Parallele Suche in Spielbäumen

Stefan Büttcher

stefan@buettcher.org

Arbeitsgemeinschaft Positionsspiele

FAU Erlangen-Nürnberg

Motivation

Warum überhaupt Parallelisierung?

- Baumsuche dauert ewig, vielleicht geht es mit sehr vielen Prozessoren in erträglicher Zeit.
- Hoffnung: n Prozessoren schaffen die Arbeit in einem n-tel der Zeit.

Ist diese Hoffnung realistisch? Abwarten!

Parallelisierung: Positivbeispiele

- Matrix-Multiplikation:
  $n^3 \rightarrow log(n)$ ($\approx n^3$ CPUs),
- Reachability:
  $n^3 \rightarrow log^2(n)$ ($\approx n^3$ CPUs),
- ???

Parallelisierung: Negativbeispiele

- Max-Flow:
  $n^3 \rightarrow \sqrt{n}\, log^2(n)$ (∞ CPUs),
- Algorithmus von EUKLID
  $n^3 \rightarrow n \cdot log(n)$ (∞ CPUs),
- ???

Parallelisierung: Kenngrößen

Wann ist ein paralleler Algorithmus gut?

Wann ist er unbrauchbar?

- $speedup = \frac{best\ solution\ time\ on\ one\ processor}{solution\ time\ on\ n\ processors}$,
- $efficiency = \frac{speedup\ on\ n\ processors}{n}$.

Parallelisierung: Problemchen

Was ist zu beachten?

- *Granularität*
  Wie lange kann ein einzelner Prozessor arbeiten, bevor er sich wieder mit anderen abstimmen muss? (Kommunikationskosten!)
- *Lastverteilung*
  Wie kann sichergestellt werden, dass kein Prozessor verhungert? Wie kann Arbeit nachträglich auf andere Prozessoren verteilt werden?

Lässt sich MinMax parallelisieren? (1)

Annahme: Der Suchbaum ist ein *Baum* T mit Tiefe d und konstantem Verzweigungsfaktor b.

Dann ist die Zeit, die b^k Prozessoren zum Durchsuchen des Baums brauchen:

$$t(T, b^k) = \frac{1}{b^k}\, t(T, 1).$$

Es findet beinahe keine Kommunikation statt.

Besser kann niemals parallelisiert werden!

Lässt sich MinMax parallelisieren? (2)

Aber:

- Der Baum ist kein Baum,
- der Verzweigungsfaktor ist nicht konstant,
- die Tiefe ist nicht an allen Stellen gleich.

Dennoch:

- MinMax ist ganz harmlos und lässt sich recht gut parallelisieren.

Wiederholung: Suchtechniken (1)

Alpha-Beta-Pruning in all seinen Varianten

- Alpha-Beta,
- Negamax,
- Scout-Algorithmus,
- SSS*.

Alle Such-Algorithmen basieren auf Alpha-Beta-Pruning.

Ziel: Kleinster Suchbaum ($b^{\lceil \frac{d}{2} \rceil} + b^{\lfloor \frac{d}{2} \rfloor} - 1$ Blätter)

Wiederholung: Suchtechniken (2)

Alpha-Beta durchsucht genau dann den minimalen Baum, wenn in jedem Knoten immer zuerst der *beste* Nachfolgerknoten ausgewertet wird.

- Heuristiken zur Umsortierung der Züge sind enorm wichtig,
- Killer-Moves,
- History-Heuristiken.

Wiederholung: Suchtechniken (3)

In Wirklichkeit ist der Suchbaum gar kein Baum, sondern bloß ein gerichteter, azyklischer Graph.

- Hash-Tabellen zur Zwischenspeicherung bereits besuchter Teilbäume,
- Transposition Tables zur gleichzeitigen Verbesserung der Suchreihenfolge.

Lässt sich AlphaBeta parallelisieren?

