Katedra Informatiky
Fakulta Matematiky, Fyziky a Informatiky
Univerzita Komenského, Bratislava

Paralelné prehľadávanie herného stromu

(diplomová práca)

Peter Jurčovič

Vedúci: RNDr. Rastislav Královič, PhD.

Bratislava, 2005

Čestne prehlasujem, že som túto diplomovú prácu vypracoval samostatne s použitím citovaných zdrojov.

.....

Poďakovanie

Chcel by som poďakovať vedúcemu mojej diplomovej práce RNDr. Rastislavovi Královičovi, PhD. za tému, na ktorej bola radosť pracovať, a za všetky konzultácie a rady, ktoré mi k tejto práci poskytol.

Ďalej by som rád poďakoval Dr. Tomášovi Plachetkovi za poskytnutie pripomienok a vylepšujúcich návrhov. Tiež by som chcel poďakovať RNDr. Jaroslavovi Janáčkovi za poskytnutie prístupu k počítačovej sieti, na ktorej bolo možné vykonať potrebné merania, a za pomoc pri jej konfigurovaní.

Obsah

Kapitola 1

Úvod

Algoritmus na prehľadávanie herného stromu založený na alfa-beta osekávaní a intenzívnom používaní transpozičnej tabuľky je ukážkovým príkladom ťažko paralelizovateľného algoritmu. Pri rozdeľovaní jeho prehľadávacieho priestoru na podproblémy, ktoré je možné riešiť paralelne, sa spravidla nevyhneme tomu, aby procesory nevykonávali zbytočné, duplicitné výpočty. Pri jednoduchých metódach rozdeľovania je ich množstvo často tak veľké, že úplne zatieni prínos získaný použitím viacerých procesorov.

Je to dané jednak tým, že alfa-beta algoritmus využíva informácie získané pri prehľadávaní predošlých častí stromu na osekávanie prehľadávania v jeho nasledujúcich častiach. Ďalším dôvodom je skutočnosť, že iba pomerne malé percento vrcholov v herných stromoch používaných hier sú unikátne vrcholy a veľká väčšina vrcholov sú ich opakované výskyty. Sekvenčný algoritmus sa vie vďaka transpozičnej tabuľke veľkému množstvu takýchto opakovaných výpočtov úspešne vyhnúť.

Vhodné rozdelenie problému na paralelne riešiteľné podproblémy a realizácia transpozičnej tabuľky v distribuovanom systéme sú teda dva kľúčové a komplikované problémy, ktoré treba pri návrhu paralelného algoritmu riešiť.

Náplňou tejto práce je riešenie týchto problémov, a na základe týchto riešení navrhnutie pokročilého paralelného algoritmu, dosahujúceho uspokojivý výkon.

1.1 Hlavné výsledky

Hlavným výsledkom práce je úspešne navrhnutý a implementovaný paralelný algoritmus na prehľadávanie herného stromu. Navrhli sme preňho hybridnú distribuovanú transpozičnú tabuľku s originálne riešenou výmenou údajov medzi lokálnymi tabuľkami. Navrhli sme metódu centralizovaného rozdeľovania práce procesorom a heuristiku pre minimalizovanie množstva zbytočne pridelenej práce.

Vďaka všetkým týmto metódam a mechanizmom sme získali algoritmus dosahujúci uspokojivé zrýchlenie aj na distribuovanom systéme bez zdieľanej pamäti, v ktorom je komunikácia medzi procesormi relatívne veľmi nákladná.

1.2 Prehľad práce

V kapitole 2 uvádzame formálne definície potrebných základných pojmov, ako sú hra, herný strom alebo minimaxová funkcia. Ako príklad tiež uvádzame herný strom jednoduchého variantu hry Nim.

V úvode kapitoly 3 popisujeme základný minimaxový algoritmus na prehľadávanie herného stromu. Ďalej ukazujeme, že pre zložitejšie hry jeho úplne vyhodnotenie týmto algoritmom nie je prakticky realizovateľné, a preto presúvame pozornosť od problému vyriešenia hry k problému hrania hry.

V ďalšom texte diskutujeme pojem kvality hry a predstavujeme niekoľko vylepšení základného minimaxového algoritmu - alfa-beta osekávanie, iteratívne prehlbovanie a transpozičné tabuľky.

V kapitole 4 uvádzame miery pre určovanie efektívnosti paralelných algoritmov. Ďalej ukazujeme, aké je zrýchlenie základného paralelného algoritmu a popisujeme hlavné problémy, ktorými je úbytok zrýchlenia spôsobený.

Kapitola 5 predstavuje hlavný výsledok práce. Systematicky v nej popisujeme naše vlastné riešenia spomínaných problémov s paralelizáciou, a následne podrobne popisujeme i samotný navrhnutý algoritmus. Kapitolu končíme diskusiou niekoľkých implementačných otázok a problémov.

V kapitole 6 uvádzame výsledky vykonaných meraní, v ktorých analyzujeme sekvenčný alfa-beta algoritmus a jeho vylepšenia, zrýchlenie paralelného algoritmu i vplyv jednotlivých pokročilých metód v paralelnom algoritme na jeho výkon.

Zhrnutie výsledkov a diskusia ďalšej možnej práce sa nachádzajú v kapitole 7.

1.3 Terminológia

V kapitole 2 uvádzame formálne definície pojmov ako sú hra, herný strom, minimaxová funkcia alebo teoretická hodnota hry.

V texte budeme používať i množstvo ďalších pojmov – najmä z teórie grafov – ktorých presné definície z priestorových dôvodov nebudeme uvádzať. Všetky tieto pojmy však používame v súlade s definíciami, ktoré je možné nájsť v literatúre, napr. v [Die00].

V kapitolách venujúcich sa paralelným algoritmom častokrát spomíname distribuovaný systém skladajúci sa z uzlov, prepojených komunikačnými kanálmi a komunikujúcich prostredníctvom posielania správ. Okrem pojmu „uzol systému“ tu často používame v rovnakom význame aj pojem „procesor“.

Kapitola 2

Základné definície

2.1 Hra

Definícia 2.1.1 Hrou *budeme nazývať 7-ticu* $\Gamma = \{I, X, S, T, \delta, t, U\}$*, kde:*

1. $I = \{i_1, i_2, \ldots, i_n\}$ *je množina hráčov,*
2. X *je konečná množina stavov,*
3. $S \subseteq X$ *je neprázdna množina počiatočných stavov,*
4. $T \subseteq X$ *je neprázdna množina terminálnych stavov,*
5. δ*:* $X \to 2^X$ *je prechodová funkcia na množine stavov,*
6. t*:* $X \to I$ *je funkcia určujúca, ktorý hráč je na ťahu v stave* $x \in X$*,*
7. $U = \{u_1, u_2, ..., u_n\}$ *je množina úžitkových funkcií,*
   $\forall i, i = 1, \ldots, n : u_i : T \to \{-1, 0, 1\}$*.*

Z uvedenej definície (ktorá vychádza zo [Sla96] a [Gre05]) vyplýva, že uvažujeme iba hry s *úplnou informáciou* a so *striedavým vykonávaním ťahov hráčov*. Sú to také hry, kde každý hráč pozná všetky informácie a nič pre neho nie je skryté, ako je to napríklad pri kartových hrách. A ďalej v každom stave hry môže „vykonať ťah“ najviac jeden hráč.

Definícia 2.1.2 *Nech* $\Gamma = \{I, X, S, T, \delta, t, U\}$ *je hra. Nech* $x \in T$ *je terminálny stav hry a* $i \in I$ *je niektorý z hráčov hry. Ak* $u_i(x) = 1$*, potom budeme hovoriť, že* **v stave** x **vyhráva hráč** i*, ak* $u_i(x) = -1$*, tak v* x **prehráva hráč** i *a ak* $u_i(x) = 0$*, tak v* x **hráč** i **remízuje**.

Našu pozornosť ďalej sústredíme na tzv. *hry dvoch hráčov s nulovou sumou*.

Definícia 2.1.3 Hrou dvoch hráčov s nulovou sumou *budeme nazývať takú hru* Γ, $\Gamma = \{I, X, S, T, \delta, t, U\}$, *kde* $|I| = |U| = 2$ *a* $\forall x \in T$:

$$u_1(x) + u_2(x) = 0.$$

Hráčov pri takýchto hrách budeme nazývať MAX a MIN, čiže $I = \{MAX, MIN\}$. Z definície vyplýva, že $\forall x \in T : u_{MAX}(x) = -u_{MIN}(x)$. Z tohoto dôvodu nám stačí jediná úžitková funkcia $u : T \to \{-1, 0, 1\}$. Definíciu výhry, prehry a remízy upravíme tak, že ak $u(x) = 1$, tak vyhráva hráč MAX a prehráva hráč MIN, ak $u(x) = -1$, tak prehráva hráč MAX a vyhráva hráč MIN a ak $u(x) = 0$, tak hráč MAX aj hráč MIN v x remízujú.

V ďalšom texte budeme uvažovať už len výlučne hry dvoch hráčov s nulovou sumou, jedinou úžitkovou funkciou u a jediným počiatočným stavom s, a pre jednoduchosť ich budeme označovať ďalej iba ako „hry“. Týmto zúžením našej pozornosti nijak neutrpí zaujímavosť skúmaných hier. Veľa dobre známych, obľúbených a zložitých hier spĺňa definíciu hry dvoch hráčov s nulovou sumou. Ako príklady môžeme uviesť hry šach, dáma, othello alebo go.

2.2 Herný strom

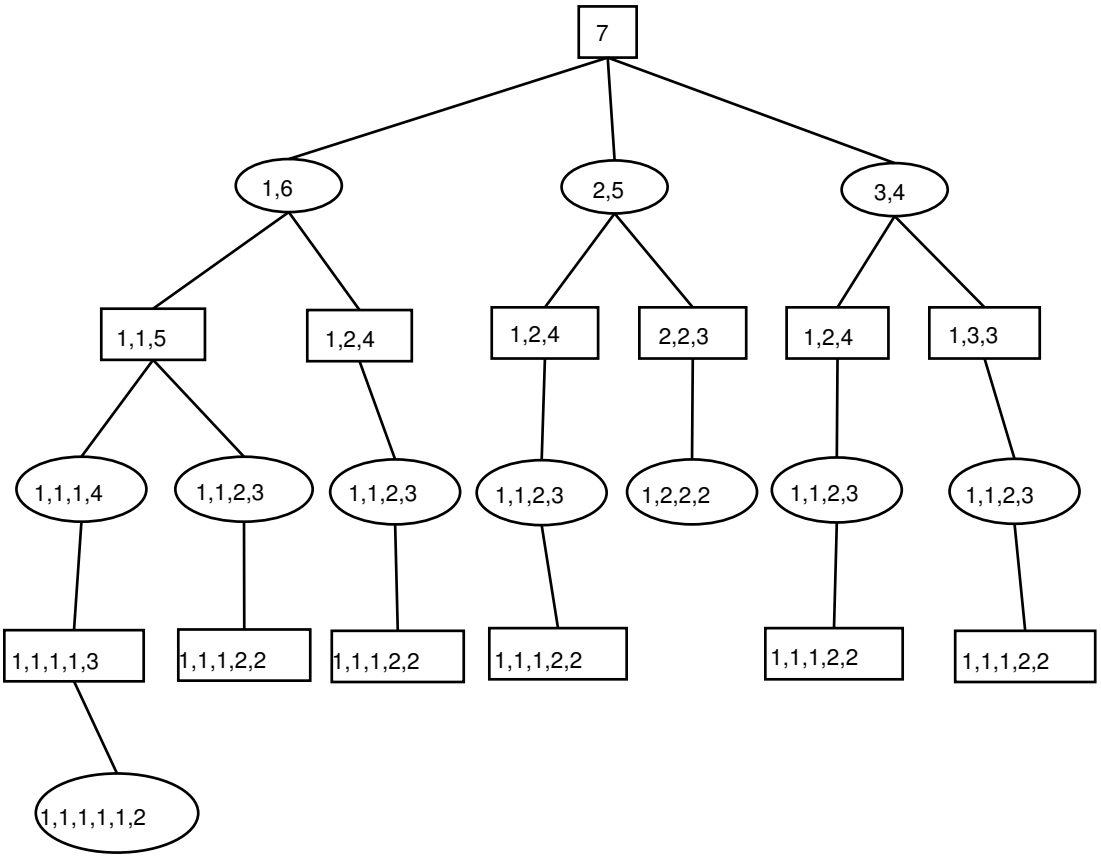
Herný strom je graf reprezentujúci aktuálny stav hry a možné budúce stavy hry. Jeho vrcholy zodpovedajú stavom hry a hrany ťahom. Formálna definícia nasleduje.

Definícia 2.2.1 Herným stromom *hry* Γ, $\Gamma = \{I, X, s, T, \delta, t, u\}$, *nazveme orientovaný graf* $G = (V, E, \Gamma)$, *kde* $V = X$ *je množina vrcholov a* $E \subseteq X \times X$ *je množina hrán taká, že* $\forall v_i, v_j \in X$ *platí:* $(v_i, v_j) \in E$ *práve vtedy, keď* $v_j \in \delta(v_i)$.

Pokiaľ pre $x \in X$ *platí, že* $t(x) = MAX$, *budeme hovoriť, že* **vrchol** x **je typu** MAX. *Ak* $t(x) = MIN$, *potom* **vrchol** x **je typu** MIN.

Tu je potrebné poznamenať, že napriek tomu, že graf $G = (V, E, \Gamma)$ nazývame herný strom, pre niektoré hry Γ môže obsahovať cykly a vrcholy môžu mať veľa vstupných hrán. Teda definíciu stromu podľa teórie grafov ([Die00]) nespĺňa. Avšak aj takéto grafy bývajú v literatúre označované ako „herné stromy“, takže aj my sa tejto konvencie budeme držať.

Na obrázku 2.1 uvádzame ako príklad herný strom variantu hry Nim so 7 zápalkami. Pri hre Nim sa pred dvoch hráčov položí určitý počet zápaliek, v našom prípade 7. Hráč na ťahu musí rozdeliť kôpku zápaliek na dve neprázdne kôpky obsahujúce rozdielny počet zápaliek. Napríklad 6 zápaliek možno rozdeliť na 1 a 5 alebo 2 a 4, ale nie 3 a 3. Hráč na ťahu, ktorý nemôže rozdeliť žiadnu kôpku zápaliek, prehráva.



Obrázok 2.1: Herný strom hry Nim so 7 zápalkami. Obdĺžnikové vrcholy reprezentujú stavy hry, v ktorých je na ťahu hráč MAX. V elipsovitých vrcholoch je na ťahu hráč MIN.

Obrázok 1 zároveň ukazuje, ako môžeme herný graf zakresliť do podoby stromu. Do stavu [(1,1,2,3),MIN] (a ďalších) existujú prechody z niekoľkých stavov. V strome však namiesto jedného vrchola s viacerými vstupnými hranami zakreslíme tento stav do viacerých oddelených vrcholov s jednou vstupnou hranou.

Základná otázka, ktorú si môžeme položiť o ľubovoľnej hre, znie: „Ktorý hráč vyhrá hru, ak hráči začnú hrať počnúc daným počiatočným stavom a každý si bude vyberať vždy ten najlepší možný ťah?“ Túto otázku vieme zodpovedať pomocou minimaxovej funkcie.

Definícia 2.2.2 Minimaxovou funkciou *hry* Γ, $\Gamma = \{I, X, s, T, \delta, t, u\}$, *nazveme funkciu* F, $F : X \to \{-1, 0, 1\}$ *definovanú rekurzívne nasledujúcim pred-*

pisom:

$$F(x) = \begin{cases} u(x) & ak\ x \in T; \\ \max\{F(y_1), F(y_2), \ldots, F(y_k)\} & ak\ x \in X \setminus T,\ t(x) = MAX \\ & a\ \delta(x) = \{y_1, y_2, \ldots, y_k\}; \\ \min\{F(y_1), F(y_2), \ldots, F(y_k)\} & ak\ x \in X \setminus T,\ t(x) = MIN \\ & a\ \delta(x) = \{y_1, y_2, \ldots, y_k\}. \end{cases}$$

Minimaxová funkcia je teda „rozšírením“ úžitkovej funkcie a definuje úžitkové hodnoty pre všetky stavy hry, nielen terminálne. Úžitková hodnota ľubovoľného stavu je potom podľa nej rovná úžitkovej hodnote toho terminálneho stavu, ktorý z daného stavu dosiahneme, ak si budú obaja hráči vyberať vždy tie „najlepšie“ ťahy. Teda ťahy vedúce do stavov s maximálnou minimaxovou hodnotou v prípade hráča MAX a minimálnou v prípade MIN.

