Parallelisierung - was bringt es in der Praxis?

Vorüberlegungen

Schachprogramme arbeiten am effektivsten sequenziell, d.h. auf lediglich einem Prozessor. Dafür haben sich die Programmierer in den letzten 50 Jahren die ausgefeiltesten Algorithmen überlegt. "Grundpfeiler" der Suchfunktion eines Schachprogramms ist der seit langem bekannte und im Artikel *Chaos-System Deep Engine* von Lars Bremer erläuterte Alpha-Beta-Algorithmus. Dieser funktioniert am effizientesten, wenn die einzelnen Knoten im Suchbaum hintereinander abgearbeitet werden. Parallelisiert man die Suche nun, wird es immer wieder vorkommen, dass ein Teil des Baumes umsonst abgearbeitet wurde, da ein parallel arbeitender Prozessor weiter vorne im Baum bereits einen Cut erzeugt hat, der die von einem anderen Prozessor geleistete Arbeit ad absurdum führt. Das muss nicht immer so sein, doch es kommt häufiger vor. Wie häufig so etwas passiert, hängt vom sonstigen Aufbau der Suche eines Schachprogramms ab. Manche Programme verhalten sich, was Pruning- (also Abschneid)-Techniken angeht, eher konservativ. Dann erreichen sie häufig nicht so hohe Suchtiefen, profitieren jedoch mehr von einer parallelen Suche als andere Programme, welche zum Teil radikal auf Vorwärtsabschneidung setzen. Diese wiederum haben das umgekehrte Problem: Sie kommen schon auf einem Prozessor auf ansehnliche Suchtiefen, doch mit zunehmender Anzahl an parallel arbeitenden CPUs wird die Suche immer ineffizienter, d.h. es wird immer häufiger die Teilarbeit einer CPU zunichte gemacht, da sich im Nachhinein herausstellte, dass dieser Zweig des Baumes abgeschnitten werde konnte.

Die oben beschriebene Problematik führt dazu, dass ein Schachprogramm auf 2 Prozessoren niemals doppelt so schnell arbeitet wie auf einem, auf 4 Prozessoren nicht doppelt so schnell wie auf zweien usf. Doch wie misst man eigentlich, wieviel Geschwindigkeitszuwachs ein zusätzlicher Prozessor bringt und welche Messgröße zieht man heran? Auf den ersten Blick erscheint es am naheliegendsten, die Knoten pro Sekunde, welche eine Engine abarbeitet, in den Blickmittelpunkt zu stellen. Doch hier liegt bereits der erste Denkfehler, denn die Engine zählt brav die Knoten, die von allen Prozessoren / Threads gleichzeitig untersucht wurden. Das werden mit 2 CPUs (je nach Parallelisierungsverfahren) annähernd doppelt soviele in der gleichen Zeit sein wie mit einem Prozessor. Wie oben bereits erläutert wurde, ist ein Teil dieser Arbeit jedoch "für die Katz" gewesen, hat also nichts dazu beigetragen, dass die Engine tiefer in den Suchbaum vordringen konnte. Wäre es sinnvoll, diese umsonst untersuchten Knoten mitzuzählen, wenn man die Effizienz der Parallelisierung messen möchte? Wohl kaum. Leider haben wir keine Möglichkeit, die "ineffizienten" Knoten auszusortieren und nur noch die effizienten zu zählen. Also bedienen wir uns einer anderen Messgröße: der Zeit, die nötig ist, um in einer bestimmten Stellung eine bestimmte Suchtiefe zu erreichen.

