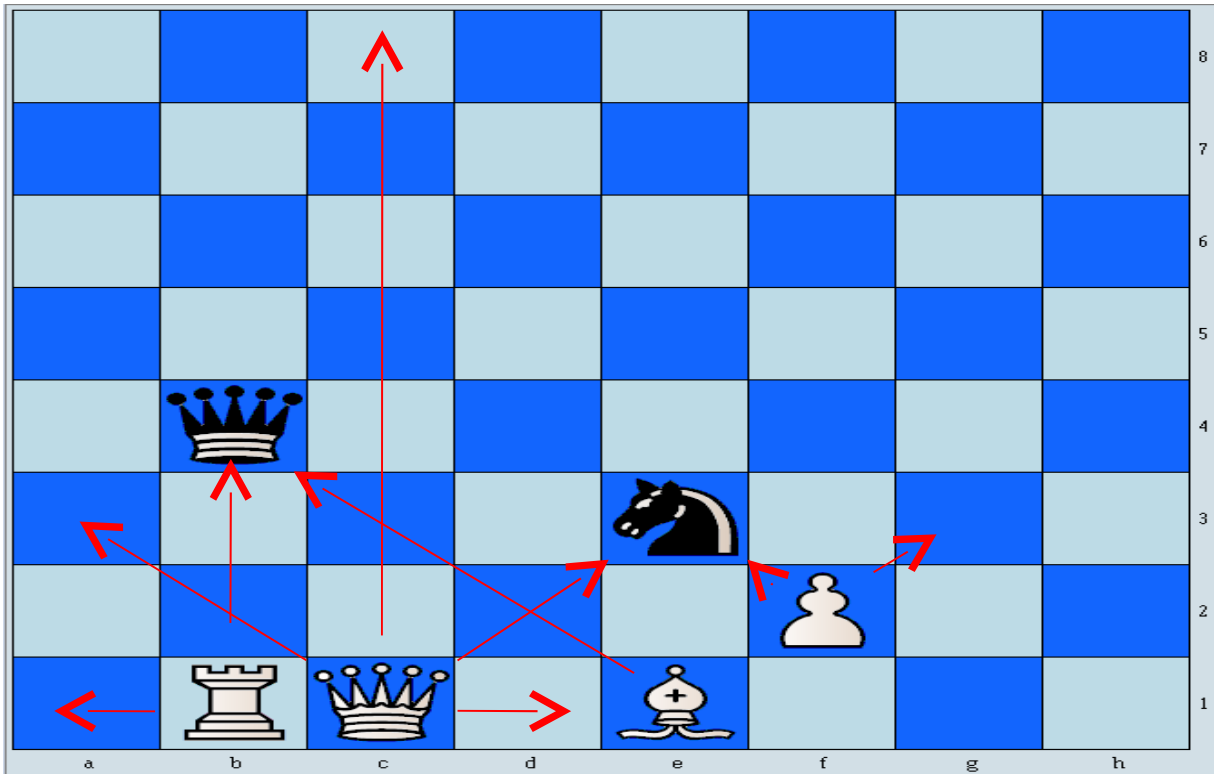

Abbildung 8: Finde-Opfer Zyklus [22]

Im Finde-Opfer Zyklus wird das höchstwertigste Opfer bestimmt. Dazu strahlen alle spielereigenen Spielsteine Angriffssignale aus. Diese werden über den Find-Victim-Transmitter an die Receiver der bedrohten Felder geschickt. In Abbildung 8 sind diese Angriffssignale mit roten Pfeilen gekennzeichnet. In diesem Beispiel werden Dame und Springer bedroht. Ein Signal wird ausgehend von den beiden Feldern an den Arbiter geschickt. Dieser wählt anhand der Priorität das Opfer aus. Die Priorität steigt mit höherwertigen Spielsteinen vom leerem Feld, Bauer, Pferd, Läufer, Turm bis zur Königin. Ein König kann nicht geschlagen werden und wird gesondert behandelt. Nachdem ein Opfer gefunden wurde, folgt der Finde-Angreifer Zyklus.