Definícia 2.2.3 *Budeme hovoriť, že sme* **vyriešili hru** $\Gamma = \{I, X, s, T, \delta, t, u\}$, *ak sme určili hodnotu minimaxovej funkcie pre jej počiatočný stav s. Hodnotu $F(s)$ budeme nazývať* **teoretickou hodnotou hry** *(game-theoretic value).*

Poznamenajme, že v literatúre (napr. [SL96] alebo [A+94]) sa takto definované vyriešenie niekedy nazýva vyriešením hry vo *veľmi slabom zmysle*. Pre vyriešenie hry v *slabom zmysle* sa potom vyžaduje, aby sme okrem vypočítania teoretickej hodnoty hry našli aj *cestu* v hernom strome z počiatočného vrchola do terminálneho vrchola s touto hodnotou (a to takú, pri ktorej si bude oponent vyberať vždy tie najlepšie ťahy). Pre vyriešenie v *silnom zmysle* sa vyžaduje vypočítanie hodnoty minimaxovej funkcie a nájdenie príslušnej cesty v hernom strome pre každý stav hry.

Daná hra sa dá vyriešiť prostredníctvom prehľadávania jej herného stromu. Niekoľko základných i pokročilejších algoritmov na prehľadávanie herného stromu si ukážeme v nasledujúcich kapitolách.

Kapitola 3

Sekvenčné prehľadávanie herného stromu

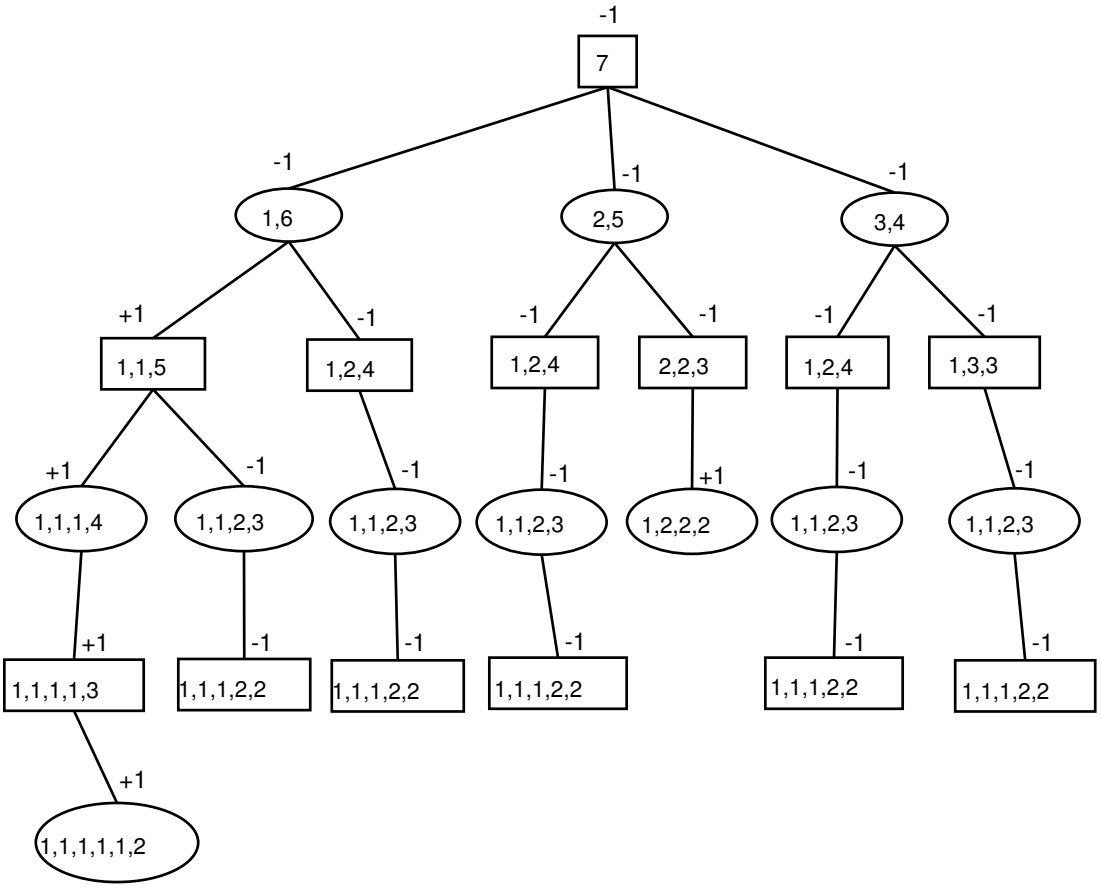
3.1 Minimaxový algoritmus

Základným algoritmom na počítanie minimaxovej hodnoty zvoleného počiatočného stavu hry je algoritmus na prehľadávanie grafu do hĺbky s post-order vyhodnocovaním vrcholov, ktorý sa tradične nazýva *minimaxový algoritmus*.

```
Algoritmus: Minimaxový algoritmus
Vstup: hra Γ = {I, X, s, T, δ, t, u}, stav v ∈ X

   procedure minimax(v)
   begin
      if v ∈ T then
         return u(v)
      else begin
         nech δ(v) = {y_1, y_2, ..., y_k}
         for i := 1 to k do
            v_i := minimax(y_i)
         if t(v) = MAX then
            return max{v_1, v_2, ..., v_k}
         else
            return min{v_1, v_2, ..., v_k}
      end
   end
```

Na obrázku 3.1 uvádzame herný strom hry Nim so 7 zápalkami s vrcholmi ohodnotenými minimaxovým algoritmom.



Obrázok 3.1: Herný strom hry Nim so 7 zápalkami s vrcholmi ohodnotenými algoritmom minimax. Ako vidno, hráč MAX nemôže pri optimálnej hre hráča MIN nikdy zvíťaziť.

Minimaxový algoritmus sa dá úspešne použiť na riešenie jednoduchých hier s malými hernými stromami. Avšak herné stromy zložitejších hier (šach, dáma, atď) obsahujú tak obrovský počet vrcholov, že ich prehľadanie nie je prakticky realizovateľné.

Uvažujme hru $\Gamma = \{I, X, s, T, \delta, t, u\}$, pre ktorú platí, že $\forall x \in X : |\delta(x)| = b$, (teda herný strom pre Γ bude mať pevný stupeň vetvenia b) a ďalej platí, že dĺžka cesty z počiatočného vrchola s do ľubovoľného vrchola $z \in T$ je d. Takýto herný strom nazývame *b/d-uniformný strom*. Potom počet listov herného stromu pre Γ je b^d a počet všetkých jeho vrcholov je $\sum_{i=0}^{d} b^i \leq b^{d+1}$.

Herné stromy skutočných hier obvykle nemajú pevný stupeň vetvenia či hĺbku listov, ale určitý odhad ich veľkosti možeme urobiť, keď budeme uvažovať ich priemerný stupeň vetvenia a priemernú hĺbku listov. Napríklad v šachu je priemerný počet ťahov v ľubovoľnej pozícii 35 a partie často majú až 50 ťahov (100 polťahov). Odhadovaná veľkosť jeho herného stromu je teda 35^{100} (okolo 10^{154}) vrcholov (i keď jeho graf má iba približne 10^{40} vrcholov [RN02]). Ak by sme uvažovali počítač, ktorý by zvládol vyhodnotiť až 10^9 vrcholov za sekundu, i tak by

trvalo prehľadanie a vyhodnotenie všetkých 10^{40} vrcholov (teda vyriešenie hry v silnom zmysle) až 10^{23} rokov.[1]

Pre zložité hry (ako sú šach, dáma, othello alebo go) teda výpočet teoretickej hodnoty minimaxovým algoritmom nie je prakticky realizovateľný. Minimaxový algoritmus (a jeho zlepšenia) však môžeme s úspechom použiť na to, aby sme vytvorili program schopný pomerne úspešne „hrať“ danú hru. Čiže program, ktorý na vstupe dostane nejakú hernú pozíciu a pre zvoleného hráča vypočíta a vyberie čo najlepší ťah, ktorý sa dá z danej pozície urobiť.

Hlavná myšlienka algoritmu na hranie hry je jednoduchá. Nebudeme prehľadávať herný strom až do listových vrcholov. Namiesto toho si zvolíme maximálnu hĺbku, a prehľadávať budeme iba vrcholy, ktoré ležia v podstrome určenom touto hĺbkou. Keď dosiahneme listy tohto podstromu, ohodnotíme ich zvolenou *heuristickou evaluačnou funkciou*. Vnútorné vrcholy následne vyhodnotíme minimaxovým algoritmom podobne ako pri počítaní teoretickej hodnoty hry. Upravený minimaxový algoritmus pre hranie hry potom môže vyzerať takto:

Algoritmus: Minimaxový algoritmus pre hranie hry
Vstup: hra $\Gamma = \{I, X, s, T, \delta, t, u\}$**, heuristická evaluačná funkcia** $h : X \to R$**, stav** $v \in X$**, hĺbka** $d \in N$

```
procedure minimax(v, d)
begin
   if d = 0 or v ∈ T then
      return h(v)
   else begin
      nech δ(v) = {y_1, y_2, ..., y_k}
      for i := 1 to k do
         v_i := minimax(y_i, d − 1)
      if t(v) = MAX then
         return max{v_1, v_2, ..., v_k}
      else
         return min{v_1, v_2, ..., v_k}
   end
end
```

Evaluačná funkcia h, $h : X \to R$, je tu používaná ako aproximácia presnej minimaxovej hodnoty pre vrcholy v maximálnej hĺbke d. Jej úlohou je rozhodnúť,

[1] Ako je ukázané v [SL96], niektoré hry je možné vyriešiť triviálne, a to aj napriek ich veľkej priestorovej zložitosti. Dôležitá je potom okrem priestorovej zložitosti totiž aj tzv. *rozhodovacia zložitosť*, ktorá udáva „obtiažnosť výberu správnych rozhodnutí“.

či je uvažovaný stav s najväčšou pravdepodobnosťou výherný, alebo či je s najväčšou pravdepodobnosťou prehrou, a či remízou. Spôsoby, akými to bude robiť, môžu byť pre rôzne hry veľmi odlišné.

Príkladom jednoduchej evaluačnej funkcie pre hry ako šach či dáma je funkcia počítajúca materiálnu hodnotu vlastných a súperových figúrok na šachovnici. Začína s 0 a za každú vlastnú figúrku f na šachovnici pripočíta číslo w_f (vyjadrujúce hodnotu – váhu figúrky f), a za každú súperovu figúrku g odpočíta číslo w_g.

V praktických programoch sa používajú oveľa zložitejšie funkcie, berúce do úvahy okrem materiálnej hodnoty aj množstvo ďalších činiteľov. J. Schaeffer popisuje ([SL96]) evaluačnú funkciu svojho programu *Chinook*, ktorý je držiteľom titulu majstra sveta ľudí i počítačov v dáme, nasledovne: „*Evaluačná funkcia má 25 heuristických komponentov, ktoré sú ováhované a zosumované na získanie výsledného ohodnotenia pre pozíciu. Hra je rozdelená do štyroch fáz a každá má vlastnú množinu váh. Definície heuristík a ich váhy sú výsledkom dlhých diskusií s expertom na dámu.*“

Ak teda sústredíme pozornosť na problém hrania danej hry, narazíme veľmi rýchlo na dve základné otázky. Ako dosiahnúť, aby program *hral hru* čo najlepšie? A ako zadefinovať alebo odmerať samotnú *kvalitu jeho hry*?

3.2 Vylepšovanie kvality hry

Napriek tomu, že pojem *kvalita hry* (resp. kvalita *hrania hry*) sa môže zdať ťažko formálne definovateľný, nie je tomu nutne tak. Existuje viacero modelov, ktoré sa snažia kvalitu hry hráča vyjadriť číselne. Významné zásluhy má v tejto oblasti Arpád Elo, ktorého formuly pre počítanie tzv. *ELO rejtingu* sú používané v šachu ako i v mnohých ďalších známych hrách.

ELO rejting je číslo vyjadrujúce relatívnu zručnosť (*skill*) hráča, či presnejšie do ktorej oblasti pravdepodobnostného rozdelenia náhodnej premennej „zručnosť“ hráč patrí. Na začiatku má každý hráč priradený určitý počiatočný rejting (napr. 1500). Vždy, keď sa hráči stretnú v zápase, ich rejtingy sú upravené podľa jeho výsledku. Podľa hráčovho aktuálneho rejtingu a rejtingu jeho protivníka je vypočítané, aký výsledok by mal hráč dosiahnúť. Pokiaľ sa jeho skutočný výkon zhoduje s očakávaným, rejting dobre odráža skutočnosť a nie je treba ho meniť. Pokiaľ je skutočný výkon odlišný od očakávaného, je treba hráčov rejting zmeniť - zvýšiť alebo znížiť. (Podrobnejší popis ELO rejtingu možno nájsť vo [Wik05] alebo v pôvodnej Elovej knihe [Elo78].)

Ako vplývajú rôzne parametre algoritmov prehľadávajúcich herné stromy na kvalitu hry programu (meranú napr. pomocou spomínaného ELO rejtingu), bolo subjektom početných výskumov. Veľmi populárne sú experimenty, v ktorých proti sebe hrá rovnaký program, avšak s rôzne nastavenými parametrami. Pokusy so šachovými programami ukazujú, že zvýšenie prehľadávacej hĺbky o 1 má za ná-

sledok zvýšenie ELO rejtingu programu o 100-250 bodov. Pritom ELO rejting je navrhnutý tak, že ak má hráč A o 200 bodov vyšší rejting ako hráč B, je 75% pravdepodobnosť, že hráč A hru vyhrá. Z ďalších pokusov zasa vyplýva, že prínos zvýšenej hĺbky prehľadávania je pre programy so sofistikovanou evaluačnou funkciou menší než pre programy s jednoduchšou funkciou. (Viac informácii možno nájsť v [Hei98] alebo [JS97].)

Teda experimentálne výsledky podporujú hypotézu, že oba hlavné parametre algoritmu na hranie hry – teda hĺbka prehľadávania a heuristická evaluačná funkcia – majú významný vplyv na výslednú kvalitu hrania. To znamená, že snaha o čo najväčšiu presnosť evaluačnej funkcie a snaha o čo najväčšiu hĺbku prehľadávania sú odôvodnené.

Evaluačné funkcie sú spravidla výrazne závislé od konkrétnej uvažovanej hry, a preto pre nás nie sú veľmi zaujímavé. Našu pozornosť teda sústredíme radšej na problém zvyšovania prehľadávacej hĺbky.

Väčšiu hĺbku prehľadávania môžeme dosiahnuť viacerými cestami. Najjednoduchším riešením je použitie výkonnejšieho počítača. Zaujímavejšie prístupy sú zvyšovanie efektívnosti prehľadávacieho algoritmu a paralelizácia prehľadávacieho algoritmu.

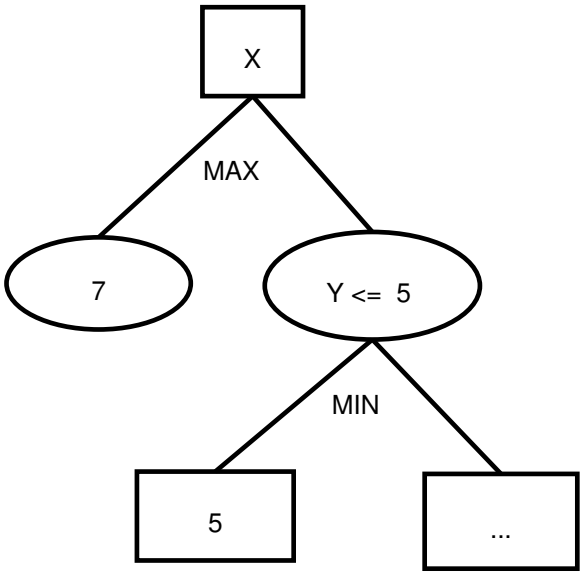
Keďže minimaxový algoritmus rieši problém veľmi priamočiaro a hrubou silou, necháva nám veľa priestoru na zlepšenia zvyšujúce efektívnosť. Najvýznamnejšie a najznámejšie zlepšenie predstavuje *alfa-beta osekávanie*. Ďalšie významné vylepšenia sú *iteratívne prehlbovanie* a *transpozičné tabuľky*. Bližšie si ich popíšeme v nasledujúcich kapitolách.

K ďalším pokročilým technikám patria algoritmy ako *Negascout*, *Null Move Search*, *Singular Extensions*, *Aspiration Search* či *Conspiracy Numbers Search* a heuristiky ako sú *killer heuristic*, *history heuristic*, *countermove heuristic* a ďalšie. Celú rodinu algoritmov, ktoré sa snažia ešte viac nahradiť hrubú silu „inteligenciou“, tvoria tzv. *selektívne prehľadávacie algoritmy*. Tie neprehľadávajú celý herný strom do zvolenej fixnej hĺbky (ako to robia vyššie uvedené „neselektívne“ algoritmy), ale v každom kroku si vyberú iba niekoľko málo vrcholov na ďalšie preskúmanie. Týmto sa však v našej práci nebudeme ďalej venovať. Viac informácii možno nájsť napr. v [Upt99], [Fel97], a mnohých ďalších.

3.3 Alfa-beta osekávanie

Technika alfa-beta osekávania (*alpha-beta pruning*) vychádza z pozorovania, že niektoré „príliš zlé“ (resp. „príliš dobré“) hodnoty vrcholov herného stromu určite nebudú pri maximalizovaní (resp. minimalizovaní) vybraté. Preto môžeme prehľadávanie vrchola ukončiť okamžite, ako zistíme, že jeho aktuálna hodnota je príliš zlá (príliš dobrá).