Diese Messgröße ist unter den Programmierern die anerkannteste, um die Effizienz der Parallelisierung zu bestimmen. Der Wert, welcher bei der Messung ermittelt wird, ist auch unter dem Begriff *Speedup* bekannt, also Beschleunigung. Wie wir oben gesehen haben, kann eine Verdoppelung der Prozessoren niemals zu einer Verdoppelung des Speedups führen. Doch wie stark werden denn nun bekannte Engines wie Fritz, Shredder, Rybka, Hiarcs & Co. von 2, 4 oder gar 8 CPUs beschleunigt? Nun, wenn es einen einfachen Test dafür gäbe, so hätten wir ihn durchgeführt und mit Stolz die knallharten Fakten in Form von Zahlen tabellarisch aufgelistet hier präsentiert. Doch so einfach ist es nicht, den Speedup zu bestimmen. Schuld daran ist, wie bereits im oben erwähnten Artikel von Lars Bremer erläutert wurde, der chaotische Ablauf der Suche, sobald mehr als ein Prozessor beteiligt ist. Der Suchalgorithmus arbeitet in diesem Fall nicht mehr deterministisch, so dass bei jeder Wiederholung des Experiments (Suche in Stellung *X* bis Tiefe *Y* und notiere die dafür benötigte Zeit *Z*) ein anderer Wert herauskommt. Zwar wird in der Regel der gleiche Zug gefunden (obwohl auch das nicht unbedingt so sein muss!), doch die Zeit bis zum Erreichen einer bestimmten Tiefe wird mal kürzer und mal länger sein. Wie stark die Schwankungen sind, hängt zum Teil auch noch von der getesteten Stellung auf dem Brett ab, denn sie bestimmt letztendlich, wie komplex der Suchbaum sein wird und welche Abschneidtechniken zum Tragen kommen werden.

Falls sie selbst einmal experimentieren möchten, empfehlen wir folgende Vorgehensweise:

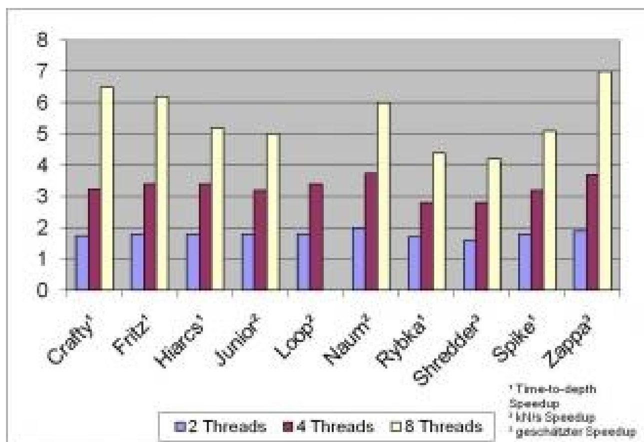
- Suchen Sie sich ein beliebiges Schachprogramm aus, von dem Sie die Effizienz der Parallelisierung bestimmen möchten. Natürlich muss dieses prinzipiell dazu fähig sein, zusätzliche Prozessoren nutzen zu können. Diese Fähigkeit erkennt man häufig an dem Zusatz *Deep* oder *MP* im Namen der Engine.
- Stellen Sie in den Engineeigenschaften die Anzahl der zu nutzenden CPUs (meist verbirgt sich dies hinter dem Namen *threads*) zunächst auf 1.
- Wählen Sie ein paar Stellungen aus, die möglichst unterschiedlich sein sollten (jeweils aus Eröffnungs-, Mittelspiel- und Endspielphase) und lassen Sie Ihre Engine bis Tiefe X rechnen (hier sollten Sie eine Tiefe wählen, welche die Engine nicht unter einer Minute erreichen kann). Dann notieren Sie sich die Zeit und testen die nächste Stellung.
- Nun erhöhen Sie in den Engineeigenschaften die Anzahl der zu nutzenden CPUs auf den Wert, von dem Sie den Speedup wissen möchten (dieser darf selbstverständlich nicht die Anzahl der tatsächlich in Ihrem Rechner vorhandenen CPUs überschreiten).
- Testen Sie jetzt jede der Stellungen mehrfach (5-10mal) wie oben beschrieben. Aus den Resultaten für jede einzelne Stellung bilden Sie einen Mittelwert, dadurch erhalten Sie den Speedup für diese spezielle Position. Wenn Sie aus den gerade errechneten Speedups einen weiteren Mittelwert bilden, erhalten Sie ein Resultat, das in etwa den gemittelten Speedup über den gesamten Partieverlauf angibt.