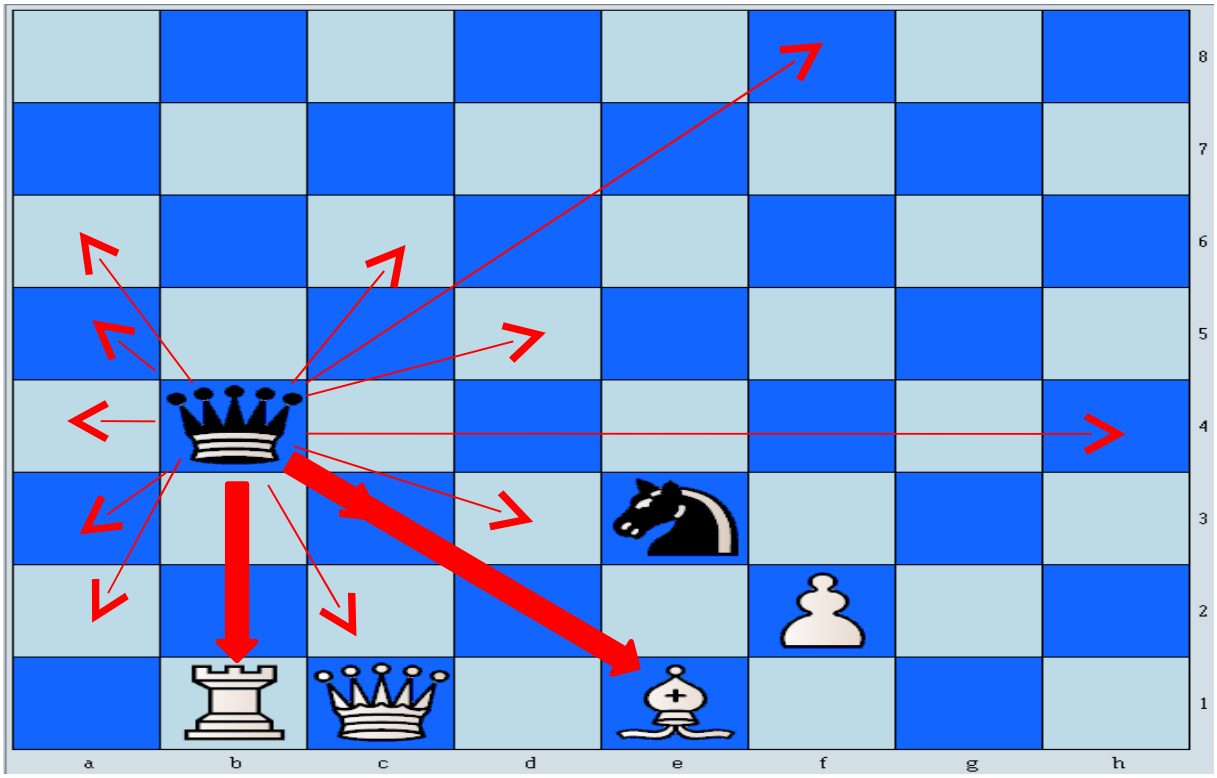

Abbildung 9: Finde-Angreifer Zyklus [22]

Beim Finde-Angreifer Zyklus soll der niederwertigste Angreifer gefunden werden. Dafür sendet das Opfer umgekehrte Signale als Superspielstein aus. Dies ist in Abbildung 9 zu erkennen. Ein Superspielstein kombiniert alle Zugmöglichkeiten der sechs unterschiedlichen Figurtypen. Die Signale verlaufen dabei in entgegengesetzter Richtung des jeweils repräsentierten Spielsteins. Die Angriffssignale werden an die Receiver aller möglichen Angreifer gesendet. Bei Übereinstimmung des möglichen Angreifers (dicke rote Linien) mit dem ansässigen Stein, wird ein Prioritätensignal an den Arbiter gesendet. In dem Beispiel trifft das auf den Turm und auf den Läufer zu. Der niederwertigste Spielstein hat hier die höchste Priorität. Sie fällt vom Bauer, Pferd, Läufer, Turm, Königin bis zum König. In diesen Fall wird der Läufer ausgewählt.

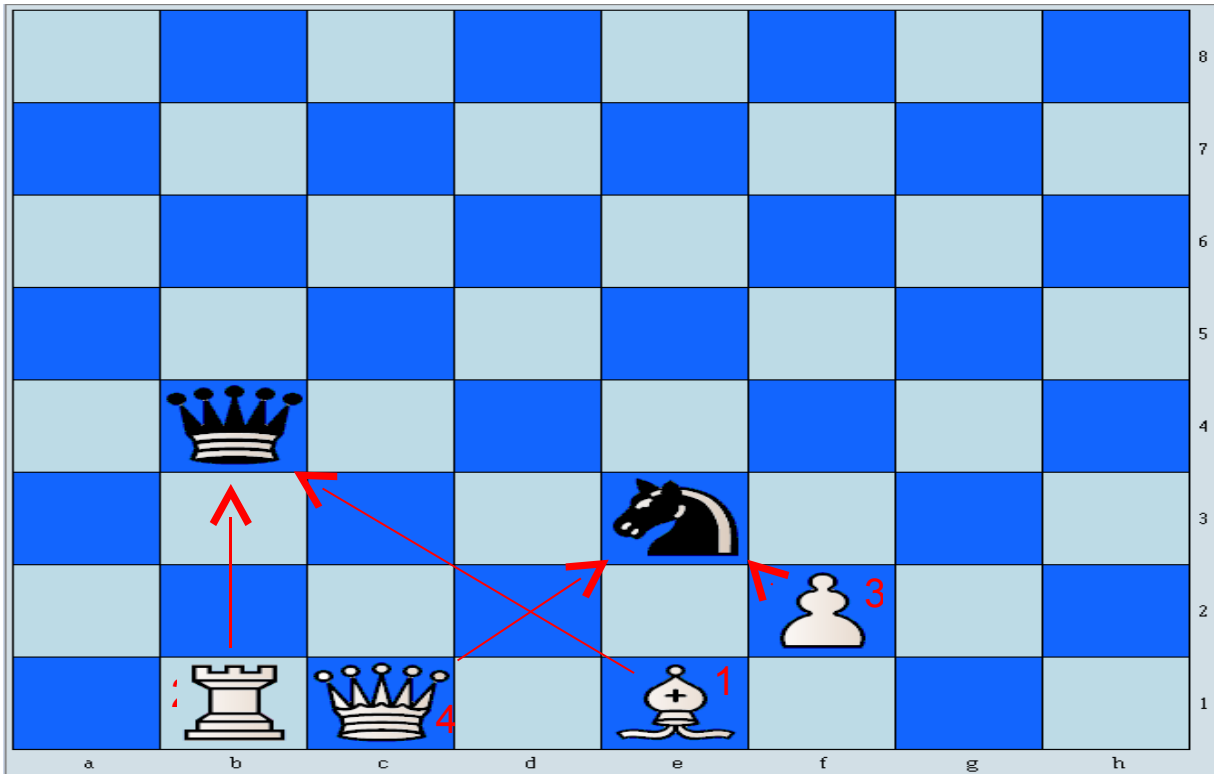

Abbildung 10: Reihenfolge der generierten Züge [22]

In Abbildung 10 ist die Reihenfolge der generierten Züge zu erkennen. Um mehrere Züge zu generieren, werden die bereits ermittelten Züge ausmaskiert. Dies wird so oft wiederholt, bis alle Angriffszüge generiert wurden. Nach den Schlagzügen werden weitere Züge generiert – dazu gehören z. B. Züge die den König Schach setzen, sowie entsprechende Ausweichzüge, die den König von der Schachposition wegbewegen.