Príklad takéhoto oseknutia prehľadávania ukazuje obrázok 3.2. Prvý potomok vrchola x typu MAX má vypočítanú hodnotu 7. Druhý potomok je vrchol y typu MIN, ktorého prvý potomok má hodnotu 5. Y teda bude mať hodnotu najviac 5, a pri maximalizovaní vo vrchole x určite nebude hodnota y vybratá. Preto ďalších potomkov vrchola y už nemusíme prehľadávať.



Obrázok 3.2: Alfa-beta oseknutie prehľadávaného stromu.

Realizovať budeme alfa-beta osekávacie prehľadávanie tak, že v každom vrchole si budeme udržiavať premenné $alfa$ a $beta$. $Beta$ predstavuje horné ohraničenie pre hodnotu vrchola typu MAX. $Alfa$ predstavuje dolné ohraničenie pre hodnotu vrchola typu MIN. Interval $(alfa, beta)$ budeme nazývať *prehľadávacie okno*. Na začiatku prehľadávania budú v počiatočnom vrchole mať $alfa$ a $beta$ hodnoty $-\infty$ a $+\infty$. Ako budeme postupne vyhodnocovať vrcholy, budeme okno *zužovať* a prehľadávanie *osekávať* podľa nasledujúcich pravidiel:

Nech $w \in \delta(v)$, a val je práve vypočítaná hodnota w.

1. Ak $t(v) = MAX$ potom:
   - ak $alfa < val < beta$ potom $alfa := val$ (zúženie prehľadávacieho okna).
   - ak $val \geq beta$ potom ukonči prehľadávanie potomkov a vyhodnoť v (oseknutie prehľadávania).
2. Ak $t(v) = MIN$ potom:
   - ak $alfa < val < beta$ potom $beta := val$ (zúženie prehľadávacieho okna).

- ak $val \leq alfa$ potom ukonči prehľadávanie potomkov a vyhodnoť v (oseknutie prehľadávania).

Nasleduje minimaxový algoritmus rozšírený o alfa-beta osekávanie:

Algoritmus: Alfa-beta osekávací algoritmus
Vstup: hra $\Gamma = \{I, X, s, T, \delta, t, u\}$, heuristická evaluačná funkcia $h : X \to R$, stav $v \in X$, hĺbka d, $d \geq 0$, $alfa = -\infty$, $beta = +\infty$

```
procedure alfabeta(v, d, alfa, beta)
begin
   if d = 0 or v ∈ T then
      return h(v)
   else begin
      nech δ(v) = {y_1, y_2, ..., y_k}
      for i := 1 to k do begin
         v_i := alfabeta(y_i, d − 1, alfa, beta)
         if t(v) = MAX then begin
            if alfa < v_i < beta then
               alfa := v_i
            if v_i ≥ beta then
               return max{v_1, v_2, ..., v_i}
         end else begin
            if alfa < v_i < beta then
               beta := v_i
            if v_i ≤ alfa then
               return min{v_1, v_2, ..., v_i}
         end
      end

      if t(v) = MAX then
         return max{v_1, v_2, ..., v_k}
      else
         return min{v_1, v_2, ..., v_k}
   end
end
```

Analýza zložitosti alfa-beta prehľadávacieho algoritmu

Na rozdiel od úplného minimaxového prehľadávania je alfa-beta osekávacie prehľadávanie citlivé na poradie vyhodnocovania vrcholov. Pokiaľ vo vrchole v môže nastať oseknutie, tak pri *perfektnom poradí vyhodnocovania potomkov* to zistíme

hneď po vyhodnotení prvého potomka. Pri najhoršom poradí to zistíme až pri vyhodnocovaní posledného potomka a tým pádom nič neušetríme.

Definícia 3.3.1 *Nech* $\Gamma = \{ I, X, s, T, \delta, t, u \}$ *je hra,* $G = (V, E, \Gamma)$ *jej strom a* F *jej minimaxová funkcia.*
Hovoríme, že prehľadávací algoritmus A **vyhodnocuje potomkov vrcholov** *herného stromu* G **v perfektnom poradí**, *pokiaľ pre ľubovoľný vrchol* $x \in X$, $\delta(x) = \{y_1, y_2, \ldots, y_k\}$, *platí:*

- *ak* $t(x) = MAX$ *a* A *vyhodnotí vrcholy* $y_1, y_2, \ldots, y_k$ *v poradí* $y_{i_1}, y_{i_2}, \ldots, y_{i_k}$, *potom musí platiť* $F(y_{i_1}) \geq F(y_{i_2}) \geq \ldots \geq F(y_{i_k})$.
- *ak* $t(x) = MIN$ *a* A *vyhodnotí vrcholy* $y_1, y_2, \ldots, y_k$ *v poradí* $y_{j_1}, y_{j_2}, \ldots, y_{j_k}$, *potom musí platiť* $F(y_{j_1}) \leq F(y_{j_2}) \leq \ldots \leq F(y_{j_k})$.

Veta 3.3.1 *Nech je daný b/d-uniformný herný strom. Alfa-beta prehľadávací algoritmus navštívi v najlepšom prípade* $b^{\lfloor d/2 \rfloor} + b^{\lceil d/2 \rceil} - 1$ *listov a v najhoršom prípade* b^d *listov.*

Knuth a Moore dokázali túto vetu v roku 1975 ([KM75]). Ukázali, že najlepší prípad je dosiahnutý, keď sú potomkovia vrcholov herného stromu vyhodnocovaní v perfektnom poradí. Taktiež ukázali, že ľubovoľný algoritmus pre určenie presnej minimaxovej hodnoty koreňa stromu musí navštíviť aspoň toľko listov herného stromu ako alfa-beta prehľadávací algoritmus v najlepšom prípade.

Experimentálne výsledky hovoria, že aj v priemernom prípade je zlepšenie získané alfa-beta osekávaním obrovské. Častokrát nám to umožní v rovnakom čase prehľadávať herný strom až do takmer dvojnásobnej hĺbky. Pritom za toto zlepšenie neplatíme žiadnu daň - potrebné úpravy minimaxového algoritmu sú minimálne, pamäťové nároky zostávajú rovnaké a osekávame skutočne iba tie časti stromu, ktoré nemôžu mať žiadny vplyv na počítaný výsledok.

Ďalej uvádzame porovnanie časov behu úplného minimaxového algoritmu a alfa-beta algoritmu na vybraných herných stavoch hry dáma. Zvolená hĺbka prehľadávania bola 10. (Ďalšie podrobnosti o podmienkach a parametroch merania je možné nájsť v kapitole 6.)

	minimaxový algoritmus		alfa-beta algoritmus		
pozícia	**čas**	**# vrcholov**	**čas**	**# vrcholov**	**zrýchlenie**
1	24	7209264	1	95163	24
2	35	9048553	1	105243	35
3	88	22703295	3	236265	29.33
4	117	30509175	2	245057	58.5
5	29	7786004	1	59479	29
6	64	17670907	1	67778	64
7	184	49773302	2	162753	92
8	31	8593315	0	46207	-
9	30	7481422	2	122748	15
10	49	13063412	2	136667	24.5
11	171	46022100	2	194897	85.5
12	90	23371298	2	119930	45

Tabuľka 3.1: Zrýchlenie alfa-beta algoritmu vzhľadom na minimaxový algoritmus.

3.4 Iteratívne prehlbovanie

Iteratívne prehlbovanie rieši problém časovo obmedzeného prehľadávania herného stromu. Časové obmedzenie prehľadávania je bežné v turnajových podmienkach, kde je daný presný limit pre vykonanie ťahu. Možnosť stanoviť maximálnu dobu výpočtu je taktiež jednou zo základných funkcií, ktoré by mal mať užívateľsky prívetivý šachový či ľubovoľný iný herný program.

Keďže nevieme dopredu povedať, ktoré časti stromu budú pri alfa-beta prehľadávaní oseknuté, koľko ich bude, či aké budú veľké, nevieme ani odhadnúť celkový čas potrebný na prehľadanie stromu do zvolenej hĺbky. A opačne, ak máme daný maximálny čas, ktorý môžeme stráviť prehľadávaním, nedokážeme dopredu určiť maximálnu hĺbku, do ktorej si môžeme dovoliť prehľadávať bez jeho prekročenia.

Iteratívne prehlbovanie tento problém rieši tak, že postupne prehľadáva do hĺbok 1, 2, 3, a tak ďalej, kým nevyprší čas. Výhodou teda je, že pre ľubovoľné časové ohraničenie vždy budeme mať skončené prehľadávanie do nejakej hĺbky (za predpokladu, že v tom čase stihneme vykonať prehľadávanie aspoň do hĺbky 1). Nevýhodou je, že zbytočne opakujeme prehľadávanie vo vrchných častiach stromu.

Počet vrcholov vyhodnotených pri iteratívnom prehľadávaní do hĺbky n je však prinajhoršom iba 2-krát väčší, než počet vrcholov vyhodnotených pri priamom prehľadávaní do hĺbky n, ak predpokladáme b/d-uniformný strom s $b \geq 2$. Vyplýva to z nasledujúcich dvoch jednoduchých liem (obidve sa dajú dokázať matematickou indukciou vzhľadom na n):

Lema 3.4.1 *Ak* $b \geq 2$, *potom* $\sum_{i=0}^{n} b^i \leq b^{n+1}$.

Lema 3.4.2 *Ak* $b \geq 2$, *potom* $\sum_{j=0}^{n-1} \sum_{i=0}^{j} b^i \leq \sum_{i=0}^{n} b^i$.

Iteratívne prehlbovanie môžeme vylepšiť tak, že výsledky z predošlých iterácií si zapamätáme a použijeme ich na preusporiadanie potomkov vrcholov pred ich vyhodnotením v nasledujúcich iteráciách. Častokrát totiž platí, že vrchol, ktorý je „dobrý“ v hĺbke d, je „dobrý“ aj v hĺbke $d + 1$. A tak výsledky z predošlých iterácií nám často umožnia vykonať alfa-beta oseknutie skôr, než by sme to inak urobili a dosiahnúť tak počet vyhodnotených vrcholov porovnateľný s počtom vyhodnoteným pri priamom prehľadávaní do rovnakej hĺbky. Ba dokonca môže byť iteratívne prehlbované prehľadávanie aj rýchlejšie.

3.5 Transpozičná tabuľka

V sekcii 2.1 sme spomínali, že veľkosť herného stromu pre hru šach je okolo 10^{154}. Ale iba asi 10^{40} z týchto vrcholov je unikátnych, zvyšných 10^{114} vrcholov sú ich opakované výskyty. Tieto opakovania sú spôsobené skutočnosťou, že do určitej hernej pozície sa vieme častokrát dostať rôznymi postupnosťami (transpozíciami) ťahov. Keby sa nám podarilo vyhnúť sa opakovanému počítaniu opakovane sa vyskytujúcich vrcholov, môžeme očakávať výrazný nárast výkonu algoritmu.

Najlepšie by bolo, keby sme mohli mať celý prehľadávací priestor uložený v pamäti v podobe grafu tak, aby sa v ňom žiaden vrchol nevyskytoval viackrát. To si ale dovoliť nemôžeme, pretože aj graf bez opakovaných výskytov vrcholov je stále príliš veľký, než aby sa nám zmestil do pamäte. Kompromisným riešením je udržiavať si tabuľku vrcholov pevne danej veľkosti, ktorú budeme ďalej nazývať *transpozičná tabuľka*.

Je dobré si uvedomiť, ako dôležitá je efektívna implementácia transpozičnej tabuľky pre výkon algoritmu. Vždy pred spustením rekurzívneho vyhodnocovania vrchola sa budeme do tabuľky pozerať, či už tam hodnotu tohto vrchola nemáme uloženú. Pri jednom behu herného programu vyhodnotíme často až niekoľko stoviek miliónov vrcholov herného stromu, čo predstavuje niekoľko stoviek miliónov operácii vyhľadania prvku v transpozičnej tabuľke. Akákoľvek závažnejšia neefektívnosť pri návrhu či implementácii tabuľky sa pri takomto náročnom nasadení môže okamžite výrazne prejaviť.

Nie je teda nijak prekvapujúce, že na realizáciu transpozičnej tabuľky sa zvyčajne používa hašovanie. Pri hrách ako sú šach, dáma, othello a podobne vieme veľmi ľahko vypočítať hašovací kľúč štandardnými technikami. Stačí, ak hernú plochu zoberieme ako n-ticu (resp. reťazec) čísel a hašovací kľúč vypočítame z nej. Keďže hašovacia funkcia samozrejme obyčajne nie je injektívna, musíme pri

vyhľadaní prvku presne porovnať hľadaný prvok s príslušným prvkom uloženým v tabuľke. Riešenie, pri ktorom by sme mali v tabuľke uloženú celý herný stav (celú n-ticu čísel), je príliš priestorovo i časovo náročné. Preto sa uprednostňujú riešenia, pri ktorých najprv určíme číselný kľúč pre herný stav a iba tento číselný kľúč ukladáme do tabuľky a používame na porovnávanie pri vyhľadávaní. Index do tabuľky vrcholov budeme rátať tiež z tohto kľúča.

Prvok hašovacej transpozičnej tabuľky je 5-tica (k, d, v, f, m), kde:

- k je kľúč identifikujúci hernú pozíciu,
- d predstavuje hĺbku, do ktorej sme podstrom začínajúci v tejto pozícii prepočítali,
- v je hodnota najlepšieho ťahu z tejto pozície,
- f je príznak určujúci, či ide o presnú hodnotu, alebo iba o alfa či beta ohraničenie a
- m je samotný najlepší ťah.

Najlepší ťah m ukladať nemusíme, ak nás nezaujíma celá vybratá cesta do terminálneho vrchola, ale iba jej počiatočný vrchol. Položka d je veľmi dôležitá, pretože hodnotu z tabuľky môžeme použiť iba vtedy, keď hĺbka d je aspoň taká ako aktuálna hĺbka, v ktorej sme narazili na opakovaný výskyt vrchola. A napokon príznak f potrebujeme preto, lebo pokiaľ vo vrchole nastalo alfa-beta oseknutie, nemali sme vypočítanú jeho presnú hodnotu, ale iba horné či dolné ohraničenie. V tomto prípade hodnotu z tabuľky môžeme použiť na zúženie prehľadávacieho okna, ale nie ako presnú hodnotu.

Ako kľúč na identifikáciu hernej pozície budeme používať tzv. *Zobristove kľúče*. Túto techniku hašovania predstavil A. Zobrist v roku 1990 ([Zob90]). Zobristov kľúč je (obvykle aspoň 64-bitové) číslo, ktoré - pre hru s figúrkami na šachovnici - vypočítame nasledujúcim algoritmom:

1. Najprv si vytvoríme 3-rozmernú tabuľku T náhodných (obvykle aspoň 64-bitových) čísel. Prvou dimenziou bude typ figúrky, druhou farba figúrky a treťou číslo pozície na šachovnici, na ktorej sa figúrka nachádza (teda napríklad $T[dáma][čierna][B3]$).
2. Pre počiatočný stav najprv položíme Zobristov kľúč rovný 0. Potom pre každú figúrku na šachovnici urobíme XOR doteraz vypočítaného kľúča s číslom z tabuľky, určeným typom, farbou a pozíciou tejto figúrky.
3. Pre ľubovoľný iný stav, do ktorého sa dostaneme vykonaním nejakého ťahu, vieme vypočítať Zobristov kľúč efektívnejšie. Stačí, aby sme vykonali zodpovedajúce XORy tak, aby sme zachytili zmenu na šachovnici spôsobenú vykonaním daného ťahu.

> Napr. ak chceme zachytiť ťah „pohni bieleho pešiaka z $E5$ na $E6$“, tak stačí vykonať XOR aktuálneho kľúča s náhodným číslom $T[pešiak][biely][E5]$ a s číslom $T[pešiak][biely][E6]$. Prvý XOR odstráni v kóde zo šachovnice bieleho pešiaka z pozície $E5$ a druhý ho umiestni na pozícii $E6$.

Je zrejmé, že takéto kódovanie nie je úplne spoľahlivé. V prvom rade 2^{64} je iba približne $1,8 * 10^{19}$, a unikátnych herných pozícií je často viac (pri šachu spomínaných 10^{40}). Ale aj bez ohľadu na to z charakteru fungovania procedúry priraďovania kľúča vyplýva, že existuje určitá pravdepodobnosť, že dvom rôznym herným pozíciám bude priradený ten istý kľúč. Avšak ako Zobrist ukázal ([Zob90]), táto pravdepodobnosť je veľmi malá a v praktických herných programoch akceptovateľná.

Z dôvodu rýchlosti taktiež hašovacie transpozičné tabuľky obvykle riešia kolízie len jednoduchým nahradzovaním prvkov. Dve základné nahradzovacie stratégie predstavujú „nahraď vždy“ a „nahraď ak je hĺbka väčšia“. Stratégia „nahraď vždy“ uprednostňuje ukladanie nedávno vypočítaných vrcholov. Stratégia „nahraď ak je hĺbka väčšia“ uprednostňuje „cennejšie“ vrcholy (vypočítané do väčších hĺbok). Dajú sa vymyslieť aj ďalšie stratégie, ktoré môžu kombinovať aj viacero prístupov. Podrobnú analýzu rôznych nahradzovacích stratégii môžno nájsť v [Bre98].