Fragen an die Programmierer

Da wir die auf der letzten Seite beschriebene Speedup-Testprozedur selbst nur ansatzweise durchführen konnten, haben wir ersatzweise die Programmierer der führenden Schachengines um Auskunft gebeten. Gleichwohl haben wir diese Gelegenheit genutzt, um weitere Fragen bezüglich der Parallelisierung ihrer Engines zu stellen:

- Wie groß ist der Speedup Ihres Programms für 2, 4 und 8 threads?
- Welches ist Ihre Messmethode für den Speedup?
- Konnten Sie den Speedup seit der ersten Parallelversion steigern und wenn ja, um wie viel Prozent? Halten Sie weitere Steigerungen für möglich?
- Glauben Sie, dass es eine Art Limit für die Effizienz der Parallelisierung Ihrer Engine gibt, über welches hinaus eine weitere Aufsplittung der Sucharbeit nicht mehr sinnvoll ist?
- Können Sie uns darüber berichten, wie die Parallelisierung in Ihrem Programm grob funktioniert (z.B. geteilte Hash-Tabelle etc.)?

Leider antworteten nicht alle angeschriebenen Programmierer und einige wollten auch lieber keine Interna zu ihrer Engine verraten. Bei zweien dieser Fälle (Shredder und Zappa) wollten wir jedoch trotzdem ein paar Werte bekommen und haben selbst nachgemessen. Leider konnten wir nur einen Kurztest durchführen, darauf wird in der folgenden Grafik auch hingewiesen. Betrachten Sie diese Werte also mit Vorsicht. Auch die anderen Werte sind nicht unbedingt vergleichbar, da die Programmierer teils unterschiedliche Messmethoden verwendeten. Die meisten wählten die auf der letzten Seite beschriebene *Time-to-depth* Methode, doch andere haben lediglich die Knotengeschwindigkeit gemessen, was aus genannten Gründen nicht so aussagekräftig ist.



Die Speedups der einzelnen Engines

Fritz, Hiarcs, Loop

Hier die Antworten von den Programmierern von Fritz, Hiarcs und Loop:

CSS: Wie groß ist der Speedup Ihres Programms bei 2, 4 und (falls getestet) 8 threads?

Mathias Feist (Fritz): 1.8, 3.4, 6.2

Mark Uniacke, Sebastian Böhme (Hiarcs): Der Speedup ist schwierig zu messen, da er von der Stellung und verschiedenen anderen Faktoren abhängt. Grob würde ich schätzen, dass er 1.8 auf 2 Prozessoren, 3.4 auf 4 Prozessoren und etwa 5 bei 8 Prozessoren beträgt.

Fritz Reul, Gerhard Sonnabend (Loop): Bisher noch nicht getestet, da LOOP 11/12 nur 2 Prozessoren unterstützt. An LOOP 13 Beta mit 64-Bit Unterstützung arbeite ich seit einigen Monaten. Das Ziel ist erstmal eine gute Skalierung für 2 und 4 Prozessoren. Ergänzung von Gerhard: 2CPU (2.4GHz) typisch im Mittelspiel 3600-3900 kN/s, 4CPU (2.4GHz) typisch im Mittelspiel 7600-7800kN/s.

CSS: Welches ist Ihre Messmethode für den Speedup?

MF: Zeit um eine bestimmte Suchtiefe zu beenden.

MU: Teststellungen und Lösungszeiten für allgemein lösbare Stellungen.

FR: Als erstes messe ich natürlich die Knotengeschwindigkeit im 1-Prozessorbetrieb. Durch den generischen Aufbau und die Verwaltung der Threads geht hier etwas Performance verloren. Dies konnte ich aber durch die neue 64-Bit Unterstütung von LOOP 13 Beta wieder herausprogrammieren. Anschliessend sammle ich mittels Teststellungen für Eröffnung, Mittelspiel und Endspiel einige Knotengeschwindigkeiten. Diese Referenzwerte vergleiche ich nun mit der der Knotenleistung im Dual-Betrieb. Derzeit erreiche ich nicht die zweifache Speed, da etwas Zeit für die Verwaltung der Threads verlorengeht. Erst, wenn diese Werte sich meinen theoretischen Erwartungen annähern, beginne ich mit den Speedup-Tests. Meistens durch Teststellungen oder durch automatisierte Zweikämpfe mit konstanter Rechentiefe. Im Taskmanager kann ich dann die Skalierung in Anhängiggkeit von den Rechentiefen gut messen und vergleichen.