Zuggenerator Fazit

Der Zuggenerator im Schachchip nutzt den Vorteil der Hardware aus. Die Generierung eines Zuges kann in konstanter Zeit umgesetzt werden. Die Ausführungszeit eines solchen Algorithmus liegt nur bei wenigen Zyklen. Durch die Parallelität der Hardware ist der Algorithmus effizient. Als reine Softwareimplementierung wäre der Algorithmus hingegen völlig inakzeptabel, da ja für jeden einzelnen Zug implizit alle möglichen Züge mit generiert werden. Der Nachteil ist auch wieder, dass sich der Zuggenerator nachträglich nicht mehr modifizieren lässt. Eine Softwarelösung wäre hier wieder flexibler. Hydra, ein spezieller Schachrechner und inoffizieller Nachfolger von Deep Blue, auf den noch eingegangen wird, setzt FPGA's ein, um trotzdem flexibel zu bleiben.

7.4.2 Evaluation function – Bewertungsfunktion

Für die gesamten Abschnitte der Bewertungsfunktionen, wurde auf den Artikel der Zeitschrift „IEEE micro“ zurückgegriffen, Quelle [5].

Die Bewertungsfunktion ist der Teil des Schachchips, welcher die einzelnen Spielzüge analysiert und bewertet. Je besser diese Analyse dabei ist, desto besser ist nachher auch der daraus resultierende Spielzug. Die Bewertungsfunktion bestand bei Deep Blue aus 66.000 logischen Einheiten, welche die einzelnen Funktionen der Bewertungsfunktion realisieren. Diese logischen Einheiten beinhalten ausschließlich die Funktionen, jedoch nicht den dafür benötigten RAM bzw. ROM. Die nachfolgende Abbildung 11 zeigt, wo die Funktionsblöcke jeweils auf dem Schachchip angeordnet sind. Die gesamte Evaluation function teilt sich in zwei Bereiche auf: 1. die schnelle Bewertung und 2. die langsame Bewertung.

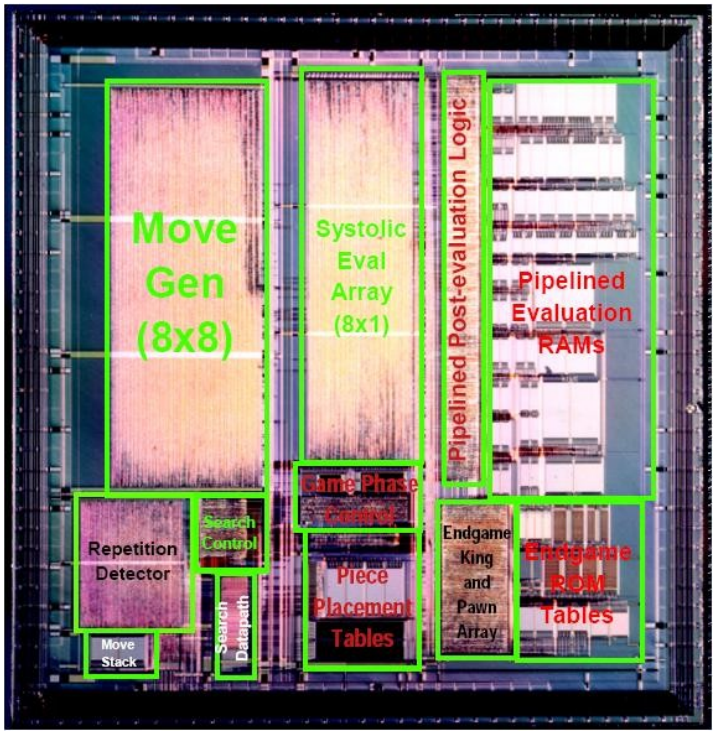

Abbildung 11: Layout des Schachchips [5]

Die schnelle Bewertung (fast) benötigt nur einen Zyklus, um einmal den anliegenden Spielzug zu analysieren und zu bewerten. Dagegen steht die langsame Bewertung (slow), die 11 Zyklen für ein Ergebnis benötigt. Dabei fallen drei Zyklen für das Einlesen und acht Zyklen für die eigentliche Beurteilung des Zuges an.

7.4.3 Aufbau der schnellen Bewertung (fast Evaluation function)

Die schnelle Bewertung beinhaltet nur die wichtigsten Analysen und ist somit um ein vielfaches schneller als die langsame Version. Die zu Verfügung stehenden Blöcke sind: die „Piece Placement Table“, das „Endgame king and pawn array”, die “Endgame logic and ROMs“ und die „Game phase control“.