Kapitola 4

Paralelné prehľadávanie herného stromu

Existujú dva základné spôsoby rozdeľovania práce procesorom pri paralelnom vyhodnocovaní herného stromu - prideľovanie častí prehľadávacieho okna a prideľovanie častí herného stromu.

Rozdelenie prehľadávacieho okna na disjunktné intervaly a ich pridelenie rôznym procesorom sa ukázalo ako výkonnostne veľmi obmedzená metóda. Bez ohľadu na počet použitých procesorov bolo dosiahnuté *zrýchlenie* obmedzené približne na faktor 5 ([MP84]). Preto paralelizácie majúce ambíciu dosahovať dobrý výkon na vysokom počte procesorov, prideľujú procesorom na výpočet rôzne časti herného stromu. Ale ani pri tomto prístupe nie je dosiahnutie dobrého výkonu vôbec jednoduché.

4.1 Miery efektívnosti paralelných algoritmov

Pri empirickom vyhodnocovaní efektívnosti paralelných algoritmov sa zvyknú tradične používať dve miery - *zrýchlenie* (speedup) a *efektivita*.

Definícia 4.1.1 *Pojmom* **zrýchlenie** *dosiahnuté paralelným algoritmom s p procesormi budeme označovať pomer:*

$$S_p(n) = \frac{T^*(n)}{T_p(n)}$$

kde $T^*(n)$ je čas behu optimálneho sekvenčného algoritmu na vstupe veľkosti n, a $T_p(n)$ je čas behu paralelného algoritmu používajúceho p procesorov na vstupe veľkosti n. [1]

[1] Namiesto optimálneho sekvenčného algoritmu sa často používa ten najlepší, ktorý je k dispozícii.

Definícia 4.1.2 Efektivitou *paralelného algoritmu budeme nazývať pomer:*

$$E(n) = \frac{S_p(n)}{p}$$

kde p je počet procesorov a $S_p(n)$ zrýchlenie dosiahnuté paralelným algoritmom na p procesoroch.

Pri počítaní zrýchlenia predpokladáme, že uvažovaný paralelný systém je homogénny, zložený z rovnakých procesorov. Pokiaľ by niektoré z procesorov boli výkonnejšie ako iné, vypočítané hodnoty zrýchlenia by nemali žiadnu vypovedaciu hodnotu.

Typicky je rast zrýchlenia v závislosti od počtu procesorov pomalší než lineárny. Spôsobené to býva tým, že spolupracujúce procesory strávia nejaký čas komunikovaním a synchronizovaním a nie samotným výpočtom. Problém sa taktiež niekedy nedá rozdeliť na úplne disjunktné časti a procesory môžu vykonávať duplicitnú prácu. S narastajúcim počtom procesorov sú tieto problémy čoraz výraznejšie a prínos získaný pridaním každého ďalšieho procesora je čoraz menší. Častým javom býva, že od určitého počtu procesorov je prínos získaný pridávaním ďalších procesorov zanedbateľne malý, ba dokonca môže dôjsť až k spomaleniu.

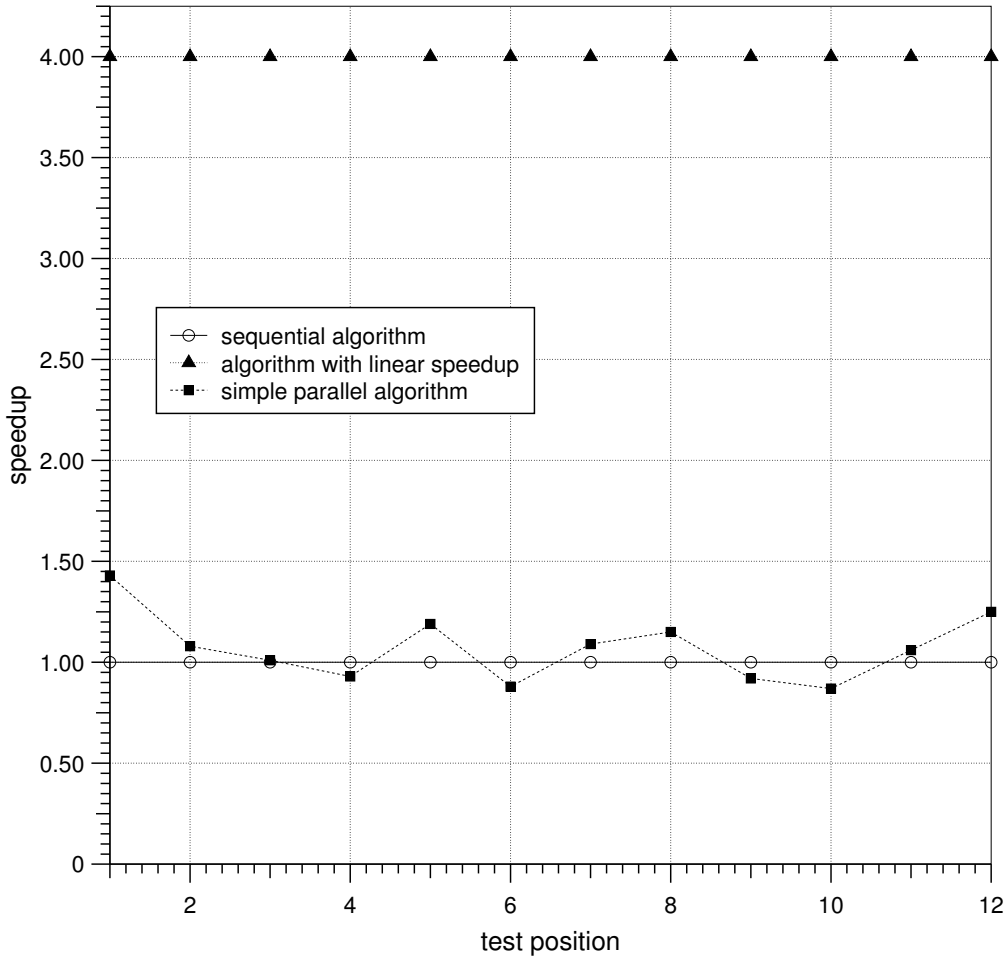
V niektorých prípadoch však môže nastať i väčšie než lineárne (tzv. *superlineárne*) zrýchlenie. Existujú teoretické analýzy i experimentálne testy, ktoré tento jav vysvetľujú a podkladajú. Príklady i zdôvodnenia, ako aj stručný prehľad literatúry možno nájsť v [Gus90].

Našim cieľom pri navrhovaní a implementovaní paralelných algoritmov teda bude získať čo najlepšie zrýchlenie – čiže také, ktoré ako funkcia od počtu procesorov rastie čo najprudšie.

4.2 Problémy pri paralelizácii prehľadávania

Priamočiare paralelizácie prehľadávacích algoritmov založených na alfa-beta orezávaní a jeho vylepšeniach dosahujú iba veľmi nízke zrýchlenie. Na obrázku 4.1 je znázornené zrýchlenie základného paralelného algoritmu vzhľadom na čas behu sekvenčného algoritmu v systéme so štyrmi procesormi a na 12 testovacích pozíciách. Priemerné namerané zrýchlenie je 1.07, a možno ho teda hodnotiť ako silne neuspokojivé.

Graf zároveň ukazuje aj zrýchlenie hypotetického paralelného algoritmu, ktorý na 4 procesoroch dosahuje na každej testovacej pozícii zrýchlenie 4. (Podrobnejší popis základného paralelného algoritmu a testovacieho prostredia možno nájsť v kapitole 6.)



Obr. 4.1: Zrýchlenie základného paralelného algoritmu.

Hlavným dôvodom nízkeho zrýchlenia sú nasledujúce tri problémy.

4.2.1 Problém rozdeľovania práce a vyrovnávania záťaže

Pri alfa-beta prehľadávaní sú veľkosti jednotlivých podstromov rôzne a dopredu neodhadnuteľné. Statické pridelenie jedného podstromu každému procesoru vedie k nízkej efektivite využitia procesorov – niektoré môžu počítať veľmi dlho (keď sa ich pridelený podstrom ukáže ako veľmi veľký), kým iné môžu byť hotové s výpočtom veľmi rýchlo a ich výkon bude ďalej zbytočne nevyužitý.

Teda je vhodné nejakým spôsobom zabezpečovať vyrovnávanie záťaže (load-balancing) tak, aby pokiaľ možno nedochádzalo k žiadnemu plytvaniu výkonu procesorov. Dynamické rozdeľovanie záťaže ale prináša určité režijné náklady – na prerozdeľovanie práce musia procesory vykonať ďalšiu komunikáciu a aj výpočtom samotného prerozdeľovania strávia určitý čas.

4.2.2 Problém vykonávania zbytočnej práce

Alfa-beta osekávanie je závislé od informácií získaných pri výpočte v predošlých (ľavých) častiach stromu. Pri paralelnom výpočte však jednotlivé podstromy môžu byť počítané súbežne a informácia vedúca k oseknutiu podstromu nemusí byť dostupná. Môže byť teda počítaných veľa podstromov, ktoré by sekvenčný algoritmus vôbec nepočítal. Toto vedie k množstvu zbytočných výpočtov, a preto aj nízkemu zrýchleniu a efektivite.

4.2.3 Problém distribuovanej transpozičnej tabuľky

Efektívna realizácia globálnej hašovacej transpozičnej tabuľky v distribuovaných systémoch, kde môžu procesory komunikovať iba výmenou správ, je problematická. Dotazy do tabuliek spravovaných inými procesormi môžu znamenať neúnosné spomalenie. Na druhej strane použitie čisto lokálnych tabuliek prináša zvýšenie zbytočnej práce, pretože každý procesor má vo svojej tabuľke uložené iba vrcholy z malej časti celého herného stromu, a opäť dochádza k výpočtom, ktoré by sekvenčný algoritmus nevykonával.

Kapitola 5

Návrh pokročilého paralelného algoritmu

Pri paralelných výpočtoch často platí, že existuje viacero protichodných činiteľov, ktoré treba pri návrhu algoritmov vhodne vyvážiť. V našom prípade to nie je inak. Hlavnými dvoma protichodnými faktormi sú pre nás *zbytočná práca* a *komunikačná réžia*.

Napríklad ak zvolíme pevné prideľovanie podstromov procesorom, ušetríme na komunikácii, ale zvýši sa nám množstvo zbytočnej práce. Ak umožníme procesorom bez obmedzenia nahliadať do transpozičných tabuliek ostatných procesorov, ušetríme si zbytočné počítanie niektorých podstromov, ale výrazne nám stúpne množstvo komunikácie.

Pri návrhu algoritmu sa teda budeme snažiť nájsť dobré vyváženie uvedených dvoch faktorov. Pri tom je asi nevyhné, aby sme dopredu prijali určité predpoklady o architektúre cieľového paralelného systému. Je to dané tým, že na rôznych hardvérových architektúrach často bývajú rôzne riešenia odlišne výkonné. Napr. pri architektúre so zdieľanou pamäťou si môžeme dovoliť zdieľanú globálnu hašovaciu tabuľku a tá pre nás nemusí znamenať podstatné zvýšenie réžie, ale pri distribuovanej architektúre bez zdieľanej pamäte môže takéto riešenie byť úplne neprijateľné.

Ak chceme, aby náš algoritmus bol skutočne efektívny, nevyhneme sa tiež tomu, aby sme nejakým spôsobom nevyriešili všetky tri hlavné problémy spomenuté v predošlej kapitole – musíme vhodným spôsobom zabezpečiť rozdeľovanie práce a vyrovnávanie záťaže, minimalizáciu zbytočnej práce i realizáciu globálnej transpozičnej tabuľky.

5.1 Cieľová architektúra

Pri návrhu nášho pokročilého paralelného algoritmu budeme uvažovať architektúru *distribuovaných procesorov komunikujúcich prostredníctvom komunikačných*

kanálov metódou posielania správ. Ďalej predpokladáme, že každý procesor je priamo prepojený s každým ďalším – sieť spojení má teda *topológiu úplného grafu*. A napokon predpokladáme, že posielanie správy je – v porovnaní s čítaním z lokálnej pamäte alebo vykonaním inštrukcie procesora – *veľmi drahá operácia*.

Popísaná architektúra dostatočne verne modeluje náš cieľový hardvér, ktorým je množina osobných počítačov prepojených lokálnou počítačovou sieťou.

5.2 Rozdeľovanie práce a vyrovnávanie záťaže

Naše riešenie vychádza z klasického rozdelenia procesov na *master proces* a *worker procesy*. Master proces dostane zadaný herný stav. Vygeneruje si úplný herný strom do zvolenej hĺbky, a jeho listy bude vhodným spôsobom prideľovať svojim workerom. Bude sa pritom snažiť neprideľovať žiadne listy zbytočne. Pri prijatí výsledku od workera otestuje, či nie je možné vrámci svojho vygenerovaného stromu vykonať alfa-beta oseknutie. Prijatý výsledok tiež použije na aktualizáciu hodnôt alfa a beta v príslušnom podstrome. Keď zistí, že niektorý z listov stromu bol pridelený workerovi zbytočne, okamžite mu pošle príkaz na prerušenie výpočtu.

Architektúra s jedným masterom a veľa workermi má podstatné výhody – je jednoduchá a komunikačne úsporná. Hlavnou nevýhodou je, že prináša *úzke hrdlo* (*bottleneck*) v podobe jediného master procesu, ktorý vďaka obmedzenému výkonu procesora či prípadne obmedzenej kapacite komunikačných kanálov dokáže riadiť činnosť iba obmedzeného počtu workerov.

Pre menšie počty procesorov (a teda aj worker procesov) to nemusí predstavovať problém. Avšak keď počet procesorov prekročí masterovu kapacitu, problémom sa to takmer určite stane. Našťastie to ale vieme vyriešiť vytvorením *hierarchie master procesov*.

Pokiaľ jeden master proces zvládne obsluhovať k workerov, ale workerov je podstatne viac, môžeme nad workermi vybudovať strom masterov, pričom master na úrovni n bude obsluhovať k masterov na úrovni $n-1$, ku ktorým sa bude správať ako k obyčajným workerom. Masteri na úrovni 0 budú potom pracovať so samotnými workermi.

Zvýšením počtu master procesov samozrejme zvyšujeme percento procesorov, ktoré sa venujú riadeniu výpočtu, a nie riešeniu samotného problému. Nech jeden master zvládne obsluhovať maximálne k procesov (masterov nižšej úrovne alebo workerov) a nech je celkovo $k^{i-1} \leq m \leq k^i$ worker procesov. Potom na riadenie týchto procesov treba $\sum_{j=0}^{i-1} k^j$ masterov. Pre niektoré vhodné hodnoty k a i môžeme túto réžiu považovať za prijateľnú (napr. pre $k = 10$ a $i = 4$ – teda 10000 workerov – by bolo treba 1111 masterov, čo je z celkového počtu 11111 procesov približne 10%).

5.3 Minimalizácia zbytočnej práce

Alfa-beta osekávací algoritmus využíva informáciu vypočítanú v predošlých častiach stromu na určenie prehľadávacieho okna pre nasledujúce časti stromu. Od aktuálnej veľkosti okna závisí, aká veľká časť aktuálneho podstromu bude prehľadaná. Veľkosti prehľadaných podstromov teda vo všeobecnosti nevieme dopredu odhadnúť. Vieme však vypočítať, ako vyzerá minimálny podstrom herného stromu, ktorý musí alfa-beta osekávací algoritmus nutne prezrieť.

Definícia 5.3.1 Minimálnym podstromom *herného stromu nazveme taký jeho podstrom, ktorý bude prehľadaný sekvenčným alfa-beta algoritmom v prípade perfektného poradia vyhodnocovania vrcholov.*

Knuth a Moore na vypočítanie minimálneho podstromu navrhli ([KM75]) nasledujúce čiastočne definované zobrazenie M, $M : X \to \{1, 2, 3\}$:

Definícia 5.3.2 *Nech Ord je funkcia, $Ord : X' \times X' \to N$, $X' \subseteq X$, priraďujúca potomkom vrcholov poradové číslo vyhodnotenia pri prehľadávaní herného stromu prehľadávacím algoritmom A (napríklad $Ord(i, j) = 3$, ak j je z potomkov vrchola i prehľadaný ako tretí v poradí).*

Funkciu M zadefinujeme nasledovne:

$$M(x) = \begin{cases} 1 & ak\ x = s,\ alebo \\ & ak\ x \in \delta(y),\ y \in X,\ M(y) = 1\ a\ Ord(y, x) = 1\ ; \\ 2 & ak\ x \in \delta(y),\ y \in X,\ M(y) = 1\ a\ Ord(y, x) > 1,\ alebo \\ & ak\ x \in \delta(y),\ y \in X,\ M(y) = 3; \\ 3 & ak\ x \in \delta(y),\ y \in X,\ M(y) = 2\ a\ Ord(y, x) = 1\ . \end{cases}$$

Knuth a Moore ďalej dokázali ([KM75]) tento výsledok:

Lema 5.3.1 *Funkcia M je pri perfektnom poradí vyhodnocovania vrcholov definovaná práve pre tie vrcholy b/d-uniformného herného stromu, ktoré patria do jeho minimálneho podstromu.*

Menej formálne (a možno trochu viac zrozumiteľne) môžeme zobrazenie M popísať tiež nasledovne:

1. Koreň stromu je typu 1.
2. Prvý potomok vrchola typu 1 má typ 1, ostatní majú typ 2.
3. Prvý potomok vrchola typu 2 má typ 3.
4. Všetci potomkovia vrchola typu 3 majú typ 2.