Angenommen, die Wurzel des Suchbaums habe n Nachfolger. Dann hängt die Art der Suche im k-ten Teilbaum des Spielbaums wesentlich von den Ergebnissen der Suche in den $k-1$ Teilbäumen ab, die weiter links liegen.

- Parallelisierung schwierig.
- Wenn Parallelisierung möglich, dann nur mit *sehr viel* Kommunikation.

Möglichkeiten für paralleles Rechnen

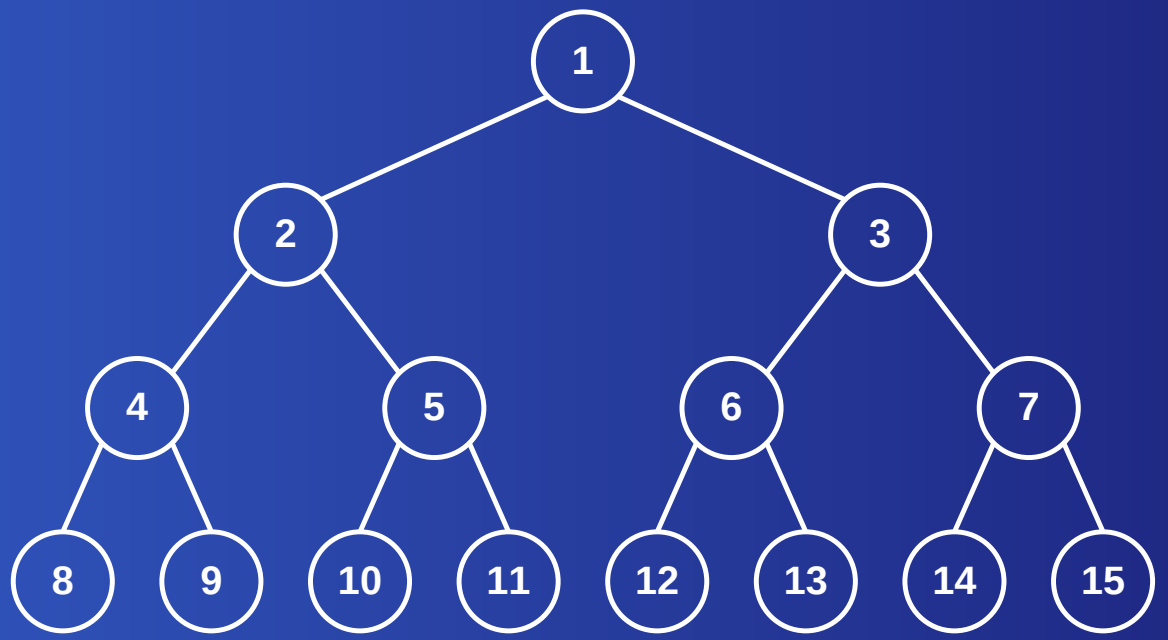
- *Parallele Heuristik / Parallele Zugerzeugung*
  (z.B. Belle, Deep Thought, Cray Blitz)
- *Parallele Fenstersuche*
  Bei Alpha-Beta lassen sich Fenster-Größen vorgeben, innerhalb derer gesucht wird.
  Aber: Der minimale Suchbaum hat $b^{\lceil \frac{d}{2} \rceil} + b^{\lfloor \frac{d}{2} \rfloor} - 1$ Blätter, bei der parallelen Suche mit n CPUs werden wenigstens $(n-1) \cdot b^{\lfloor \frac{d}{2} \rfloor} + b^{\lceil \frac{d}{2} \rceil} + b^{\lfloor \frac{d}{2} \rfloor} - 1$ Blätter besucht.
  $\Rightarrow$ Keine dolle Verbesserung!