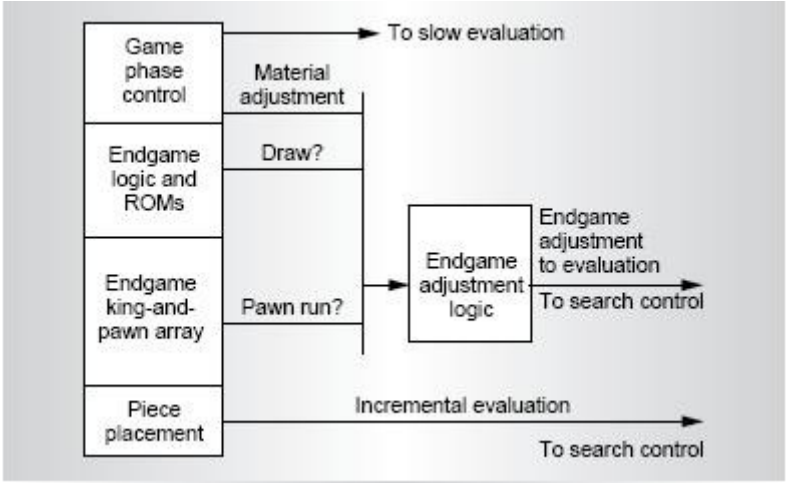

Abbildung 12: Fast Evaluation [5]

Piece Placement Table

Die Piece Placement Table ist für die Bewertung der aktuellen Position der Figur auf dem Spielfeld verantwortlich. Hierbei schaut die Funktion in eine zuvor gefüllte Tabelle, um für das aktuelle Feld den jeweiligen Wert zu erhalten. Die Bewertung der einzelnen Felder wird vorher, meist von einem Schachgroßmeister, übernommen. Im Falle des Deep Blue war es Joel Benjamin.

Endgame king and pawn array

Das „Endgame king and pawn array“ hat zwei Funktionen, erstens die Bedrohung der gegnerischen Bauern in Hinsicht auf ihre Position auf dem Spielfeld zu beurteilen und ob der gegenüberstehende Bauer bereits geschlagen wurde oder geschlagen hat („Frei Bauer“). Ihre zweite Aufgabe besteht in der Beurteilung der Sicherheit des Königs. Diese Beurteilung erfolgt immer und hat einen entscheidenden Einfluss auf die letztendliche Auswahl des Spielzuges.

Endgame logic and ROMs

Diese Funktionsblock beinhaltet eine Bibliothek mit einer Vielzahl von einfachen Spielzügen, die zuvor vom Schachgroßmeister eingetragen wurden. Über die implementierte Logik wird die Figurenstellung mit der Bibliothek verglichen. Am Ende des Spiels ist es von entscheidender Wichtigkeit, taktisch zu handeln (auf lange Sicht). Hier hilft auch die Möglichkeit, das gesamte Spiel zum Ende hin durchrechnen zu können, um den „richtigen“ Spielzug zu finden, da die Möglichkeit der Züge gering genug ist.

Game Phase Control

Die Game Phase Control bewertet Spielstellungen auf dem Spielfeld. So hat eine diagonale Anordnung von Bauern einen höheren Stellenwert, den es beizubehalten gilt, als eine lose Anordnung. Wiederum bekommen schlechte Anordnungen Minuspunkte.

7.4.4 Aufbau der langsamen Bewertung (slow Evaluation function)

Der Aufbau der langsamen Bewertung war das komplexeste Vorhaben in der gesamten Computerschachliteratur, das es bislang gibt (laut F-h. Hsu). Dieser Abschnitt beinhaltet ca. 88.000 einzelne Bewertungsfunktionen, die Einfluss auf das Ergebnis haben. Aufgrund ihrer benötigten Zeit wird Sie nur zu 15 % der Fälle angewandt. Die dafür nötigen Baugruppen belegen etwa die Hälfte des Schachchips.

Die langsame Bewertung wird mittels einer 3-Stufen-Pipeline mit Daten versorgt, welche ein 8x1 Array anspricht. Die nächste Baugruppe ist ein 40-plus Synchronisations-RAM, gefolgt von einem Addierbaum, welcher die Resultate sammelt.

Diese Funktion stellt Bewertungen wie z.B. Zughäufigkeit, Doppelbauer (hierbei befinden sich zwei Bauern hintereinander auf einer Linie), Blockade[3], eine verbesserte Königsicherheit und vieles mehr bereit.

Beispiel einer Bewertung anhand einer einfachen Rochade:
Bevor ein Statement abgegeben werden kann, muss vorher jede der drei Möglichkeiten (kleine, große und keine Rochade) berechnet werden. Ausgehend von diesen drei „sicheren“ Stellungen für den König, werden hier die einzelnen Gewichtungen, wie der Stellungen der Figuren, der Zustand der Königsburg (Anordnung der eigenen Figuren um den König), die Anwesendheit von feindlichen Bauern, die Farbverhältnisse, das neue Feld an sich, die „Strahlen“ (siehe hierzu Kapital „Zuggenerator“) um den König und so weiter berücksichtigt. Das Ergebnis der einzelnen Bewertungen, ist nachher eine lineare Gewichtung der Ergebnisse der drei Stellungen.

7.4.5 Smart Move Stack

Der „Smart Move Stack“ existierte in der vorherigen Version des Deep Blue noch nicht. Der alte Chip besaß lediglich einen normalen „Move Stack“, aber nicht den in Deep Blue II vorhandenen „Repetition Detector“. Der „Repetition Detector“ besteht aus einem Ringspeicher, gefüllt mit den letzten 32 möglichen Spielzügen. Dieser wird mittels eines Alpha-Beta Such-Algorithmus durchsucht. Im Speicher selbst wird einmal die neue Position gespeichert sowie die alte Position. Wenn nun kein Zug im Ringspeicher verdrängt wird, hat man die Erkenntnis, dass es sich hierbei um eine Wiederholung eines Zuges handelt und eine weitere Untersuchung des Zuges nicht vorgenommen werden braucht. Der „Repetition Detector“ erkennt zusätzlich zu den normalen Wiederholungen von Zügen, ob eine Situation schon mal vorhanden war und beugt so dem Aufeinandertreffen von identischen Stellungen vor. (Im Schach ist es so, dass nach Eintreten einer Stellung, zum dritten Mal in Folge, die Partie von der gegnerischen Partei als unentschieden gewertet werden kann.)
Die Zeitkomplexität des Detectors beträgt O(n), wobei n die Tiefe des Wiederholungsbuffers ist. In Software ist der „Wiederholungserkenner“ gewöhnlich als Hash-Table realisiert, welcher eine Zeitkomplexität von O(1) aufweist (laut F-h. Hsu).