CSS: Konnten Sie den Speedup seit der ersten Parallelversion steigern und wenn ja, um wie viel Prozent? Halten Sie weitere Steigerungen für möglich?

MF: Hier haben wir keine Daten.

MU: Ich bin sicher, dass noch Verbesserungen möglich sind. Um wieviel, kann ich allerdings nicht sagen. Der Raum für Verbesserungen liegt vor allem in dem Bereich, wenn man mehr als 2 Prozessoren verwendet.

FR: Ja, LOOP 13 Beta skaliert wesentlich besser, da als seine Vorgänger! Trotzdem sind noch Performance-Lecks vorhanden. Beispielsweise sind die Wartezeiten der "Slave-Threads" zu hoch. Hieraus resultieren natürlich Speedverluste. Ausserdem ist eine Einstellung für Mittelspielstellungen nicht optimal fürs Endspiel und umgekehrt. Das ganze Verfahren muss dynamischer werden.

CSS: Glauben Sie, dass es eine Art Limit für die Effizienz der Parallelisierung Ihres Programms gibt, über welches hinaus eine weitere Aufsplittung der Sucharbeit nicht mehr sinnvoll ist?

MF: Das ist schwer vorherzusagen, da wir keinerlei Erfahrung mit noch mehr CPUs haben.

MU: Es gibt einen abnehmenden Nutzen je mehr Prozessoren hinzugefügt werden, aber ich hatte noch nicht die Zeit, um dies genauer zu testen.

FR: Ich kann hierzu nichts Konkretes sagen, aber ab 16 Prozessoren soll die Skalierung wieder schlechter werden, soviel sagt zumidest die Literatur. Parallelisierungsansätze, die auch auf grösseren Systemen gut skalieren sind extrem kompliziert zu entwickeln und wegen sehr begrenzten Möglichkeiten der Fehlersuche auch derzeit nicht zu empfehlen.

CSS: Können Sie uns darüber berichten, wie die Parallelisierung in Ihrer Engine grob funktioniert (z.B. geteilte Hash-Tabelle etc.)?

MF: Dynamische Teilung des Suchbaums.

MU: Ich würde nicht so gerne ins Detail gehen, aber ich benutze Threads, welche soviel Information wie möglich teilen und kooperativ die Arbeit im Suchbaum aufteilen.

FR: Basierend auf Version LOOP 10.32 habe ich LOOP MP 11/12 entwickelt. Um die Rechnleistung (nps) weitgehend zu erhalten kommen hier parallele Prozesse zum Einsatz. Die Kommunikation läuft über eine

gemeinsame Hashtabelle. Da es sehr wenige Schnittstellen gibt und beide Prozesse ihren eigenen Adressarum besitzen ist dieser Ansatz sehr sicher und stabil. Allerdings ist die Ausbeute mit dieser Technologier spätestens ab 4 Prozessoren eher gering. Daher arbeite ich seit einigen Monaten an einem SplitPoint basierten Ansatz. Es war allerdings nötig LOOP 13 fast vollständig neu zu entwickeln, damit die Engine optimal auf 2 und 4 (später sicher auch 8) Prozessoren arbeitet.

Naum, Spike, Rybka

Hier die Antworten von den Programmierern von Naum, Spike und Rybka:

CSS: Wie groß ist der Speedup Ihres Programms bei 2, 4 und (falls getestet) 8 threads?

Alex Naumov (Naum): Die Knoten pro Sekunde steigen um 2 bei 2 Threads, bei 4 threads ist es eine Steigerung um 3.75. Wenn ich mich richtig erinnere, beträgt die Steigerung bei 8 threads ungefähr 6.