Möglichkeiten für paralleles Rechnen

- Gleichzeitige Verwendung verschiedener Such-/Bewertungsverfahren, z.B.
  - einmal mit Verwendung von viel Hintergrundwissen,
  - einmal durch (Quasi-)Brute-Force Suche (Alpha-Beta, ...),
- Zerlegung des Suchbaums in mehrere (voneinander unabhängige?) Teilprobleme.

Baumzerlegung (1)

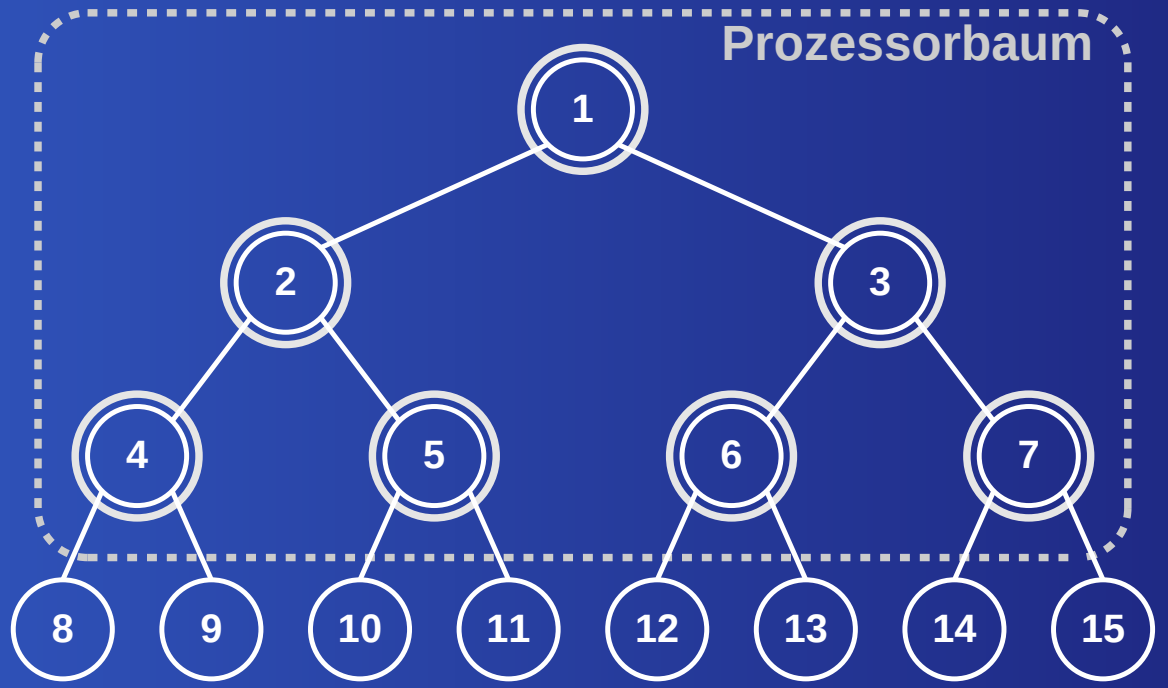
Algorithmus von Finkel und Fishburn



Spielbaum

Baumzerlegung (2)

Algorithmus von Finkel und Fishburn



Spielbaum

Baumzerlegung (3)

Algorithmus von Finkel und Fishburn

- Wenn nur der minimale Suchbaum durchsucht werden muss, ergibt sich ein $speedup = O(\sqrt{n})$.
- Simulationsergebnis: $speedup = 5,31$ bei Verwendung von 27 Prozessoren ($b_P = 3,\ d_P = 3$).

Baumzerlegung (4)

Mandatory Work First

So tun, als wäre der Baum perfekt geordnet.

- 1. Alle Blätter, die zum Minimalbaum gehören, parallel besuchen und bewerten; daraus den Wert der Wurzel berechnen.
- 2. Die verbliebenen Teilbäume parallel durchsuchen.

Experimentell: $speedup \approx 6$.