[3] Die Blockade ist eine strategische Ausrichtung im Schach die darauf zielt, gegnerische Bauern am weiteren Vorrücken zu hindern.

7.4.6 Search Control

In der „Search Control“ wurde der Alpha-Beta Algorithmus nicht exakt realisiert. Es wurde ein sogenannter Minimum Fenster Alpha-Beta Algorithmus implementiert. Der Vorteil von diesem gegenüber dem Gebräuchlicheren ist, dass er keinen Werte-Stack benötigt. Der reguläre Algorithmus beinhaltet zwei temporäre Variablen, Alpha und Beta, die auf einem Stack gespeichert werden müssen. Der relativ neue Minimum Fenster Algorithmus kann nicht sagen, um wieviel eine Zugmöglichkeit besser ist als eine andere Zugmöglichkeit. Der Minimum Fenster Algorithmus sagt, welche Zugmöglichkeit besser oder schlechter ist und das, ohne genau zu sagen um wieviel. Ein weiterer Nachteil des Minimum Fenster Algorithmus gegenüber dem Alpha Beta Algorithmus ist, dass er am Ende noch einen letzten Zug bewerten muss, wenn zuvor der beste Zug gefunden wurde. Der Vorteil ist aber, dass man die Werte Alpha und Beta nicht noch zusätzlich auf dem Stack speichern muss.
Die „Search Control“ besteht aus einem 16-Bit Datenpfad und drei Zustandsautomaten, die den Datenpfad unter sich aufteilen. Zwei von den Zustandsautomaten kontrollieren zusätzlich indirekt noch den Zuggenerator. Der Datenpfad benutzt Addierer / Subtrahierer zum Berechnen der Bedingungsflags, welche die Suchalgorithmen eventuell für ihre nächsten Züge benötigen.

7.4.7 Performance des Schachchips

Dieser und er nachfolgende Abschnitt beinhaltet Informationen aus folgender Quelle: [5].

Um die Performance des Schachchips zu testen, wurde er in verschiedenen Systemen mit unterschiedlichen Konfigurationen eingesetzt. Im ersten Spiel, Anfang 1997, kam ein single Chip zum Einsatz, welcher mit 70 % der Taktrate betrieben wurde. Diese Maßnahme reduzierte die Leistung des Chips auf 7% bis 14% von seiner normalen Leistung. Dies war nötig, um die Leistungsfähigkeit des Schachchips gegenüber eines damals gebräuchlichen PC-Systems zu testen. Dieses PC-System hatte einen Pentium Prozessor mit 180 MHz. Als Software wurde ein sehr schnelles kommerzielles Schachprogramm eingesetzt. Zwei von diesen Systemen mussten in den frühen Tests des Chips herhalten. Von 10 Spielen, die gegeneinander gespielt wurden, gewann der Schachchip (single Chip) 10 Spiele. Bezogen auf die Elo-Zahl resultiert daraus ein mindestens 200 Punkte stärkeres Ranking.
Am Ende wurden noch weitere Spiele gegeneinander gespielt, von denen der Schachchip zwei verlor (von insgesamt 40 Partien). Dieses Ergebnis zeigte nun, dass der Chip gegenüber den kommerziellen Programmen 300 bis 500 Elo-Punkte höher liegt. Aber wie bereits im Kapital „Elo-Zahl“ erwähnt, lässt sich dieser Wert nur bedingt mit den Elo-Werten menschlicher Spieler vergleichen. Wenn nun der Schachchip einen groben Fehler machen würde und das immer wieder bei gleichem Spielverlauf, würde es ein erfahrener menschlicher Spieler ausnutzten. Programme können diese Fehler nur schwer so effizient ausnutzen. Es lässt sich zwar die Bewertungsfunktion dementsprechend anpassen, aber auch hier wir es die eine oder andere „Lücke im Wissen“ geben. Das menschliche Gehirn ist in diesen Dingen einfach dynamischer und passt sich der Situation besser an.

7.5 Spiel gegen Kasparov

Die 97er-Version des Deep Blue spielte nur sechs Spiele, alle gegen Kasparov. Deep Blue gewann das Match 3.5 : 2.5. Kasparov hatte damals ein offizielles Ranking von 2815 Elo-Punkten. Diese sechs Spiele wurden zur Bestimmung des Elo-Wertes von DeepBlue verwendet. Es ergab sich ein Wert von 2875. Da dieser Wert aber nur aus sechs Spielen ermittelt wurde, ist dieser Wert nicht besonders representativ.

Unter dem Link http://www.chessgames.com/index.html können alle Partien der Beiden nachgespielt und analysiert werden.

8. Hydra

Der Abschnitt „Hydra“ beruht zum größten Teil auf der Quelle [20]. Folgende Quellen wurden noch für Ergänzungen und Erläuterungen verwendet: [16], [17] [18] und [19].