Volker Böhm (Spike): Bei 2 CPUs ca. 1.8, bei 4 CPUs ca. 3.2 und bei 8 CPUs in etwa 5. Aber: Nicht alles vom "fehlenden" Speedup ist wirklich verschenkt. Ein Teil der Zeit rechnen Threads in Varianten, die bei einer Single-Version nicht berechnet worden wären (wg. Selektivität). Wenn darunter dann doch ein guter Zug ist...

Vasik Rajlich (Rybka): Es ist schwierig, hier präzise Daten zu nennen, da der Speedup von sehr vielen Faktoren abhängt. Meine Schätzungen sind: 1.7, 2.8, 4.4.

CSS: Welches ist Ihre Messmethode für den Speedup?

AN: Knoten pro Sekunde ist eigentlich keine gute Messmethode für die Effizienz der parallelen Suche. Es gibt einen großen Such-Overhead, wenn man eine große Anzahl an Threads benutzt. Wenn es zum Beispiel einen Beta-Cutoff in einem Thread gibt, dann muss man die Arbeit aller anderen Threads, die an diesem Teilbaum arbeiten, über den Haufen werfen, denn diese Arbeit war sinnlos. Dieser Overhead kann bei 8 oder mehr Threads sehr groß werden (bis zu 50%).

VB: 100 Teststellungen bis zu einer festen Suchtiefe durchrechnen. Messen der Zeit im Vergleich zur Single Version.

VR: Rybkas parallele Suche ist leicht "konservativ". Wenn Information fehlt, wird die Multiprozessor-Version eher mehr Arbeit verrichten als weniger. Also sind die MP-Tiefen gegenüber denen der Single-Version leicht "besser" [Anm. d. Red.: Da weniger Cuts erfolgten, ist die MP-Suche etwas präziser]. Ich verwende die time-to-depth Methode und runde ein wenig auf.

CSS: Konnten Sie den Speedup seit der ersten Parallelversion steigern und wenn ja, um wie viel Prozent? Halten Sie weitere Steigerungen für möglich?

AN: Ich habe nicht soviel Zeit investiert, um die Parallelisierung zu verbessern, da ich glaube, dass sie bereits zu den besseren gehört. Wenn jedoch irgendwann die 8 CPU-Rechner weiter verbreitet sind, wird es eine Menge Arbeit auf diesem Gebiet geben. Bis dahin halte ich es für produktiver an der Bewertung und an der allgemeinen Suche zu arbeiten.

VB: Ja, ich fing bei ca. 2.0 für 4 Cores an. Ich glaube nicht, dass noch viel drin ist. Höhere Selektivität vermindert den Speedup.

VR: Die erste Version war sehr schlecht. Ich denke, dass sicherlich noch Raum für Verbesserungen da ist.

CSS: Glauben Sie, dass es eine Art Limit für die Effizienz der Parallelisierung Ihres Programms gibt, über welches hinaus eine weitere Aufsplittung der Sucharbeit nicht mehr sinnvoll ist?

VB: Die derzeitige Implementierung wird bei 16 Cores schon sehr schlecht aussehen. Mehr ist dann kaum sinnvoll.

VR: Ich weiß es leider nicht, denn mehr als 8 cores konnten wir bislang noch nichts ausprobieren.

CSS: Können Sie uns darüber berichten, wie die Parallelisierung in Ihrer Engine grob funktioniert (z.B. geteilte Hash-Tabelle etc.)?

AN: Naum benutzt den normalen Young Brothers Wait Concept (**YBWC**) Algorithmus für die parallele Suche mit einigen eigenen Verbesserungen, die die Effizienz und Geschwindigkeit verbessern. Diese Methode ist sehr effizient für eine kleine Anzahl von Threads, was bedeutet, dass die Knotenleistung fast perfekt ansteigt.

VB: Der Algorithmus heißt "dynamic tree splitting". Dabei sucht sich das Schachprogramm gute Punkte zum Teilen des Suchbaums, schickt mehrere Prozessoren auf die gleichzeitige Suche nach unterschiedlichen Zügen und führt das Ergebnis wieder zusammen. Natürlich gibt es eine geteilte Hash-Tabelle, die dient aber nicht zur Parallelisierung.