Baumzerlegung (5)

Def. Die Principal Variation („Hauptvariante”) ist ein Pfad von der Wurzel zu einem Blatt, bei der in jedem Knoten der jeweils beste Zug (aus Sicht von Min bzw. Max) ausgewählt wird.

Def. Die Actual Variation („tatsächlich gespielte Variante”) ist der beste Pfad von der Wurzel zu einem Blatt, der bisher untersucht worden ist.

Baumzerlegung (6)

Der PV-Split-Algorithmus (Principal Variation)

So tun, als wäre der Baum perfekt geordnet.

- 1. Den linken Teilbaum parallel durchsuchen.
- 2. Die anderen Teilbäume parallel asuchen.

Dabei werden die Teilbäume wieder mit Finkel-Fishburn durchsucht.

Experimentell: Bis zu 41% schneller als F-F pur.

(Marsland: $speedup = 3,2$ mit 4 CPUs.)

Was sagt die Theorie?

- Althöfer: $average\ speedup = \Theta(n)$ (n CPUs), falls der Baum wenigstens Tiefe $\Omega(n \cdot log(n))$ hat.
- Karp, Zhang: $worst - case\ speedup = \Theta(n)$ (n CPUs), falls der Baum wenigstens Tiefe $\Omega(n)$ hat.
- Harting: $speedup = (\frac{1}{76})^k \cdot n^k$ (n^k CPUs), falls der Baum wenigstens Tiefe n hat (gilt nur für 2-wertige binäre Bäume).