Hydra ist ein international vorangetriebenes Projekt, welches von PAL Computer Systems in Abu Dhabi, Vereinigte Arabische Emirate finanziert wird. Das Kernteam besteht aus dem Programmierer Chrilly Donninger aus Österreich, den Forscher Ulf Lorenz von der Universität aus Paderborn (Deutschland), Schachgroßmeister Christopher Lutz, ebenfalls aus Deutschland, sowie dem Projektmanager Muhammad Nasir Ali aus Abu Dhabi.

Die Rechnercluster sind von der Firma Megware aus Deutschland hergestellt worden. Megware hat diese mit Unterstützung der Universität Paderborn entwickelt. Die FPGA’s, welche Bestandteil der PCI-Karten von AlphaData aus England sind, stammen von Xilinx.

Das Ziel des Hydraprojekts ist die Entwicklung der besten Schachmaschine der Welt. Als Höhepunkt sollte ein Sieg über einen Schachgroßmeister stehen. Um dies zu beweisen, entschied man sich, ein Turnier mit folgenden Teilnehmern auszutragen:

- Shredder, von Stephen Meyer-Kahlen, das dominierende Programm der letzten Zeit
- Fritz, von Frank Morsch, dem bekanntesten Programm
- Junior, von Air Ban und Shay Busghinsky, der damalig aktuelle Computerschachchampion
- Hydra, in der Erwartung das stärkste Programm zu sein

Diese vier Programme erreichten damals mehr als 95% der Punkte gegen die alten Programme in der Computerweltmeisterschaft 2003. Rechengeschwindigkeit, sowie anspruchsvolles Schachwissen, sind die zwei wichtigsten Kernpunkte eines Schachprogramms. FPGAs spielen zur Zeit die wichtigste Rolle in Hydra. Folgende Punkte sind bei den FPGAs besonders interessant:

- Mehr Platz auf den FPGAs, um mehr Wissen zu implementieren, bei sehr schnellen Abrufzeiten.
- FPGA Code kann man testen und die Software ändern, ohne lange ASIC – Produktionszyklen zu durchlaufen. Dieses ist ein entscheidendes Kriterium für FPGAs, weil somit ihre Entwicklung an den Schachprogrammen niemals endet und die dynamischen Prozesse im Computerschach in kurzer Zeit umgesetzt werden können. Daher: Flexibilität ist genauso wichtig wie Geschwindigkeit.
- Wie auch bei Deep Blue lassen sich auch hier Parallelitäten des Algorithmus ausnutzen. Siehe hierzu den Abschnitt „Die Software Architektur“.

8.1 Technische Beschreibung

Die Stärken des Programms, gegenüber menschlichen Spielern, sind seine Suchalgorithmen. Der verwendetet Algorithmus von Hydra stellt sich jedes Mal beim Übergeben einer Position die Fragen: 1. „Was kann ich tun?“ (wo kann meine Figur als nächstes hin), 2. „Was kann mein Gegner als nächstes tun?“ (der unmittelbar nachfolgende Spielzug des Gegners) und 3. „Was kann ich darauf erwidern?“.
Wie Deep Blue benutzt auch Hydra den Alpha-Beta Algorithmus. Dieser hat eine Komplexität von O(b^t/2) anstelle von O(b^t), bei einem Suchalgorithmus der nicht die uninteressanten Fälle von vornherein abschneidet. (Spielbaum-Tiefe = t und b = Anzahl der Abzweigungen im Spielbaum).

Obgleich dieser Algorithmus sehr effizient ist, kann ein Rechner am Anfang des Spiels nicht alle Werte einer Position, mit all ihren Möglichkeiten, im Schachspiel berechnen. Der daraus entsprechende Spielbaum würde einfach zu groß werden. Daher wird die Baumsuche als Approximationsannäherung bezeichnet.

8.2 Die Hardware Architektur

Hydra benutzt als GUI die Oberfläche von CheesBase / Fritz, die auf dem Betriebssystem Windows XP läuft. Es verbindet sich via Internet mittels SSL Verschlüsselung zum Linux-Cluster, welcher wiederum aus 8 Dual PC Serverknoten besteht. Jeder dieser Dual PC Serverknoten kann bis zu zwei PCI Busse gleichzeitig verwalten, an denen wiederum eine FPGA Karte steckt. Über ein Message Passing Interface (MPI) wird jeweils ein Serverknoten verwaltet, die mittels eines Myrint Netzwerks verbunden sind. Siehe dazu auch Abbildung 13: Hydra Netzwerk.

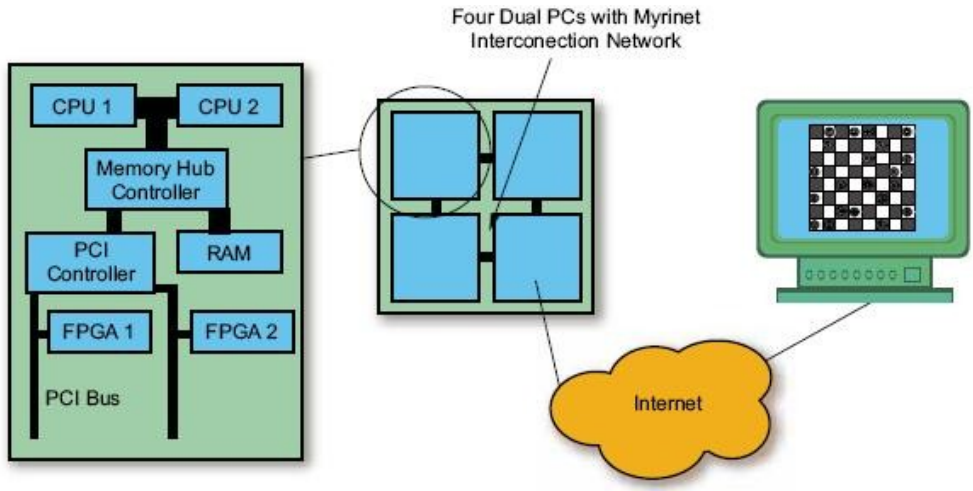

Abbildung 13: Hydra Netzwerk [20]