Etwas genauer: Ein Thread fäng an, ganz normal zu rechnen. An Positionen, die sich zum Aufteilen eignen ("All-Nodes"), wird eine spezielle Version der "Negamax"-Funktion aufgerufen, die eine Aufteilung des Baumes an dieser Position ermöglicht. In dieser Zeit laufen alle anderen Prozessoren in einer Schleife und schauen, ob es schon "freigegebene" Splitpositionen gibt. Wenn es freigegebene Splitpositionen gibt, dann wählen sie eine möglichst gute aus und fangen an der Splitposition zu rechnen an.

Dabei sind auch solche Features implementiert wie "Informieren anderer Threads über neue Alpha-Beta Grenzen" und die Möglichkeit, dass der ursprüngliche Thread, der auf die Fertigmeldung aller anderen Threads an der Splitting-Position wartet, selber wieder anderen Threads zur Hilfe kommt.

VR: Es ist nichts Spektakuläres: Ich benutze Prozesse statt Threads, die Split-Information wird über ein Memory-Mapped-File transportiert. Natürlich ist auch die Hash-Tabelle geteilt.

Junior und Crafty

Zum Schluss die Antworten von den Programmierern von Junior und Crafty:

CSS: Wie groß ist der Speedup Ihres Programms bei 2, 4 und (falls getestet) 8 threads?

Amir Ban (Junior): Der Speedup liegt bei ca. 1.8, 3.2 und 5.

Robert Hyatt (Crafty): Es wurden zu diesem Thema schon viele Daten veröffentlicht. Die Zahlen variieren, aber im Allgemeinen liegen sie bei 1.7-1.8 für 2 CPUs, bei 3.1-3.4 für 4 CPUs und bei 6-7 für 8 CPUs.

CSS: Welches ist Ihre Messmethode für den Speedup?

AB: Knoten pro Sekunde und die Leistung in Stellungstests.

RH: Grob gesagt lasse ich eine große Anzahl von Stellungen bis zu einer bestimmten Tiefe durchsuchen, zuerst mit einer CPU, dann mit 2, 4 und so weiter. Ich nehme die Zeit, die eine CPU brauchte und teile sie durch die Zeit für n CPUs, dann bekomme ich den Speedup für eine bestimmte Stellung. Das ist der einfachste mir bekannte Weg um ziemlich genaue Werte zu erhalten.

CSS: Konnten Sie den Speedup seit der ersten Parallelversion steigern und wenn ja, um wie viel Prozent? Halten Sie weitere Steigerungen für möglich?

AB: Die erste Version von Deep Junior spielte 1999. Seitdem gab es einige Verbesserungen, aber keine dramatischen. Ich habe z. Zt. keine kurzfristigen Pläne, an der Parallelisierung von Junior weiterzuarbeiten.

RH: Da gibt es viele Möglichkeiten: Z.B. NUMA (Non-Uniform Memory Architecture) Rechner zu verwenden oder die Parallelsuche auf bestimmte Eigenheiten des Rechners zu optimieren (Memory Interleaving ja/nein, shared/private cache, cache coherency design[MOESI/MESI/etc], Prozessorgeschwindigkeit etc).

CSS: Glauben Sie, dass es eine Art Limit für die Effizienz der Parallelisierung Ihres Programms gibt, über welches hinaus eine weitere Aufsplittung der Sucharbeit nicht mehr sinnvoll ist?

AB: Ganz sicher ist für den derzeitigen Algorithmus bei 16 threads das Maximum erreicht. Bereits bei 6-7 threads fängt er an sich zu "verschlucken".

RH: Ja und nein. Momentan würde ich nicht über 16 threads probieren wollen ohne etwas zu ändern. Aber die Änderungen sind nicht so schwierig. Ich warte lediglich darauf, entsprechende Hardware zur Verfügung zu haben, um die Änderungen testen zu können.

CSS: Können Sie uns darüber berichten, wie die Parallelisierung in Ihrer Engine grob funktioniert (z.B. geteilte Hash-Tabelle etc.)?

AB: Semidynamisches tree-splitting und das Young Brothers Wait Concept (**YBWC**).