Hsu rozšíril definíciu Knutha a Moora ([Hsu90]) tak, aby bola definovaná pre všetky stavy hry:

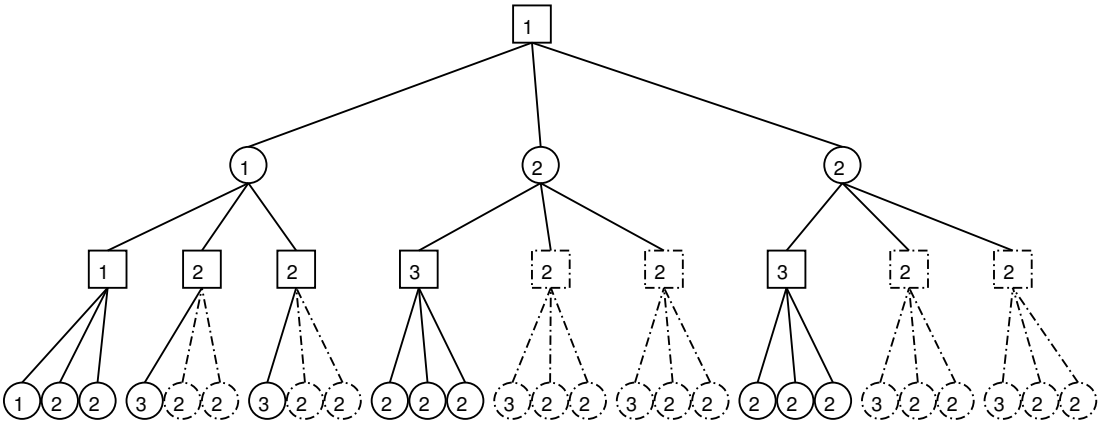
$M(x) = 2$, ak $x \in \delta(y)$, $y \in X$, $M(y) = 2$ a $Ord(y, x) > 1$

respektíve:

5. Ostatní potomkovia vrchola typu 2 majú typ 2.

Na obrázku 5.1 je znázornený hypotetický herný strom, ktorého vrcholy majú priradený typ podľa funkcie Knutha, Moora a Hsua. Plnou čiarou sú zaznačené vrcholy minimálneho podstromu, prerušovanou čiarou tie vrcholy, ktoré do minimálneho podstromu nepatria.

Z takto rozšírenej definície vyplýva veľmi dôležité pozorovanie – pri prehľadávaní stromu s perfektným poradím vyhodnocovania sú vždy prehľadaní všetci potomkovia vrcholov typu 1 a vrcholov typu 3, ale iba prvý potomok vrchola typu 2. Vrcholy typu 2 sú teda tými vrcholmi, v ktorých nastáva oseknutie.



Obrázok 5.1: Strom s vrcholmi s priradenými typmi podľa funkcie Knutha, Moora a Hsua. Plnou čiarou sú vyznačené vrcholy minimálneho podstromu, prerušovanou čiarou vrcholy, ktoré do minimálneho podstromu nepatria.

Pokiaľ poradie vyhodnocovania nebude perfektné, môžu byť prehľadaní aj ďalší potomkovia vrchola typu 2, v najhoršom prípade všetci. Teda ak chceme minimalizovať množstvo zbytočnej práce pri paralelnom prehľadávaní, je vhodné riadiť sa nasledujúcimi pravidlami:

- Potomkovia vrchola typu 1 alebo 3 budú prehľadaní okamžite a paralelne.
- Potomkovia vrchola typu 2 budú prehľadávaní postupne, zľava-doprava.

Pre potreby nášho algoritmu navrhneme trochu odlišnú funkciu. Táto bude vychádzať z funkcie Knutha, Moora, a Hsua, ale bude zadefinovaná tak, aby prirodzene korenšpondovala so spôsobom rozdeľovania práce worker procesom, ktorý sme popísali v časti 5.2.

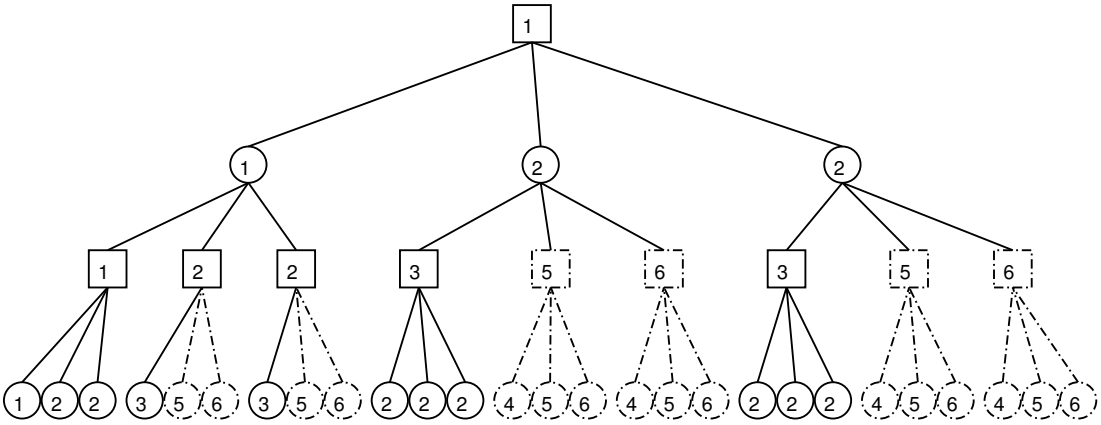
Funkciu $M' : X \to N$ zadefinujeme nasledovne:

Definícia 5.3.3 *Nech Ord je funkcia, $Ord : X' \times X' \to N$, $X' \subseteq X$, priraďujúca potomkom vrcholov poradové číslo vyhodnotenia pri prehľadávaní herného stromu prehľadávacím algoritmom.*

Funkciu M' zadefinujeme takto:

$$M'(x) = \begin{cases} 1 & ak\ x = s,\ alebo \\ & ak\ x \in \delta(y),\ y \in X,\ M(y) = 1\ a\ Ord(y,x) = 1; \\ 2 & ak\ x \in \delta(y),\ y \in X,\ M(y) = 1\ a\ Ord(y,x) > 1,\ alebo \\ & ak\ x \in \delta(y),\ y \in X,\ M(y) = 3; \\ 3 & ak\ x \in \delta(y),\ y \in X,\ M(y) = 2\ a\ Ord(y,x) = 1; \\ 3{+}i & ak\ x \in \delta(y),\ y \in X,\ M(y) = 2\ a\ Ord(y,x) = i > 1,\ alebo \\ & ak\ x \in \delta(y),\ y \in X,\ M(y) > 3\ a\ Ord(y,x) = i\ . \end{cases}$$

Obrázok 5.2 znázorňuje hypotetický herný strom s typmi priradenými podľa našej funkcie.



Obrázok 5.2: Herný strom s typmi vrcholov priradenými podľa našej funkcie M'.

Pripomeňme, že naše riešenie paralelného rozdeľovania práce funguje tak, že master proces si vygeneruje úplný herný strom do zvolenej hĺbky a potom jeho listy prideľuje na prehľadávanie svojim worker procesom. S ohľadom na vyššie uvedenú funkciu M' prideľujúcu typy vrcholom potom môžeme zadefinovať jednoduché pravidlá pre určenie poradia, v akom budú listy prideľované:

- Voľnému worker procesu priraď ešte nepriradený list s čo najmenším typom.

- V prípade viacerých nepriradených listov s rovnakým typom priraď ten najviac vľavo.

Algoritmus prideľovania listov podľa vyššie uvedených pravidiel bude fungovať tak, že najprv pridelí list typu 1, potom bude prideľovať listy typu 2. Všetky tieto listy patria do minimálneho podstromu, a teda ich prednostné pridelenie pre paralelné vyhodnotenie nepredstavuje žiadnu potenciálne zbytočnú prácu. Po vyčerpaní listov typu 2 budeme prideľovať listy typu 3 a viac. Tu je treba si všimnúť, že typy sú pridelené listom takým spôsobom, že potomkovia vrchola s typom 2 alebo typom väčším ako 3 sú prehľadávaní postupne, zľava doprava. Teda namiesto toho, aby sme v nich prehľadali veľa potomkov jedného vrchola naraz, budeme prehľadávať po jednom potomkovi z veľa vrcholov naraz.[1]

Tento spôsob nám dovolí dosiahnúť pomerne nízke percento zbytočne pridelenej práce, ale úplne ju eliminovať nedokáže. Pokiaľ totiž oseknutie nastane niekde na vyššej úrovni v masterovom strome, všetky zostávajúce procesory pridelené listom ležiacom v zodpovedajúcom podstrome boli pridelené zbytočne.

Je ešte namieste poznamenať, že potomkov vrchola s typom 2 alebo typom väčším než 3 neprehľadávame v takom poradí, v akom boli vygenerovaní funkciou na generovanie ťahov. Namiesto toho ich triedime podľa údajov z transpozičnej tabuľky, podobne ako sme to popisovali pri sekvenčnom iteratívne prehlbovacom algoritme.

5.4 Realizácia distribuovanej transpozičnej tabuľky

Návrh distribuovanej transpozičnej tabuľky je pri architektúre bez zdieľanej pamäte dôležitý a pomerne komplikovaný problém. Na jednej strane potrebujeme mať tabuľku čo „najbohatšiu“ (obsahujúcu čo najviac vrcholov z rôznych podstromov herného stromu), na strane druhej musíme udržať čas potrebný na vyhľadanie záznamu v tabuľke čo najnižší.

Existujú tri základné spôsoby realizácie distribuovanej transpozičnej tabuľky:

1. **globálna distribuovaná tabuľka**: Existuje jedna veľká „logická“ tabuľka, ktorá je fyzicky rozdelená na bloky uložené v rôznych uzloch systému. Každý procesor môže pristupovať k svojmu bloku, ako aj k blokom uloženým v ostatných uzloch.

2. **centralizovaná tabuľka**: Tabuľka nie je fyzicky rozdelená, namiesto toho je celá umiestnená v jednom vyhradenom uzle. Tento vybavuje požiadavky ostatných uzlov na záznamy z tabuľky.

[1] Ak máme viac procesorov ako vrcholov, budeme prehľadávať aj viac potomkov jedného vrchola naraz.

3. **lokálne tabuľky**: Každý uzol má svoju vlastnú tabuľku a pri prehľadávaní pristupuje iba do nej.

Každý z uvedených prístupov má svoje klady a zápory. Globálna distribuovaná tabuľka je veľmi bohatá, porovnateľná s tabuľkou sekvenčného programu, ale vyžaduje buď zdieľanú pamäť, alebo veľké množstvo správ. Navyše je každý procesor zaťažený okrem svojho vlastného prehľadávania aj odpovedaním na dotazy ostatných uzlov.

Centralizovaná tabuľka opäť vyžaduje veľa komunikácie, ale znižuje záťaž ostatných procesorov, ktoré sa môžu venovať iba samotnému prehľadávaniu. Zároveň však uzol s centralizovanou tabuľkou predstavuje úzke hrdlo systému.

Riešenie s lokálnymi tabuľkami prináša vynikajúce nízke komunikačné náklady, avšak tabuľky sú dosť „chudobné“ a procesory vykonávajú veľa zbytočnej práce počítaním podstromov, ktoré by systémy s „bohatou“ tabuľkou nepočítali.

V našom algoritme budeme používať **hybridné riešenie**, ktoré vychádza z riešenia s lokálnymi tabuľkami, ale zároveň sa snaží prekonať problém „chudobnosti“ zavedením určitej vhodnej výmeny údajov medzi tabuľkami.

Riešenie vychádza z (odôvodneného) predpokladu, že výmena údajov medzi tabuľkami jednotlivých procesorov počas výpočtu („v reálnom čase“) je pri uvažovanej architektúre príliš nákladná. Avšak určitú výmenu údajov zaviesť musíme, pretože riešenie úplne bez výmeny, s čisto lokálnymi tabuľkami, nie je z dôvodu veľkého množstva zbytočnej práce dostatočne efektívne. To nám potvrdili i vykonané merania (viac v kapitole 6).

Preto rozdelíme paralelný výpočet na *výpočtové fázy*, v ktorých bude vykonávané prehľadávanie, a *synchronizačné fázy*, v ktorých bude vykonávaná výmena údajov z tabuliek medzi procesormi. Tieto fázy budeme opakovane striedať.

Keďže algoritmus, ktorý paralelizujeme, je iteratívny, vieme doňho popísané rozdelenie na fázy prirodzeným spôsobom začleniť. Každá iterácia paralelného prehľadávacieho algoritmu bude pre nás predstavovať výpočtovú fázu. Synchronizačná fáza bude vykonaná vždy po skončení každej výpočtovej fázy – čiže jednej iterácie.

Ďalej treba vhodne vyriešiť problémy, ktoré údaje budeme synchronizovať, aké množstvo údajov je vhodné posielať a akým spôsobom (protokolom) budú procesy výmenu realizovať.

Keďže v transpozičných tabuľkách používame nahradzovaciu stratégiu „nahraď, ak je hĺbka väčšia nanajvýš rovná“, prirodzeným riešením otázky „Ktoré údaje synchronizovať?“ je vyberať si práve tie vrcholy z tabuliek, ktoré boli vypočítané do najväčších hĺbok. Je to vhodná voľba, pretože tieto vrcholy obyčajne prinesú (v prípade ich úspešného vyhľadania v tabuľke pri ich opakovanom výskyte) najväčšiu úsporu zbytočnej práce.[2] Aby sme ale v synchronizačných fázach

[2]Za predpokladu, že hĺbka stromu je dobrým ukazovateľom veľkosti stromu.

zbytočne neposielali stále tie isté vrcholy, budeme pri vyberaní vrcholov určených na posielanie ostatným procesorom ďalej uprednostňovať tie vrcholy, ktoré boli zmenené v poslednej výpočtovej fáze.

Protokol výmeny údajov medzi procesmi je v našom riešení centralizovaný. Najprv si každý worker proces vytvorí výber tých „najlepších“ vrcholov (tj. prehľadaných do najväčšej hĺbky a pokiaľ možno zmenených v poslednej iterácii) z tabuľky zvolenej veľkosti. Tieto výbery pošlú workeri svojmu masterovi. Master ich vhodným spôsobom spracuje a uloží do svojej tabuľky. Z tej vytvorí vlastný výber toho „najlepšieho“ a tento spätne pošle všetkým workerom, ktorí si ním aktualizujú údaje vo svojich tabuľkách. Tým bude synchronizačna fáza ukončená a bude zahájená ďalšia výpočtová fáza.

Zvolený spôsob spracovania výberov od workerov v masterovi je jednoduchý a účinný – master vrcholy z výberov vkladá do svojej tabuľky rovnako, ako ich ukladá pri prehľadávaní stromu. Rovnako je používaná aj nahradzovacia stratégia „nahraď ak je hĺbka väčšia nanajvýš rovná“. A výber z tabuľky mastera je vytváraný rovnako, ako výbery z tabuliek workerov – čiže uprednostňované sú vrcholy prepočítané do najväčších hĺbok a zmenené v poslednej iterácii.[3]

Tento centralizovaný protokol je – podobne ako náš centralizovaný spôsob rozdeľovania práce – komunikačne úsporný, ale s potenciálnym úzkym hrdlom. Komunikovaných bude (v jednej synchronizačnej fáze) len $2 * k$ správ (výberov), kde k je počet workerov, oproti $O(k^2)$ správam (výberom), ak by workeri robili výmenu decentralizovane systémom „každý s každým“. A na druhej strane spracovanie a zlučovanie výberov v master procese môže byť pre veľký počet procesorov časovo náročné – a kým ho master vykonáva, ostatné procesory zostávajú nevyužité.

Podobne ako pri rozdeľovaní práce však možno použiť hierarchiu masterov, v ktorej by každý najprv spracoval k výberov od svojich workerov alebo masterov nižšej úrovne a vzápätí poslal výsledok masterovi na vyššej úrovni. Master na najvyššej úrovni by potom spätne poslal nový výber svojim podriadeným masterom, a tak by sa postupne výber dostal až k workerom.

Aké množstvo údajov je vhodné posielať sme určili experimentálne. Podľa očakávania posielanie veľmi veľkého množstva údajov (v krajnom prípade celých tabuliek) je neefektívne. Výhodnejšie je posielanie iba pomerne malej časti tabuľky. Ako vhodnú hodnotu sme určili pre tabuľky veľkosti 200003 prvkov hodnotu 1000.