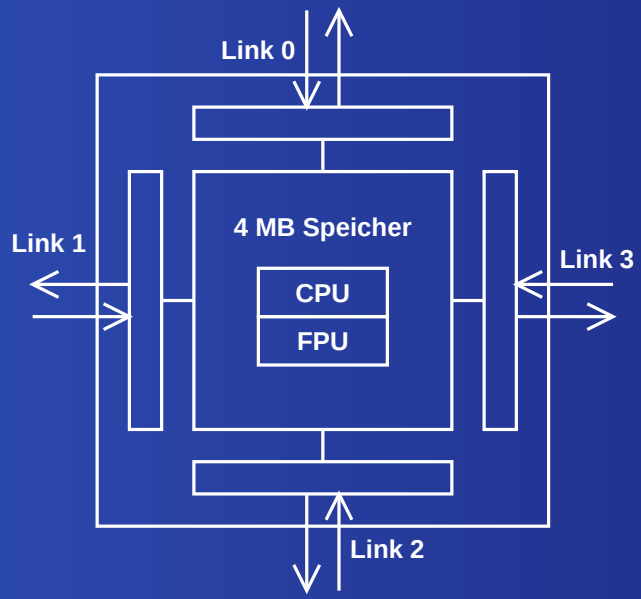
Konkretes Beispiel: Zugzwang

Zugzwang
Ein paralleles Schach-Programm

Universität Paderborn
$\sim$ 1992

Hardware: Inmos Transputer T800

4 MB Hauptspeicher. 25 MHz Prozessortakt.
4×20 MBit Vollduplex. Hardware-Scheduler.



Netztopologien

Interessante Maße bei der Analyse eines Computernetzes

- Durchmesser

$$\Delta = \max_{(u,v) \in V^2} \{distance(u,v)\},$$

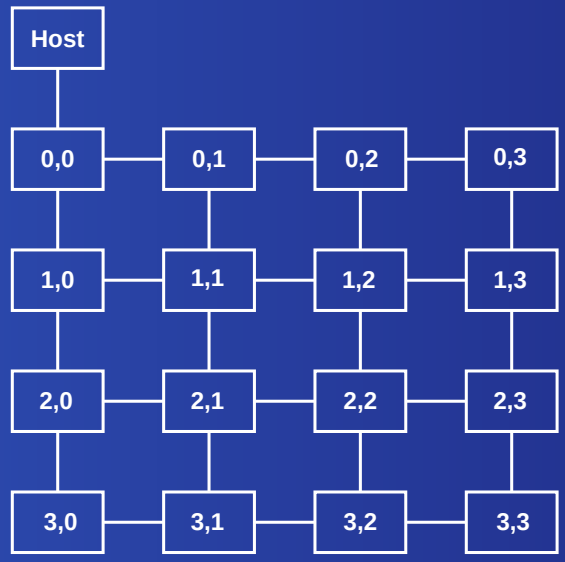
- mittlere Distanz

$$\frac{1}{|V|^2} \sum_{(u,v) \in V^2} distance(u,v).$$

Netztopologie: Grid

Ein $n \times m$ - Grid ist $G(n,m) = (V_G, E_G)$ mit

$$\begin{aligned} V_G &= \{(i,j) \mid 0 \leq i < n,\ 0 \leq j < m\}, \\ E_G &= \{((i,j),(i',j')) :\ |i-i'| + |j-j'| = 1\}. \end{aligned}$$

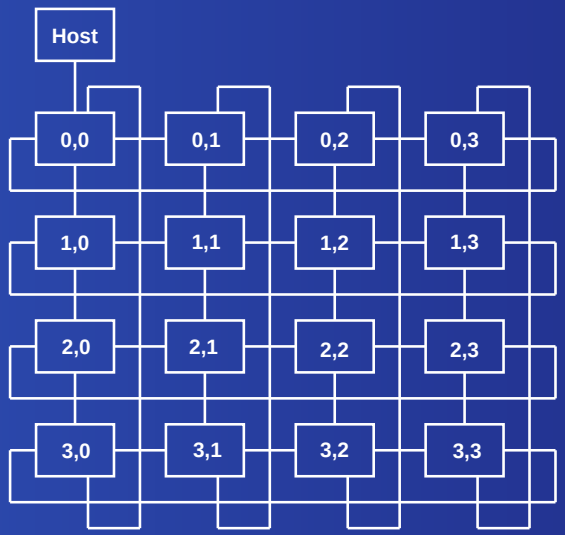


Ein 4x4 - Grid

Netztopologie: Torus

Ein $n \times m$ - Torus ist $T(n,m) = (V_T, E_T)$ mit

$$\begin{aligned} V_T &= V_G, \\ E_T &= E_G \cup \{((0,j),(n-1,j)),((i,0),(i,n-1)) : \ 0 \leq i < n, 0 \leq j < n\}. \end{aligned}$$