8.3 Myrint Netzwerk

Das Myrint Netzwerk ist ein proprietäres Hochgeschwindigkeits-Local Area Network-System von der Firma Myricom. Es wurde für die Vernetzung von Maschinen in einem Computercluster entwickelt. Ein wichtiger Vorteil gegenüber herkömmlichen Protokollen (wie z.B. Ethernet) ist, dass es im Vergleich deutlich weniger „overhead“ produziert und somit einen besseren Datendurchsatz ermöglicht.
Myrinet besteht physikalisch aus zwei Glasfaserkabeln, Sende- und Empfangsleitung, die mit einem einzigen Stecker an den Computer angeschlossen

werden. Die einzelnen Computer werden dann mit Hilfe von Routern und Switches miteinander verbunden. Myrinet enthält eine Reihe von Vorkehrungen zur Fehlertoleranz, die größtenteils von den Switches abgehandelt werden. Diese enthalten Flusskontrolle, Fehlerkontrolle und "heartbeat" Überwachung auf jeder einzelnen Verbindung. Die erste Generation der Hardware bot 512 Mbit/s Datenübertragung in beide Richtungen, spätere Versionen unterstützten dann 1,28 Gbit/s und 2 Gbit/s. Die neueste "Fourth-generation Myrinet" unterstützt 10 Gbit/s und ist auf der Hardwareebene (Physische Schicht) kompatibel zu 10 Gigabit Ethernet.
Laut der Firma Myricom benutzen 141 (28.2%) der im Juni 2005 ermittelten TOP500 Supercomputer Myrinet Technologie und machen Myrinet somit zu den am häufigsten verwendeten Hochgeschwindigkeitsnetz.

8.4 Message Passing Interface (MPI)

Mittels eines auf dem Communicating Sequential Process[4] basierenden Protokolls, das parallel Berechnungen auf verteilten, lose-gekoppelten Computersystemen ermöglicht, werden die einzelnen Serverknoten angesprochen.

8.5 Die Software Architektur

Die Software ist in zwei Teile aufgeteilt: in den Suchalgorithmus, welcher auf den Pentiumknoten des Linux-Clusters läuft und in den Software Co-Prozessor auf den Xilinx FPGAs. Die Idee hinter der Aufteilung ist wie folgt: Beim Abarbeiten des Suchbaums anhand der Suchreihenfolge werden die einzelnen Suchen parallel betrieben um die Balance der Auslastung dynamisch zu kontrollieren.
Der erste der speziellen Prozessoren, der P_o, bekommt die Suchanfrage und startet den Vorhersagealgorithmus so, als ob er die Suche sequenziell lösen würde. Sobald nun die Suchanfrage ins Netzwerk gekommen ist, stellen alle anderen Anfragen an die Prozessoren im Netzwerk, um Arbeit zu erhalten. Dieses Anfragen nach Arbeit passiert zufällig, so kann es vorkommen, dass nicht sofort alle mit Arbeit beschäftigt sind.

Anwendungsbeispiel

Wenn P_i ein Prozessor, der mit Arbeit beauftragt wurde eine derartige Anfrage erreicht, kontrolliert er, ob er nicht erledigte Aufgaben für seinen Suchbaum hat und diese an den Anfragenden weiterleiten kann, damit dieser für ihn die Bewertung vornimmt. Diese unerforschten Abschnitte sind Teile vom Knoten für P_i. Entweder sendet P_i, dass er die Arbeit bzw. die Aufteilung nicht machen kann, zurück oder er sendet ein entsprechendes Arbeitspaket an den auftragsstellenden Prozessor P_j. Folglich wird P_i Master und P_j startet eine sequenzielle Suche vom genannten Knoten. Somit geht hervor, dass ein Prozessor gleichzeitig Master als auch Slave sein kann.
Wenn P_j seine Arbeit fertig gemacht hat (wahrscheinlich mit Hilfe von anderen Prozessoren) sendet er seine Antwort an P_i. Die Master/Slave–Beziehung zwischen P_j und P_i ist vorbei, und P_j geht in den Ruhezustand (idle). Das Ganze passiert beim Eintritt einer neuen Suchanfrage ins Netzwerk an P_0 von Neuem.
Angenommen Rechner P_i hat ein falsches Arbeitspaket an P_j geschickt, so hat er die Möglichkeit, einfach ein neues Arbeitspaket (und hoffentlich dieses Mal das richtige

[4] Communicating Sequential Process (CSP) ist eine Prozessalgebra von Interaktionen und kommunizierenden Prozessen. Siehe auch Quelle [6]

Arbeitspaket) zu senden. P_j wird darauf hin aufhören, am alten Paket zu arbeiten und sich dem neuen Arbeitspaket widmen.
Eine weitere Möglichkeit für P_i, die Arbeit von P_j zu unterbrechen, ist das Senden eines „cutoffs“, welches andeutet, dass die Rechnung überflüssig ist und P_j daraufhin seine Arbeit einstellt.

Bei 12 bis 16 Clustereinheiten liegt die geschätzte Geschwindigkeit bei 36 Millionen Bewertungen pro Sekunde.