5.5 Algoritmus

V tejto kapitole uvedieme vysokoúrovňový popis algoritmu pre worker aj pre master procesy.

[3] V našej implementácii master aj worker používajú na vytvorenie výberu z tabuľky tú istú funkciu.

5.5.1 Worker proces

Algoritmus: worker proces

1. Prijími príkaz P od svojho mastera.
2. Ak P = „skonči“, potom ukonči počítanie.
3. Ak P = „pošli výber z tabuľky“, potom vytvor výber z tabuľky a pošli masterovi.
4. Ak P = „prijmi výber z tabuľky“, potom prijmi od mastera výber a aktualizuj si podľa neho svoju tabuľku.
5. Ak P = „nastav si novú hĺbku prehľadávania“, tak prijmi a nastav si novú hĺbku prehľadávania.
6. Ak P = „prijmi nový vrchol na počítanie“, tak prijmi kód vrchola a začni na ňom vykonávať sekvenčné alfa-beta usekávacie prehľadávanie za použitia svojej lokálnej tabuľky. Po skončení pošli výsledok masterovi.
7. Ak P = „preruš výpočet“, tak preruš výpočet.
8. Choď na krok 1.

5.5.2 Master proces

Algoritmus: master proces
Vstup: D_0, D_{max}, D_{pre}

1. Sekvenčne vykonaj iteratívne prehlbované prehľadávanie až po zvolenú hraničnú hĺbku D_0 použitím alfa-beta osekávacieho algoritmu a svojej lokálnej transpozičnej tabuľky.
2. Pre každú vyššiu iteráciu s hĺbkou i, $D_0 < i \leq D_{max}$ rob:
3. Vypočítaj výber zo svojej tabuľky a pošli ho svojim workerom (príkaz „prijmi výber z tabuľky“).
4. Pošli svojím workerom príkaz „nastav si novú hĺbku prehľadávania“ s parametrom i.
5. Vygeneruj si úplný prehľadávací strom do pevne zvolenej hĺbky D_{pre}. Pritom označkuj generované vrcholy podľa heuristiky popísanej v časti 5.3.
6. Ulož si listy vygenerovaného stromu do poľa A a označ ich ako „nepridelené“.

7. Utrieď pole A vzostupne podľa typov vrcholov určených heuristickou funkciou.

8. Prideľuj listy z pola A worker procesom za použitia procedúry *manage_work*.

9. Prijími výsledky všetkých dobiehajúcich výpočtov (pomocou procedúry *receive_work*).

10. Pokiaľ $i < D_{max}$ (čiže i nie je posledná iterácia), prijmi od svojich worker procesov výbery z ich lokálnych tabuliek (príkaz „pošli výber z tabuľky“) a aktualizuj si nimi svoju tabuľku. Zvýš si hodnotu premennej i a pokračuj v ďalšej iterácii (choď na krok 2).

Algoritmus: procedúra *manage_work*

1. Kým existujú listy poľa A, ktoré ešte neboli pridelené na výpočet, vypočítané, alebo označené ako „mŕtve“ (tj. bolo zistené, že sa nachádzajú v podstrome, ktorý bol oseknutý), potom rob:

2. Kým existujú workeri, ktorí nemajú pridelenú žiadnu robotu, vykonávaj procedúru *send_work*.

3. Ak majú všetci workeri pridelenú robotu, čakaj na výsledok od niektorého workera (vykonaj procedúru *receive_work*).

4. Choď na krok 1.

Algoritmus: procedúra *send_work*

1. Vyber si ľubovoľného workera, ktorý nemá pridelenú prácu (je označený ako „voľný“).

2. Vyber si prvý list z utriedeného poľa A taký, že nie je označený ani ako „pridelený“, ani ako „vypočítaný“, ani ako „mŕtvy“.

3. Pošli vybranému workerovi kód vybraného listu a označ list ako „pridelený“ a workera ako „zaneprázdneného“.

Algoritmus: procedúra *receive_work*

1. Prijími výsledok počítania od niektorého workera w.

2. Nastav hodnotu pre zodpovedajúci list a označ ho ako „vypočítaný“.

3. Pokiaľ list má v strome rodiča, over či je v rodičovi možné vykonať oseknutie, pomocou procedúry *check_pruning* s argumentami *rodič listu*, *hodnota listu*.

4. Označ workera w ako „voľného“.

Algoritmus: procedúra *check_pruning*
Parametre: vrchol v, hodnota h

1. Nech typ v je MAX. Ak $alfa < h < beta$, tak nastav $alfa := h$. Ak $h > beta$, potom vykonaj oseknutie – označ všetkých potomkov, ktorí sú ešte počítaní alebo ešte nie sú pridelení, a taktiež všetky vrcholy v podstromoch, ktorých sú títo potomkovia koreňmi, ako „mŕtvych“. Pošli všetkým workerom, ktorí počítajú nejaký z listov, ktoré boli práve označené ako „mŕtve“ správu „preruš výpočet“.

2. Nech typ v je MIN. Ak $alfa < h < beta$, tak nastav $beta := h$. Ak $h < alfa$, potom vykonaj oseknutie podobne ako v kroku 1.

3. Ak si v kroku 1 alebo 2 vykonal zúženie prehľadávacieho okna, tak vykonaj aktualizáciu prehľadávacieho okna u všetkých potomkov vrchola v, pomocou procedúry *set_alpha_beta*.

4. Ak nastalo oseknutie, alebo už je vypočítaná hodnota všetkých potomkov uzla v, potom vypočítaj hodnotu pre v (maximum z hodnôt potomkov, ak je v typu MAX, minimum ak je v MIN). Pokiaľ v nie je koreň stromu a teda má rodiča, tak rekurzívne zavolaj procedúru *check_pruning* s argumentami *rodič v* a *hodnota v*.

Algoritmus: procedúra *set_alpha_beta*
Parametre: vrchol v, hodnoty a a b

1. Nech typ v je MAX. Potom ak $alfa < b < beta$, tak nastav $beta := b$. Ak $b \leq alfa$, tak vykonaj oseknutie vo v rovnako ako v kroku 1 procedúry *check_pruning*.

2. Nech typ v je MIN. Potom ak $alfa < a < beta$, tak nastav $alfa := a$. Ak $a \geq beta$, tak vykonaj oseknutie vo v rovnako ako v kroku 1 procedúry *check_pruning*.

3. Ak si zúžil v kroku 1 alebo 2 prehľadávacie okno, zúž okno aj všetkým potomkom v zavolaním procedúry *set_alpha_beta* s parametrami *potomok v*, *alfa hodnota v*, *beta hodnota v*.

V procedúre *set_alpha_beta* stojí za pozornosť, že osekávanie vykonávame inak, než v štandardnom alfa-beta osekávacom algoritme. Vo vrchole v typu MAX nastáva pri sekvenčnom prehľadávaní oseknutie, ak hodnota práve vypočítaného potomka je väčšia ako beta hranica. V tomto prípade sme však dostali „skutočnú“ hodnotu beta hranice pre v oneskorene a „spätne“ ideme overovať, či nemalo nastať oseknutie.

Keďže sme počítali s menej presnou hodnotou beta (v najhoršom prípade s $beta = +\infty$), mohlo sa stať, že niektorá z vypočítaných hodnôt potomkov nespôsobila oseknutie, hoci pri sekvenčnom prehľadávaní by to urobila. Maximálna takáto hodnota je pritom uložená v $alfa$ vo v. Čiže pri získaní nového (presnejšieho) odhadu pre *beta* vo v treba správne porovnávať, či $alfa \geq$ *nová beta* (ako je to robené v procedúre *set_alpha_beta*). Pre vrcholy typu MIN platia analogické vzťahy.

5.6 Implementácia

5.6.1 Programovací jazyk a prostredie

Zvoleným programovacím jazykom bol jazyk C++. Implementácia bola vykonaná za použitia knižnice PVM a v prostredí operačného systému LINUX. Okrem štandardných knižníc jazyka C a C++ (stdio, stdlib, time a math) a knižnice PVM už neboli použité žiadne ďalšie knižnice. Knižnica PVM bola zvolená kvôli jednoduchosti a rýchlosti použitia a vývoja a tiež dobrej dostupnosti na zvolenej platforme LINUX.

O systéme PVM

PVM (Parallel Virtual Machine) je softvérový systém umožňujúci efektívny vývoj a nasadenie paralelných programov v heterogénnom prostredí. Umožňuje, aby množina rôznorodých prepojených počítačov vystupovala ako jeden veľký virtuálny počítač. PVM transparentne zabezpečuje smerovanie správ, konverziu dát medzi nekompatibilnými architektúrami i riadenie behu úloh.

Paralelný program v prostredí PVM je zložený z množiny úloh (procesov) bežiacich vo virtuálnom PVM počítači a komunikujúcich prostredníctvom posielania správ. Fyzicky môže viacero PVM úloh bežať na jednom počítači alebo na viacerých počítačoch. Užívateľ má možnosť presne špecifikovať, kde budú jednotlivé úlohy spustené, alebo môže prenechať zodpovednosť za umiestňovanie úloh systému PVM.

PVM systém sa skladá z dvoch hlavných častí – PVM démona (pvmd) a programátorskej knižnice funkcií. PVM démon je program, ktorý beží na každom z použitých počítačov a zabezpečuje všetky vyššie uvedené nevyhnutné kontrolné funkcie – spúšťa a riadi procesy, vykonáva konverziu dát a najmä posielanie, prijímanie a smerovanie správ. Programátorská knižnica obsahuje bohatú sadu primitív potrebných na spoluprácu paralelne bežiacich procesov. Tieto zahŕňajú predovšetkým posielanie a prijímanie správ, spúšťanie ďalších procesov a nastavovanie parametrov virtuálneho PVM počítača.

Vyčerpávajúci popis architektúry PVM systému, histórie jeho vzniku, podrobný zoznam funkcií PVM knižnice, príklady paralelných programov i mnoho ďalších informácií je možné nájsť v PVM manuále ([GB+94]).

5.6.2 Zvolená hra

Ako cieľová hra bola zvolená hra dáma. Je na to niekoľko dôvodov:

1. Aj keď je herný strom (resp. graf) hry dáma výrazne menší, než napr. hry šach (odhaduje sa okolo 10^{20} oproti 10^{40} pri šachu), stále je dostatočne veľký na to, aby bolo jeho prehľadávanie ťažkým problémom.

2. Prehľadávacie algoritmy sme sa snažili navrhovať tak, aby boli nezávislé od konkrétnej použitej hry.

3. V porovnaní s napr. šachom je pre dámu podstatne jednoduchšie zadefinovať funkciu na generovanie ťahov či evaluáciu pozícii. Pre väčšinu hier je to špecifický problém a vzhľadom na náš „herne-nezávislý“ prístup pre nás nebol zaujímavý.

Pri implementácii generátora ťahov boli použité oficiálne pravidlá variantu dámy „česká dáma“ podľa Českej federácie dámy [4].

5.6.3 Implementačné problémy a ich riešenia

Prerušenie výpočtu worker procesu

V predošlej kapitole sme spomínali, že worker proces vie prerušiť svoj výpočet, pokiaľ mu master na to pošle príkaz (keď zistí, že výpočet je vykonávaný zbytočne). Ako to realizovať je pomerne zaujímavý implementačný problém. Existujú dve základné riešenia:

1. worker proces bude pozostávať z dvoch threadov („vlákien“). Jeden thread sa bude venovať výpočtu a druhé bude kontrolovať, či od mastera nedošiel príkaz na prerušenie.

2. worker proces bude pozostávať iba z jedného threadu, ale vždy pri rekurzívnom zavolaní procedúry na prehľadávanie herného stromu overí, či od mastera nedošiel príkaz na prerušenie.

Kvôli jednoduchosti a aby sme sa vyhli komplikáciám vyplývajúcim zo skutočnosti, že knižnica PVM nie je navrhovaná pre multithreadové aplikácie, rozhodli sme sa vychádzať z riešenia 2. Avšak operácia na kontrolu, či od mastera nedošla príslušná správa (funkcia *pvm_probe* z PVM knižnice) je pomerne drahá. Preto ju nevykonávame pri každom rekurzívnom volaní prehľadávacej procedúry, ale iba pri každom k-tom.

Na základe testovania sme si ako vhodné k vybrali hodnotu 1000.

[4] *http://www.damweb.cz/pravidla/cdfull.html*

Evaluačná funkcia

Ako evaluačná funkcia bola použitá jednoduchá funkcia na počítanie materiálnej hodnoty figúrok na šachovnici. Prijali sme tu predpoklad, že evaluačná funkcia nemá na priebeh zrýchlenia paralelného algoritmu rozhodujúci vplyv. A keďže nás zaujíma práve zrýchlenie (a nie kvalita hrania), dovolili sme si použiť jednoduchú funkciu.

Umiestňovanie procesov na procesory

Pri worker procesoch sme sa riadili pravidlom „jeden procesor – jeden worker proces“. Spúšťanie viacerých worker procesov na jednom procesore by vďaka virtuálnosti PVM prostredia bolo možné, ale asi by nemalo veľký význam. Skôr by mohlo dôjsť k spomaleniu z dôvodu zvýšenej réžie pri prepínaní procesov.

Pri master procese naopak možnosť púšťať viacero procesov na jednom procesore bola veľmi užitočná. Pri malom počte worker procesov totiž master väčšinu času strávi čakaním na výsledky. Pokiaľ by na danom procesore už okrem neho nebežal žiadny ďalší proces, procesor by zostal väčšinu času zbytočne nevyužitý.

Teda pri menších počtoch procesorov sme okrem master procesu púšťali na rovnakom procesore aj jeden worker proces. Na ostatných procesoroch sme potom púšťali po jednom worker procese.

Vykonané testy ukázali, že pre menej ako 5 procesorov sa neoplatí nechávať pre master proces vyhradený procesor. Pri 5 a viac procesoroch bol už master proces zaťažený dostatočne a mohol byť púšťaný na vyhradenom procesore.

Pri testoch sme teda v konfigurácii s 2, resp. 4, počítačmi nechali bežať 1 mastera a 2, resp. 4, worker procesy. Pri konfigurácií s 8 a 16 procesormi sme nechali bežať master proces na vyhradenom procesore a teda okrem neho bežalo iba 7 a 15 worker procesov.

5.6.4 Popis zdrojových súborov

Náš paralelný program pozostáva z nasledujúcich zdrojových súborov: `gts.h`, `gts.cc`, `hash_const.h`, `hash_item.cc`, `hash_table.cc`, `gts_dist.h`, `gts_master.cc`, `gts_slave.cc` a `send.cc`. Všetky možno nájsť na priloženom CD (viď príloha B).

Súbor `gts.cc` obsahuje implementáciu triedy *Game*, ktorá predstavuje stav hry. Zahŕňa metódy na generovanie ťahov, evaluačnú funkciu, implementácie rôznych algoritmov na prehľadávanie či generovanie herného stromu a mnohé ďalšie. Súbor `gts.h` obsahuje definície konštánt používaných v triede *Game*.

Súbory `hash_table.cc` a `hash_item.cc` obsahujú implementáciu hašovacej tabuľky založenej na Zobristových kľúčoch. Hašovaciu tabuľku predstavuje trieda *HashTable* s funkciami na vkladanie a vyhľadávanie prvkov a prácu so Zobristovými kľúčmi. Prvkami hašovacej tabuľky sú objekty triedy *HashItem*, implementovanej v príslušnom súbore `hash_item.cc`. `Hash_const.h` obsahuje definície

konštánt používaných v triede *HashTable*.

Worker proces je implementovaný v súbore `gts_slave.cc`. Kvôli jednoduchosti je implementovaný procedurálne a nie ako objekt a je presnou implementáciou algoritmu popísaného v časti 5.5.1.

Master proces je podobne ako worker proces implementovaný procedurálne. Je presnou implementáciou algoritmu popísaného v časti 5.5.2 a pozostáva z procedúr *main*, *manage_work*, *send_work*, *receive_work*, *check_narrow*, *set_alpha_beta*, *set_abort*, *send_table_updates*, *receive_table_updates*, *distribute_zobrist_keys*, *store_leaves*, *compare_leaves*, *sort_leaves* a ďalších pomocných procedúr. Súbor `gts_dist.h` obsahuje definície konštánt používaných v súboroch `gts_master.cc` a `gts_slave.cc`.

Kapitola 6

Merania

V tejto kapitole poskytneme výsledky meraní výkonu popísaných algoritmov. Merania sme zamerali dvoma smermi. Najprv sme porovnávali základný sekvenčný minimaxový algoritmus a jeho rôzne vylepšenia. Potom sme preskúmali zrýchlenie dosahované základným a našim pokročilým paralelným algoritmom, a napokon sme analyzovali, ako vplývajú jednotlivé pokročilé metódy v paralelnom algoritme na jeho celkový výkon.

Za účelom porovnávania vylepšení sekvenčných algoritmov zovšeobecníme definíciu zrýchlenia z kapitoly 4 nasledujúcim spôsobom.