4x4 - Torus

Netztopologie: DeBruijn-Netz

Shuffle-Funktion

$s : \{0,1\}^n \rightarrow \{0,1\}^n, v_n..v_1 \mapsto v_{n-1}..v_1 v_n$

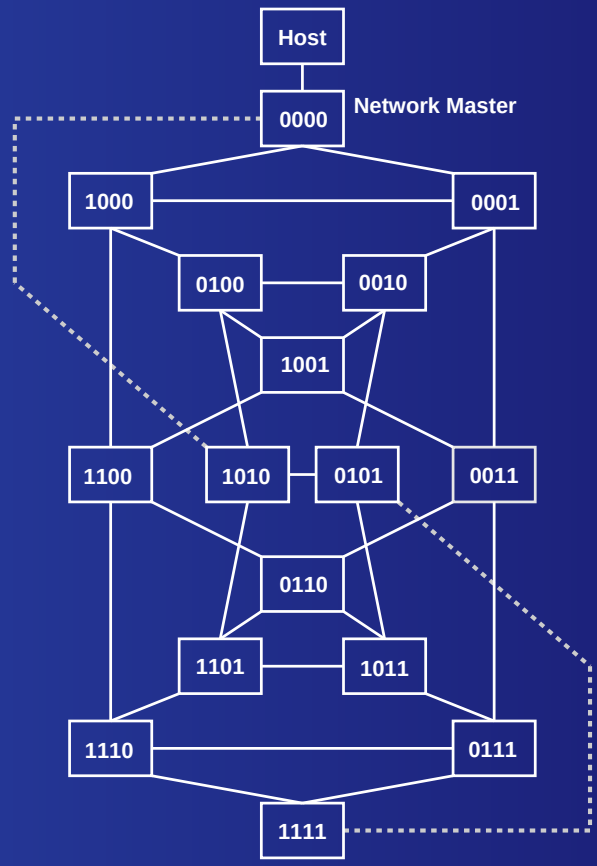
Exchange-Funktion

$e : \{0,1\}^n \rightarrow \{0.1\}^n, v_n..v_1 \mapsto v_{n-1}..v_1 \overline{v_n}$

$DB(n) = (V_n, E_n)$

$V_n = \{0,1\}^n$

$E_n = \{(v, s(v)), (v, e(v)) \mid v \in V_n\}$



DeBruijn-Netz der Dimension 4

Netztopologien: Vergleich

Durchmesser und mittlere Distanz

- DeBruijn: $\Delta, \Delta_\Phi \in \Theta(log(n))$,
- Grid: $\Delta, \Delta_\Phi \in \Theta(\sqrt{n})$,
- Torus: $\Delta, \Delta_\Phi \in \Theta(\sqrt{n})$.

Obwohl DeBruijn-Netze hinsichtlich Δ und Δ_Φ überlegen sind, verwendet man häufig Grids oder Tori. (einfache Struktur, skalierbar!).

Baumzerlegung in Zugzwang (1)

Dynamische Zerlegung

Mehrere Prozessoren durchsuchen Teilbäume und ermitteln dabei jeder für sich eine Actual Variation im Teilbaum. Alle Teilprobleme, die links von dieser Actual Variation liegen, können dann von anderen Prozessoren parallel durchsucht werden.

Baumzerlegung in Zugzwang (2)

Grundsätzliches

- Jeder Prozessor erzeugt Teilprobleme, die er ggf. an andere Prozessoren abgeben kann (Master-Slave-Beziehung, Workload).
- Veränderungen an der $\alpha\beta$-Grenze, die bei der Suche festgestellt werden, müssen anderen Prozessoren mitgeteilt werden.
- Manchmal ist es sinnvoll, Dinge *nicht* parallel zu erledigen (Kommunikations-Overhead).

Baumzerlegung in Zugzwang (3)

Def. Ein Nachfolger $v.j$ eines Knotens v der Actual Variation ist *frei*, falls:

- $v.j$ nicht gerade durchsucht wird,
- der Wert von $v.j$ noch nicht bekannt ist und
- $j > 1$ ist.

Arbeitlose Prozessoren verschicken er zufällig REQ-Nachrichten. Bekommt eine CPU ein REQ, kann sie mit einem freien Knoten antworten.

Baumzerlegung in Zugzwang (4)

Nachrichten, die verschickt werden:

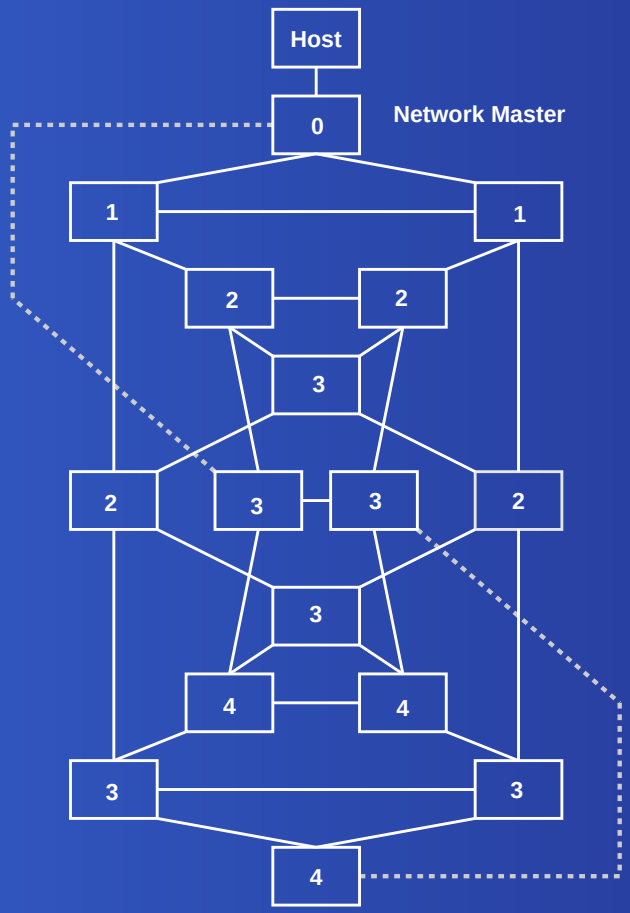
- REQUEST(sndr, rcvr),
- SUBPROBLEM(sndr, rcvr, v, α, β, v.depth, v.hash),
- CANCELLATION(sndr, rcvr, workload),
- RETURN(sndr, rcvr, value, pv, v.hash),
- CUTOFF(sndr, rcvr, v.hash),
- WINDOW(sndr, rcvr, α, β, v.hash).

Baumzerlegung in Zugzwang (5)

Routing ist teuer, also warum überhaupt globale Arbeitssuche?

- Wenn durch Globalisierung mehr Arbeitskräfte zur Verfügung stehen, sinken die Löhne. Das ist gut für den Shareholder-Value.
- Lokale Jobsuche führt dazu, dass viele Prozessoren nur sehr kleine Häppchen zugeteilt bekommen.

Baumzerlegung in Zugzwang (6)



DB(4) mit minimalen Tiefen der von den Prozessoren bearbeiteten Knoten

Jeder Prozessor schickt nur an seine Nachbarn Jobgesuche.

Resultat: 6 Knoten suchen mindestens auf Ebene 3, 3 Sogar mind. auf Ebene 4.

Baumzerlegung in Zugzwang (7)

Das Young-Brothers-Wait-Concept (YBWC)

In der Praxis tauchen häufig Bäume auf, die dem minimalen Baum sehr ähnlich sind (z.B. Faktor $1,5$). Es ist wünschenswert, dass man geringen Overhead im minimalen Baum hat.

YBWC: Die Suche in einem Nachfolger $v.j$ von v darf erst beginnen, wenn $v.1$ komplett ausgewertet ist.

Baumzerlegung in Zugzwang (8)

Def. Ein Nachfolger $v.j$ des Knotens v einer Actual Variation ist *YBWC-frei*, falls

- $v.j$ frei ist und
- $v.1$ bereits komplett ausgewertet ist.

Vorteile von YBWC: Minimaler Baum kann ohne Overhead durchsucht werden. Auch in nicht-minimalem Baum hilft YBWC:
$\alpha < F(v.1) < \beta$

$\Rightarrow$ Neues Suchfenster: $[\alpha, F(v.1)]$ bzw. $[F(v.1), \beta]$.

Baumzerlegung in Zugzwang (9)