RH: Der Algorithmus splittet den Suchbaum an jedem Punkt auf, an dem (a) wenigstens ein Zug untersucht wurde (YBW/principal variation splitting idea) und (b) ein Prozessor nicht ausgelastet ist. Der Hash ist geteilt, aber darüber wird die Arbeit nicht verteilt, es hilft nur der Alpha/Beta Suche, effizienter zu sein... Wann immer ein Prozessor keine Arbeit mehr hat, wird ein globaler "idle"-Zähler erhöht und alle aktiven Suchen beginnen damit, nach einem guten "split point" zu suchen (ein Knoten, an dem wenigstens ein Ast untersucht wurde, so dass wir hoffentlich nicht an einem Cut-Knoten splitten). Das Ganze ist rekursiv, so dass nicht alle Prozessoren am gleichen "split point" im Baum arbeiten müssen (dies tun sie in der Praxis so gut wie nie). Der Algorithmus ist dem DTS sehr ähnlich, welchen ich in den frühen 90er Jahren schrieb...

Fazit

Parallelisierung hilft einer Schachengine, in einer bestimmten Zeit mehr Stellungen zu untersuchen. Aus dem Umstand heraus, dass heutige Prozessoren zwar mehrere Rechenkerne zur Verfügung haben, sich dafür aber Taktsteigerungen im Vergleich zu früher nachgelassen haben, holt eine parallele Suche das Beste aus einer modernen CPU heraus. Wieviel das in der Praxis wirklich sein wird, hängt stark vom Schachprogramm und der jeweiligen Stellung ab. Die genannten Zahlen und die Aussagen der Programmautoren legen nahe, dass die meisten Engines schon recht effizient arbeiten, doch diese Effizienz mit steigender Anzahl von Prozessoren abnimmt. Bei 8 oder spätestens bei 16 threads scheint eine vorläufige Grenze für die "gängigen" Algorithmen zu sein. Hier liegt es bei den Programmierern, sich Optimierungen zu überlegen oder ggf. sogar ganz neue Parallelisierungsalgorithmen zu kreieren. Doch dies wird niemand ernsthaft angehen, bis nicht ein Prozessor mit mehr als 16 Rechenkernen zum Standard oder zumindest bezahlbar geworden ist. Warum auch, kennen Sie jemanden, der ein 32-fach System zu Hause hat?

Ein anderer Weg könnte der sein, die Rechenlast über den einzelnen Computer hinaus zu verteilen, also einen sog. *Cluster* zu verwenden. Hier werden einfach mehrere Rechner (möglichst über ein Hochgeschwindigkeitsnetzwerk) miteinander verbunden und arbeiten dann alle im Dienste des Schachprogramms. Dass hierfür die Engine speziell angepasst sein und auf den Rechnern eine bestimmte "Umgebung" bereit gestellt werden muss, macht dieses Verfahren bislang zum Ausseiter. Freilich sind einige Exoten bekannt, die auf Cluster setzen: Prominentestes Beispiel hierfür ist das Hardware-Monster *Hydra*, doch auch *Gridchess* (setzt eine clusterfähige Toga-Version ein) machte in letzter Zeit von sich Reden. Von *Zappa*, *ParSOS* und *Diep* existieren ebenfalls clusterfähige Versionen, denn diese Programme haben bereits bei einigen wenigen Turnieren auf Clustern bzw. "Supercomputern" gespielt (Zappa spielte bei der WCCC 2006 in Turin auf einem Supercomputer mit 1024 Rechenkernen). Für die Clusterversion einer Engine gilt natürlich das Gleiche wie bei einer normalen MP-Version: Ohne spezielle Anpassungen nimmt die Effizienz mit steigender Prozessorzahl weiter ab. Also ist Clustering zwar ein Weg, der Engine mehr CPUs zur Verfügung zu stellen, doch es erfordert den gleichen bzw. sogar mehr Aufwand, um diese auch effizient nutzen zu können. An weiteren Verbesserungsmöglichkeiten für ihre Schachprogramme wird es den Autoren in den nächsten Jahren deshalb wohl nicht fehlen...

Informationen zum Autor:
Timo Klaustermeyer