Definícia 6.0.1 *Pojmom* **zrýchlenie** *dosiahnuté algoritmom A vzhľadom na algoritmus B budeme označovať pomer:*

$$S_{A,B}(n) = \frac{T_B(n)}{T_A(n)}$$

kde $T_B(n)$ je čas behu algoritmu B na vstupe veľkosti n, a $T_A(n)$ je čas behu algoritmu A na vstupe veľkosti n.

Merania sme vykonali na sade 12 herných pozícií, ktoré boli dostupné v literatúre ([Pla96]). Všetky použité pozície možno nájsť na CD prílohe.

6.1 Zrýchlenie sekvenčných algoritmov

Všetky testy sekvenčných algoritmov sme vykonali na počítači s nasledujúcou konfiguráciou:

- procesor: AMD Duron 1,3 GHz
- operačná pamäť: 512 MB
- operačný systém: Mandrake Linux 9.2.

Porovnanie minimaxového a alfa-beta algoritmu

V prvom teste sme porovnali úplné minimaxové prehľadávanie do hĺbky 10 s alfa-beta prehľadávaním do hĺbky 10. V súlade s teoretickými očakávaniami je alfa-beta prehľadávanie mnohonásobne rýchlejšie. V tabuľke 6.1 sú uvedené časy výpočtu, počty prehľadaných vrcholov a zrýchlenie pri výpočtoch na použitých 12 testovacích pozíciách.

	minimaxový algoritmus		alfa-beta algoritmus		
pozícia	**čas**	**# vrcholov**	**čas**	**# vrcholov**	**zrýchlenie**
1	24	7209264	1	95163	24
2	35	9048553	1	105243	35
3	88	22703295	3	236265	29.33
4	117	30509175	2	245057	58.5
5	29	7786004	1	59479	29
6	64	17670907	1	67778	64
7	184	49773302	2	162753	92
8	31	8593315	0	46207	-
9	30	7481422	2	122748	15
10	49	13063412	2	136667	24.5
11	171	46022100	2	194897	85.5
12	90	23371298	2	119930	45

Tabuľka 6.1: Zrýchlenie alfa-beta algoritmu vzhľadom na minimaxový algoritmus.

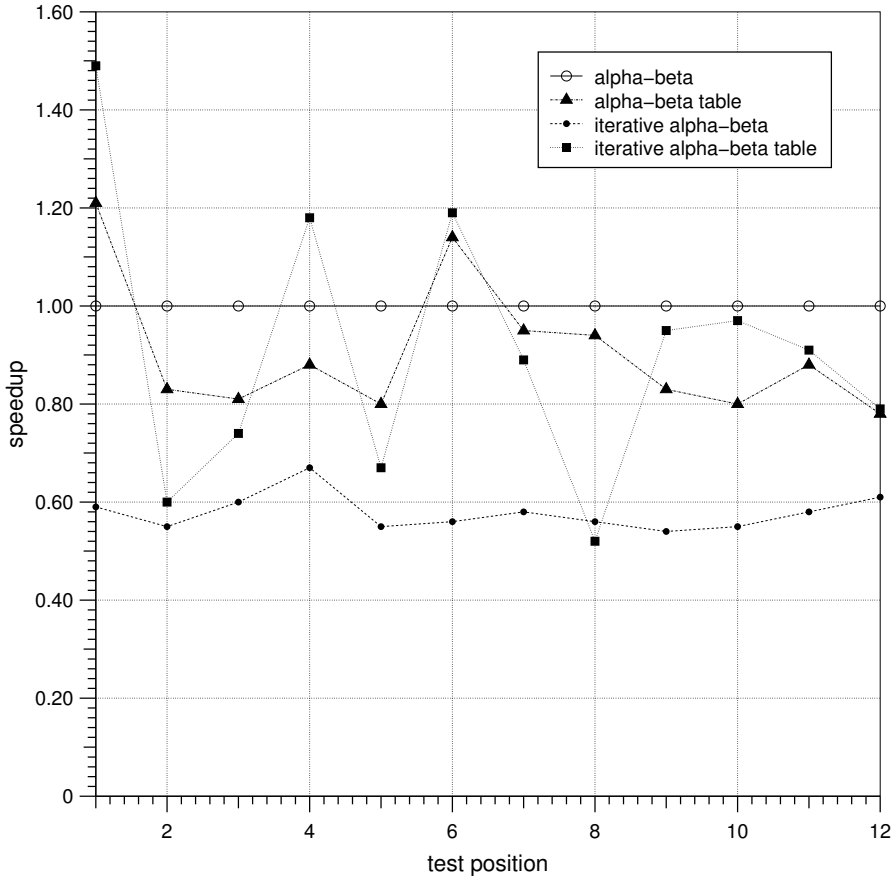
Porovnanie vylepšení alfa-beta algoritmu

V tomto teste porovnávame štyri verzie alfa-beta prehľadávacieho algoritmu: „čisté“ alfa-beta prehľadávanie, prehľadávanie využívajúce tabuľky, iteratívne prehlbovacie prehľadávanie a iteratívne prehlbovacie prehľadávanie využívajúce tabuľky. Všetky prehľadávania boli vykonané do hĺbky 14, transpozičné tabuľky mali veľkosť 100003 prvkov a použitá bola nahradzovacia stratégia „nahraď, ak je hĺbka väčšia nanajvýš rovná“.

Ako vidieť, algoritmy používajúce tabuľky prehľadávajú výrazne menej vrcholov, než ich bez-tabuľkové verzie. Avšak réžia práce s tabuľkou je taká veľká, že v niektorých prípadoch aj pri prehľadaní menšieho počtu vrcholov je čas behu algoritmu vyšší.

	alfa-beta		alfa-beta s tabuľkou		iteratívny alfa-beta		iteratívny alfa-beta s tabuľkou	
poz.	**čas**	**vrcholy**	**čas**	**vrcholy**	**čas**	**vrcholy**	**čas**	**vrcholy**
1	58	5748243	48	2602747	98	9596415	39	2223531
2	48	3942956	58	3100518	87	7262616	79	4438732
3	137	11836354	170	9613214	227	18971982	185	10526661
4	141	12342119	161	9119724	242	21009800	120	6839500
5	24	2389385	30	1889228	44	4242587	36	2213270
6	43	4033328	38	2252278	76	7013908	36	1988010
7	103	8549757	108	5680733	178	15400697	115	6306590
8	18	946731	19	680899	32	1893043	35	1345007
9	44	3164947	53	2530635	81	6445261	46	2224071
10	73	5260170	91	4279097	133	10491591	75	3777796
11	117	9344942	133	6800977	201	16058662	129	6517904
12	82	6677275	106	5619294	133	10635264	103	5258720

Tabuľka 6.2: Porovnanie vylepšení alfa-beta algoritmu.



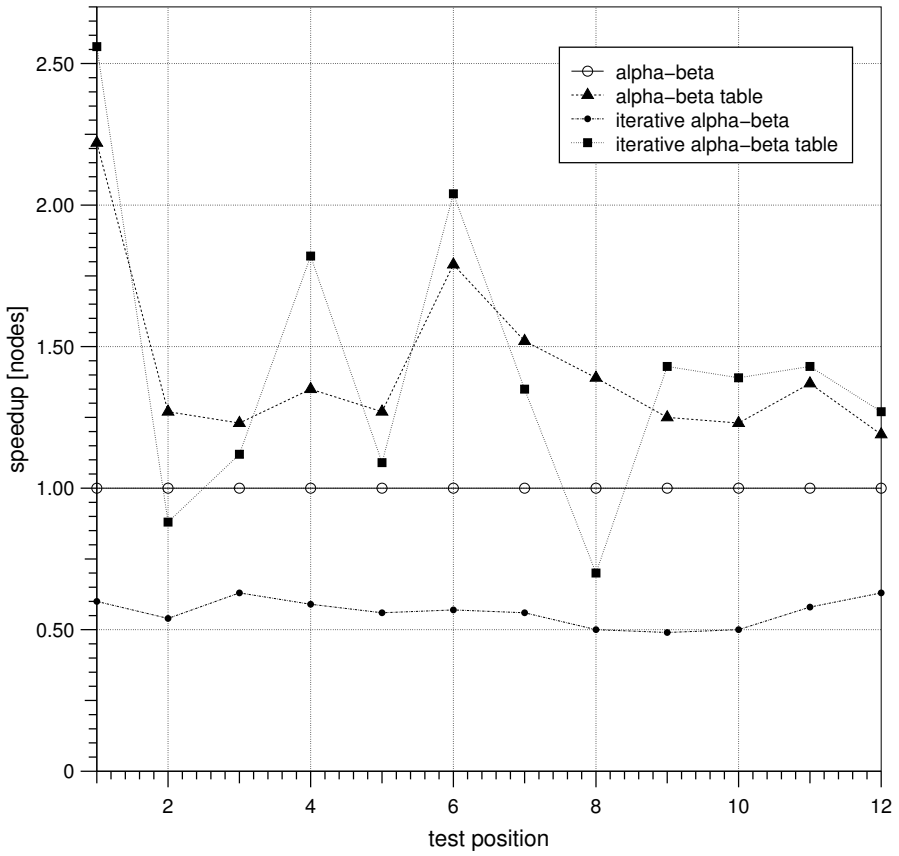
Obr. 6.1: Zrýchlenie vylepšení alfa-beta algoritmu.

algoritmus	**priemerné zrýchlenie**	**„vrcholové“ zrýchlenie**
alfa-beta s tabuľkou	0.90	1.42
iteratívny alfa-beta s tabuľkou	0.91	1.42

Tabuľka 6.3: Priemerné a „vrcholové“ zrýchlenia.

Graficky je porovnanie znázornené na obrázkoch 6.1 a 6.2. Na prvom grafe porovnávame zrýchlenia dosiahnuté algoritmami vzhľadom na základný alfa-beta algoritmus. V druhom grafe porovnávame „vrcholové“ zrýchlenia algoritmov - čiže zrýchlenia vypočítané nie na základe časov behov algoritmov, ale namiesto toho na základe počtu prehľadaných vrcholov. Teda zanedbávame tu čas, ktorý si vyžaduje práca s tabuľkou.

Vrcholové zrýchlenie takto predstavuje určitý odhad, aké zlepšenie zrýchlenia môžeme očakávať, keď budeme zlepšovať implementáciu transpozičnej tabuľky. (Avšak nemusí to byť horný odhad zlepšenia, ktoré je možné získať použitím tabuliek. Zrýchlenie totiž môže ešte výraznejšie narastať so zväčšovaním veľkosti transpozičnej tabuľky. Praktické testy a ich výsledky možno nájsť v [Bre98] alebo [Fel93].)



Obr. 6.2: „Vrcholové zrýchlenie“ vylepšení alfa-beta algoritmu.

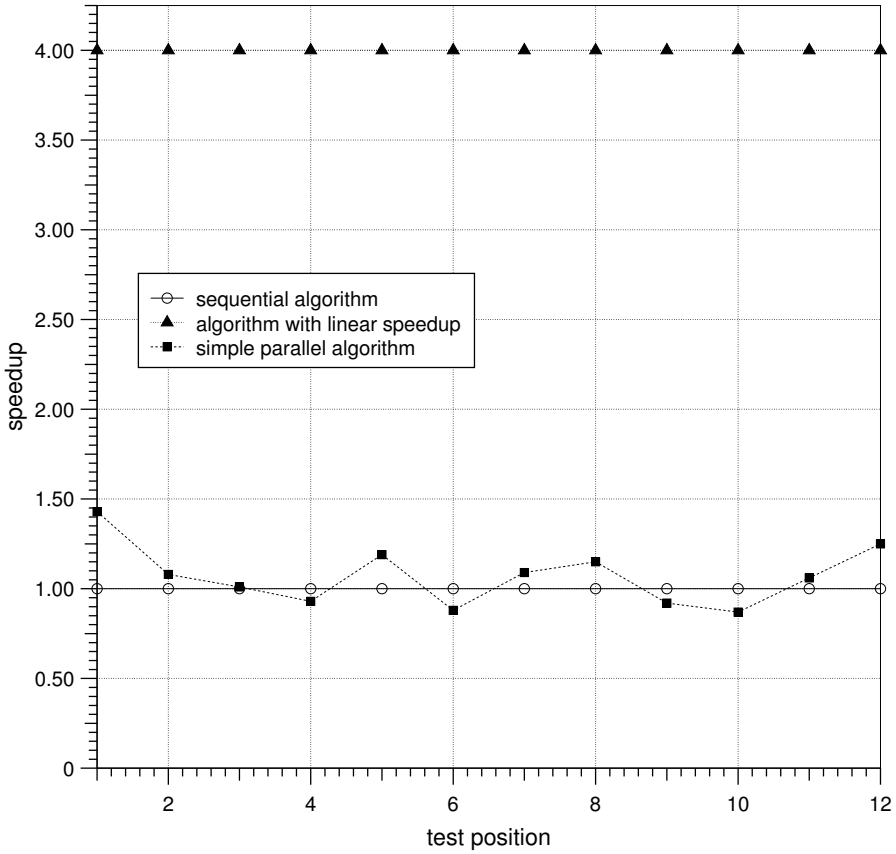
6.2 Zrýchlenie paralelných algoritmov

Všetky merania paralelných algoritmov boli vykonané na lokálnej počítačovej sieti, v ktorej bol každý počítač týchto parametrov:

- procesor: Intel Pentium 4 1,7 GHz
- operačná pamäť: 256 MB
- operačný systém: Debian Linux
- rýchlosť sieťového pripojenia: 100 Mbit/s

Zrýchlenie základného paralelného algoritmu

Jednoduchý základný algoritmus sme testovali pre demonštráciu, že dosiahnúť dobré zrýchlenie pri paralelizácii prehľadávania herného stromu nie je jednoduché.



Obr. 6.3: Zrýchlenie základného paralelného algoritmu (4 procesory).

Testovaný algoritmus fungoval nasledujúcim spôsobom:

1. Pre zadaný počiatočný stav v vygeneruj jeho potomkov $\delta(v) = \{w_1, w_2, \ldots, w_k\}$.
2. Každému procesoru priraď na výpočet jedného potomka w_i z $\delta(v)$ a vypočítanú hodnotu ulož do premennej h_i.
3. Ak existujú nevyhodnotené vrcholy z $\delta(v)$, choď na krok 2.
4. Ak $t(v) = MAX$, vráť $\max\{h_1, h_2, \ldots, h_k\}$. Inak vráť $\min\{h_1, h_2, \ldots, h_k\}$.

Algoritmus bol testovaný na konfigurácii so 4 počítačmi. Dosiahnuté zrýchlenie je na obrázku 6.3. (Krivka s trojuholníkovými bodmi predstavuje zrýchlenie hypotetického algoritmu, ktorý na každej testovacej pozícií dosahuje konštantné zrýchlenie 4.)

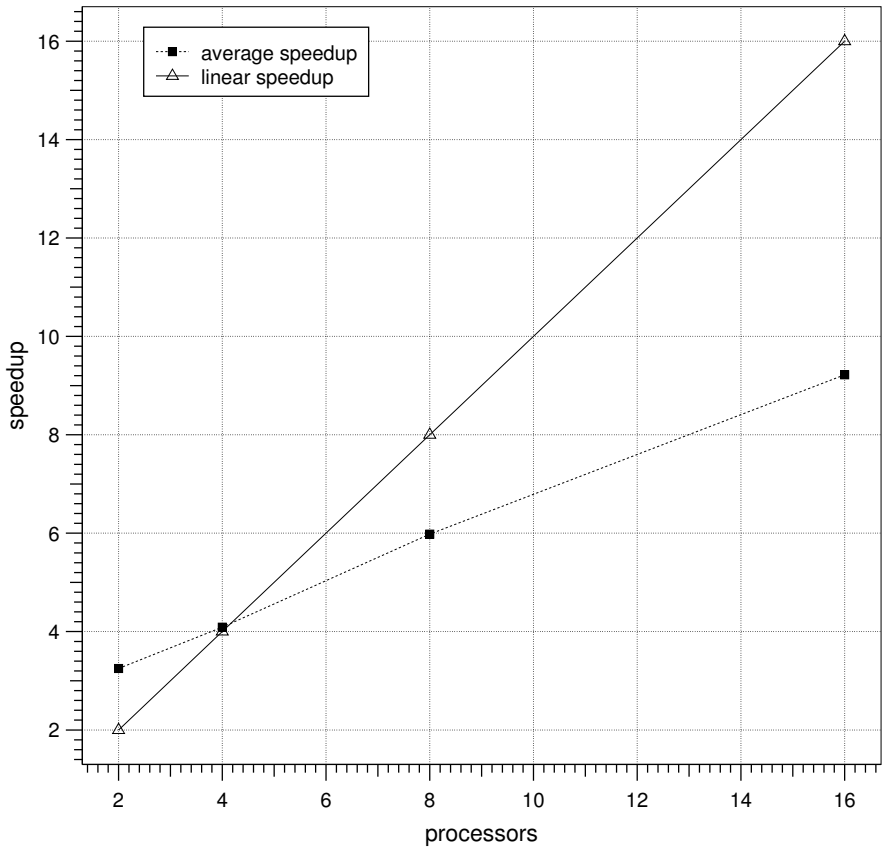
Priemerné zrýchlenie dosiahnuté so 4 procesormi je 1,07, a teda možno ho hodnotiť ako veľmi slabé a neuspokojivé.