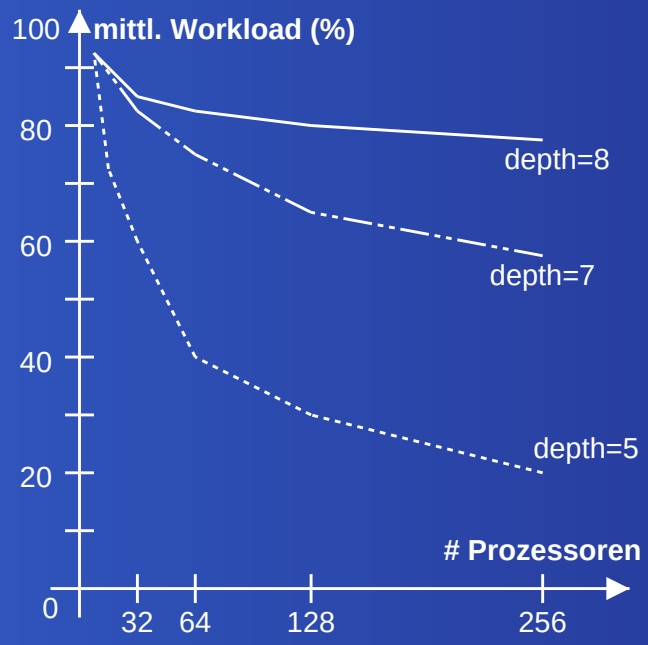
Weitere Details der Implementierung:

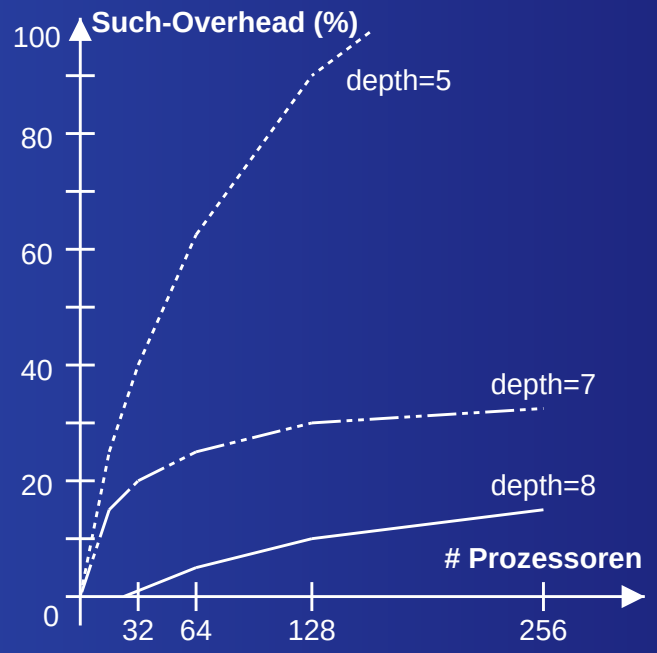
- lokale Killer-Moves, History-Heuristik, aber
- globale Transposition Table,
- Splitting ist nicht in allen Knoten gleich sinnvoll (vgl. Typ-1/2/3-Knoten, Knuth/Strehl),
- globale Arbeitssuche ist notwendig $\Rightarrow$ kleines Δ_Φ wünschenswert (daher DeBruijn-Netz).

Zugzwang: Ergebnisse (1)

Konfiguration: DeBruijn-Netz mit 2^k Knoten

k	2	3	4	5	6	8
Speedup	4,1	7,6	14,8	26,8	49,4	142,3





Zugzwang: Ergebnisse (2)

Bratko-Kopec-Testset (Schach)

No	k_1	t_1	k_{256}	t_{256}	Load	Overhead	Speedup
B01	661	2	950	1	0,0	43,7	2,0
B02	3498480	5972	4366554	58	67,2	24,8	103,0
B03	5882296	11255	7148739	96	63,5	31,5	117,24
B04	61791734	120014	40390075	428	77,1	-34,6	280,4
B08	258323	412	419999	18	16,7	62,6	22,9
B14	33382037	72628	26388785	318	74,5	-21,0	228,4
B24	18694564	35562	18383955	231	65,8	-1,7	154,0
Φ	14398898	27510	16890293	193	74,1	17,3	142,3

Literatur

Literatur

- R. Feldmann: „Game Tree Search on Massively Parallel Systems”. Universität Paderborn.
- C. P. Lu: „Parallel Search of Narrow Game Trees”. University of Alberta.
- M. Brockington: „A Taxonomy of Parallel Game-Tree Search Algorithms”. University of Alberta.