Ab einer bestimmten Tiefe des Suchbaums kann man mit Hilfe der Coprozessoren eine Lösung finden, hierfür wird die „Fine-Grain“ -Parallelität genutzt, die in der Applikation implementiert wurde.

Hierfür wurde auf jedem Chip ein komplettes Schachprogramm mit den jeweiligen Funktionen (Bewertungsfunktion, Zuggenerator und eine Funktion für das Zurückgehen bzw. Setzten) implementiert.

Im „Fine-Grain“-Design hat man einen schnellen, einfachen Zugsgenerator, welcher etwas anders arbeitet als ein Softwarezuggenerator. Im Prinzip konstruiert er zwei 8x8 Schachfelder. Die Module „GenAggressor“ und „GenVictim“ realisieren 64 Felder in jedem Schachfeld. Beide bestimmen, an welches Nachbarfeld ankommende Signale geschickt werden. Von nun an läuft die Suche von einem möglichen Zug genauso ab wie bei Deep Blue.

9. Fazit

Man kann beobachten, dass der Trend von spezieller Schachhardware zu reinen Softwarelösungen geht. Die Programme sind in Verbindung mit modernen PC's heute schon so stark, dass sie Spielstärken erreichen, die auf Großmeisterniveau liegen. Im Rahmen der World Chess Challenge 2006 in Bonn spielte das Schachprogramm Deep Fritz gegen Schachweltmeister Wladimir Kramnik. Der Vergleich endete mit 4:2 für Deep Fritz [26]. Man kann sich daher fragen, ob weitere Duelle zwischen Mensch und Maschine in Zukunft überhaupt noch Sinn machen.

Die Parallelität hält nun auch in die PC-Welt Einzug und dies machen sich Programme wie beispielsweise Deep Fritz zunutze. Dabei stellt sich die Verteilung der Arbeit auf die Prozessoren als große Herausforderung dar. So wurde Deep Fritz für das oben erwähnte Match auf einem System mit zwei Intel-Core-2 Prozessoren ausgeführt [26].

Es hat sich aber auch gezeigt, dass die reine Geschwindigkeit, mit der die Programme Stellungen bewerten, zweitrangig ist. In erster Linie kommt es auf eine gute Bewertungsfunktion mit entsprechendem Schachwissen und auf effiziente Algorithmen an.

10. Quellenverzeichnis

1. http://web.telia.com/~u85924109/ssdf/list.htm
   Rangliste von PC-Schachprogrammen, Abruf: 17. Januar 2007

2. http://www.schachklub-hietzing.at/elosystem.php
   Elo-System, Abruf: November 2006

3. http://de.wikipedia.org/wiki/Alpha-Beta-Suche
   Abruf: November 2006

4. http://www.cis.uab.edu/hyatt/pubs.html
   Technische Dokumentation des Programms Crafty, Prof. Robert Hyatt, Abruf: November 2006

5. Artikel in der Zeitschrift IEEE micro, IBM's Deep Blue Chess Grandmaster Chips, Feng-hsiung Hsu, March/April 1999 (Vol. 19, No. 2)

6. http://de.wikipedia.org/wiki/Communicating_Sequential_Processes
   Abruf: November 2006

7. http://de.wikipedia.org/wiki/Elo-Zahl
   Abruf: Januar 2007

8. http://de.wikipedia.org/wiki/Nullsummenspiel
   Abruf: November 2006

9. „Duden Informatik A-Z. Fachlexikon für Studium, Ausbildung und Beruf“, Volker Claus und Andreas Schwill, Bibliographisches Institut, Mannheim, Februar 2006

10. http://de.wikipedia.org/wiki/Deep_Blue
    Abruf: November 2006

11. www.swisseduc.ch/informatik/puzzles/spiel/docs/spieltheorie.doc
    Spieltheorie, Abruf: Dezember 2006

12. http://www.heise.de/ct/04/19/048/
    Frei erhältlicher CT-Artikel: Schachmonster, Computerschach zwischen PC-Programm und parallelisierter Software für ein Cluster-System, Abruf: November 2006

13. http://tournament.hydrachess.com/ahydra.php
    Homepage von Hydra, Abruf: Dezember 2006

14. http://www.bs.informatik.uni-siegen.de/www/lehre/ss05/pv/v10.pdf
    Parallelverarbeitung IBM RS6000, Abruf: Dezember 2006

15. http://de.wikipedia.org/wiki/Computerschach
    Abruf: November 2006

16. http://de.wikipedia.org/wiki/Schachprogramm
Abruf: November 2006

17. http://de.wikipedia.org/wiki/Message_Passing_Interface
Abruf: November 2006

18. http://de.wikipedia.org/wiki/Myrinet
Abruf: November 2006

19. http://en.wikipedia.org/wiki/Hydra_(chess)
Abruf: November 2006

20. Artikel in der Zeitschrift Xcell Journal, The Hydra Project, Chrilly Donninger, Second Quarter 2005

21. http://de.wikipedia.org/wiki/Erwartungswert
Abruf: Januar 2007

22. Selbsterstellte Grafik

23. http://www.valavan.net/mthesis.pdf
Parallel Alpha-Beta Search on Shared Memory Multiprocessors, Abruf: Januar 2007

24. http://iacoma.cs.uiuc.edu/~greskamp/pdfs/412.pdf
Parallelizing a Simple Chess Program, Abruf: Januar 2007

25. http://www.playwitharena.com
Arena – freie GUI für Schachengines, Abruf: Dezember 2006

26. http://www.rag.de/microsite_chess
Offizielle Wettkampfseite der World Chess Challenge, Abruf: Januar 2007