Zrýchlenie pokročilého paralelného algoritmu

Testy zrýchlenia pokročilého algoritmu boli vykonávané s použitím 2, 4, 8 a 16 počítačov. Každé meranie bolo vykonané trikrát a pri počítaní zrýchlenia sme použili priemer časov výpočtu. Výsledné zrýchlenie je potom priemer zrýchlení na jednotlivých testovacích pozíciách.

Na uvedenom grafe asi okamžite zaujme superlineárne priemerné zrýchlenie pri konfigurácii s dvoma procesormi. Je to spôsobené hlavne asi skutočnosťou, že sme paralelnému programu dovolili, aby s rastúcim počtom procesov rástla aj celková sumárna veľkosť transpozičných tabuliek systému. Pri konfigurácii s dvoma procesormi sme nechali bežať celkovo tri procesy - jedného mastera a dvoch workerov (viď kapitola 5), a každý mal k dispozícii tabuľku o veľkosti 200003 prvkov. Sumárna veľkosť transpozičných tabuliek v systéme bola teda 600009 prvkov, pričom sekvenčný program mal k dispozícii iba tabuľku veľkosti 200003 prvkov.

Pri niektorých testovacích pozíciách bolo teda vďaka tomuto možné dosiahnúť aj vyššie zrýchlenie, než lineárne. Pri iných pozíciách bolo naopak zrýchlenie výrazne nižšie. (Graf závislosti zrýchlenia od testovacej pozície pri konfigurácii so 4 procesormi možno nájsť v nasledujúcej časti.) Tento výsledok zodpovedá aj pozorovaniu pri meraní zrýchlenia sekvenčných algoritmov - pri niektorých testovacích pozíciách je prínos získaný použitím transpozičných tabuliek vysoký, pri iných naopak veľmi nízky.



Obr. 6.4: Zrýchlenie pokročilého paralelného algoritmu.

Superlineárne zrýchlenie pozorovali aj Feldman, Mysliwietz a Monien ([FMM91] a [Fel93]). Ich program síce hral šach, používal odlišný spôsob prideľovania práce procesorom a úplne inak riešil aj distribuovanú transpozičnú tabuľku, ale tiež mal umožnené, aby s rastúcim počtom procesorov rástla aj celková sumárna veľkosť tabuliek. Feldman et al. takisto superlineárne zrýchlenie vyhodnotili ako dôsledok väčšej transpozičnej tabuľky.

Vyjadrené číselne sú dosiahnuté priemerné zrýchlenia takéto: pri dvoch procesoroch 3,25, pri štyroch procesoroch 4,09, pri ôsmich 5,98 a pri šestnástich 9,22.

Porovnanie vylepšení paralelného algoritmu

V tejto časti sa zameriame na to, akú veľkú zásluhu majú jednotlivé vylepšenia použité v pokročilom paralelnom algoritme na jeho výkon. Porovnávať budeme nasledujúce verzie algoritmu:

- algoritmus bez posielania výberov z tabuliek medzi master a worker procesmi (tj. ide vlastne o algoritmus používajúci lokálne tabuľky).
- algoritmus bez heuristiky na prideľovanie vrcholov procesorom, a

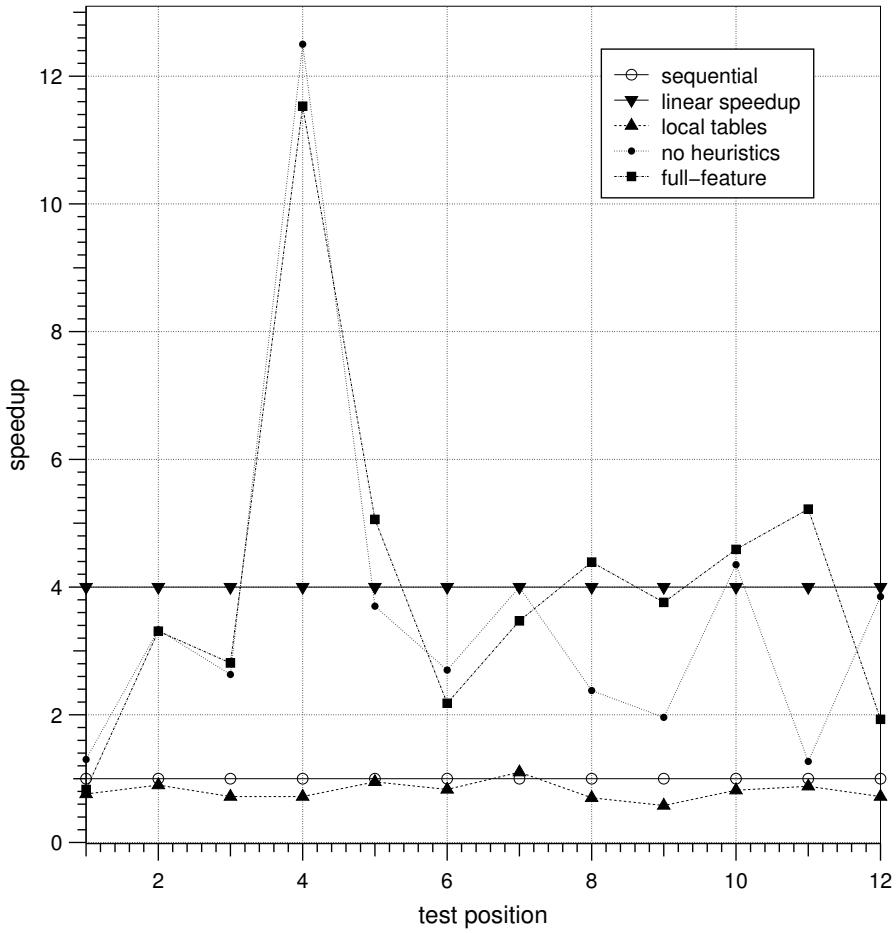
- „full-feature“ algoritmus (tj. náš pokročilý paralelný algoritmus so všetkými funkciami a vylepšeniami zapnutými).

Testy boli vykonané pri konfigurácií so 4 procesormi. Ich výsledky sú zachytené v grafe na obrázku 6.5.

Na grafe sú veľmi dobre vidieť oba krajné prípady spomínané v predošlej časti. Na testovacej pozícii č. 4 dokáže algoritmus až šokujúcim spôsobom využiť prítomnosť – v tomto prípade až 5-násobne – väčšej transpozičnej tabuľky, kdežto pri pozícii č. 1 to preňho neznamená žiadny prínos. Naopak, režijné náklady paralelného algoritmu pri nej úplne zatienia prínos vyplývajúci z použitia väčšieho počtu procesorov, a dosiahnuté zrýchlenie je menej než 1.

Ďalšie veľmi zaujímavé pozorovanie je, že zrýchlenie algoritmu s čisto lokálnymi tabuľkami je veľmi slabé. V priemernom prípade dosahuje iba 0.81 - čiže algoritmus beží na 4 procesoroch pomalšie ako sekvečný algoritmus na jednom. Tento výsledok dokladá, aký veľký prínos znamená použitie nášho hybridného spôsobu distribúcie transpozičnej tabuľky. Zároveň ukazuje, aký veľký vplyv môže mať transpozičná tabuľka pri paralelnom prehľadávaní herného stromu.

Priemerné zrýchlenie algoritmu bez heuristiky na prideľovanie vrcholov procesorom je 3,66. Priemerné zrýchlenie „full-feature“ algoritmu s heuristikou je 4,09. Naša heuristika teda nemá na zrýchlenie až taký výrazný vplyv ako hybridné riešenie distribuovanej tabuľky, ale nárast zrýchlenia je aj tak nezanedbateľný – 12%.



Obr. 6.5: Vplyv vylepšení paralelného algoritmu.

6.3 Zhodnotenie výsledkov merania

Z meraní sekvenčných algoritmov vyplýva, že alfa-beta osekávanie aj transpozičné tabuľky prinášajú veľkú úsporu množstva prehľadaných vrcholov. Pri použití transpozičných tabuliek je ale výsledná úspora času veľmi závislá od efektívnosti ich implementácie.

Testy paralelných algoritmov ukázali, že naše riešenie distribuovanej transpozičnej tabuľky i naša heuristika na minimalizovanie zbytočnej práce majú výrazný vplyv na zrýchlenie algoritmu. Výmena údajov medzi lokálnymi tabuľkami sa ukázala ako úplne kľúčová a algoritmus bez nej dosahuje iba veľmi slabé zrýchlenie.

Ďalším pozorovaným javom je, že zrýchlenie paralelného algoritmu je silne závislé od konkrétnej testovacej pozície. Vyplýva to aj z toho, že úspora získaná transpozičnými tabuľkami je rôzna pre jednotlivé testovacie pozície (teda rôzne podstromy herného stromu). Podobne je odlišná aj účinnosť použitej heuristiky na minimalizovanie zbytočnej práce.

Na testovanie bolo použitých 12 testovacích pozícií vybratých z literatúry.

Keďže aj pri opakovaných behoch boli dosahované časy behu odlišné (v niektorých prípadoch výrazne), bol test na každej pozícii vykonaný trikrát a pri výpočtoch bol použitý priemerný dosiahnutý čas. Odlišné výsledky sú tu spôsobené tým, že výpočty boli vykonávané na distribuovanom systéme a taktiež tým, že pri realizácii transpozičnej tabuľky bolo použité randomizované hašovanie.

Vzhľadom na popísanú premenlivosť výsledkov na rôznych testovacích pozíciách či dokonca aj pri opakovaných testoch na rovnakej testovacej pozícii, by bolo asi vhodné vykonať väčšie množstvo testov na väčšom množstve testovacích pozícií. Avšak vzhľadom na to, koľko času a úsilia by si takéto vyčerpávajúce testovanie vyžadovalo, jeho uskutočnenie pre nás nebolo realizovateľné.

Kapitola 7

Záver

Úspešne sme navrhli a implementovali pokročilý paralelný algoritmus pre prehľadávanie herného stromu. Navrhli sme hybridnú distribuovanú transpozičnú tabuľku s originálne riešenou výmenou údajov medzi lokálnymi tabuľkami. Uviedli sme mechanizmus pre centralizované rozdeľovanie práce procesorom a navrhli heuristickú funkciu pre minimalizovanie množstva zbytočne pridelenej práce. Uviedli sme aj spôsob, akým možno prekonať obmedzenie na počet procesorov v systéme vyplývajúce z centralizovaného riešenia.

Vďaka centralizácii riadenia, dobrej metóde minimalizácie zbytočne pridelenej práce a účinnému riešeniu distribuovanej transpozičnej tabuľky sme získali algoritmus, ktorý dosahuje uspokojivé zrýchlenie aj pri použití distribuovanej architektúry bez zdieľanej pamäte, pri ktorej je komunikácia medzi procesormi veľmi nákladná.

Vykonali sme množstvo meraní a testov, pomocou ktorých sme analyzovali vplyv jednotlivých pokročilých metód použitých v algoritme na jeho celkový výkon. Zistili sme, že mechanizmus výmeny údajov medzi tabuľkami je pre výkon algoritmu úplne kľúčový. Menší, ale nezanedbateľný pozitívny vplyv mala aj heuristika na minimalizovanie množstva zbytočne pridelenej práce.

Podobne sme analyzovali aj sekvenčný alfa-beta algoritmus a jeho zlepšenia. Ukázali sme, že použitie transpozičných tabuliek má veľký potenciál na zvýšenie výkonu algoritmu, ale výsledok je silne závislý na efektívnosti implementácie týchto tabuliek.

Aplikácie výsledkov práce

Popísaný paralelný algoritmus môže byť použitý pri vytváraní herných programov pre rôzne hry na distribuovaných architektúrach, v ktorých je komunikácia medzi uzlami relatívne veľmi nákladná.

Navrhnutý spôsob realizácie distribuovanej transpozičnej tabuľky môže byť veľmi prínosný aj pri riešení všeobecnených problémov metódou prehľadávania priestoru stavov, pokiaľ sa tomto priestore vyskytuje veľké množstvo duplicitných

stavov.

Ďalšia práca

Zostáva pomerne dosť vecí, na ktorých sa dá ďalej pracovať. Viaceré časti algoritmu môžu byť ďalej vylepšované. Vhodným zlepšením master procesu by napríklad bolo automatizované určovanie vhodnej hĺbky jeho vygenerovaného stromu, podľa počtu dostupných procesorov a priemerného stupňa vrcholov v strome (jeho vygenerovaný strom by ani nemusel mať všetky listy v rovnakej hĺbke).

Prínosné by mohlo byť aj podrobnejšie preskúmanie ďalších nahradzovacích stratégii pre transpozičnú tabuľku a na nich založených stratégií výmeny údajov medzi tabuľkami. Samotná implementácia transpozičnej tabuľky nie je tak optimalizovaná, ako by si zaslúžila.

Vzhľadom na obmedzený počet počítačov v dostupnom distribuovanom systéme taktiež nebolo možné prakticky vyskúšať správanie algoritmu na veľkom počte procesorov. Rozšírenie centralizovaného riadenia na hierarchiu masterov tak bolo ponechané v teoretickej rovine.

Dodatok A

Tabuľky výsledkov meraní

poz.	proc.	hĺbka	časy			priem.čas
1	1	15	90	89	92	90
2	1	15	132	125	130	129
3	1	15	288	291	296	292
4	1	15	224	216	218	219
5	1	15	79	88	76	81
6	1	15	95	82	85	87
7	1	15	209	208	207	208
8	1	15	107	50	81	79
9	1	15	77	75	84	79
10	1	15	125	123	123	124
11	1	15	259	253	285	266
12	1	15	187	198	182	189

Obr. A.1: Časy behov sekvenčného algoritmu.

poz.	proc.	hĺbka	časy			priem.čas	zrýchlenie
1	2	3+12	149	159	151	153	0.59
2	2	3+12	60	60	61	60	2.15
3	2	3+12	170	174	166	170	1.72
4	2	3+12	28	29	29	29	7.55
5	2	3+12	17	17	16	17	4.76
6	2	3+12	47	46	48	47	1.85
7	2	3+12	81	82	81	81	2.57
8	2	3+12	12	12	13	12	6.58
9	2	3+12	32	34	35	34	2.32
10	2	3+12	41	44	42	42	2.95
11	2	3+12	90	89	82	87	3.06
12	2	3+12	63	67	61	64	2.95

Obr. A.2: Pokročilý paralelný algoritmus na 2 procesoroch.

poz.	proc.	hĺbka	časy			priem.čas	zrýchlenie
1	4	3+12	117	89	118	108	0.83
2	4	3+12	40	39	39	39	3.31
3	4	3+12	99	107	107	104	2.81
4	4	3+12	19	19	18	19	11.53
5	4	3+12	22	12	13	16	5.06
6	4	3+12	46	29	45	40	2.18
7	4	3+12	50	46	83	60	3.47
8	4	3+12	12	30	13	18	4.39
9	4	3+12	21	21	21	21	3.76
10	4	3+12	28	27	27	27	4.59
11	4	3+12	51	48	53	51	5.22
12	4	3+12	49	199	47	98	1.93

Obr. A.3: Pokročilý paralelný algoritmus na 4 procesoroch.

poz.	proc.	hĺbka	časy			priem.čas	zrýchlenie
1	8	3+12	72	63	51	62	1.45
2	8	3+12	24	24	24	24	5.38
3	8	3+12	63	65	63	64	4.57
4	8	3+12	15	14	14	14	15.64
5	8	3+12	9	9	9	9	9
6	8	3+12	22	29	34	28	3.12
7	8	3+12	65	35	35	45	4.62
8	8	3+12	22	6	6	11	7.18
9	8	3+12	15	15	16	15	5.27
10	8	3+12	20	18	20	19	6.53
11	8	3+12	41	42	41	41	6.49
12	8	3+12	130	47	46	74	2.55

Obr. A.4: Pokročilý paralelný algoritmus na 8 procesoroch.

poz.	proc.	hĺbka	časy			priem.čas	zrýchlenie
1	16	3+12	23	10	16	16	5.63
2	16	3+12	8	33	4	25	5.16
3	16	3+12	38	39	26	34	A.59
4	16	3+12	4	7	4	5	43.8
5	16	3+12	20	10	11	14	5.76
6	16	3+12	20	30	21	24	3.63
7	16	3+12	31	23	5	20	10.4
8	16	3+12	28	10	33	24	3.29
9	16	3+12	12	9	33	18	4.39
10	16	3+12	9	16	4	10	12.4
11	16	3+12	92	23	33	49	5.43
12	16	3+12	113	46	106	88	2.15

Obr. A.5: Pokročilý paralelný algoritmus na 16 procesoroch.

Dodatok B

Obsah CD prílohy

Zoznam adresárov na CD prílohe a popis ich obsahu:

- `/bibliography/` - plné texty použitej literatúry, ktoré boli voľne dostupné v elektronickej podobe.
- `/parallel/` - zdrojové súbory paralelného programu.
- `/positions/` - súbory s kódmi použitých testovacích pozícií
- `/sequential/` - zdrojové súbory sekvenčného programu
- `/thesis/` - táto práca v elektronickej podobe (formát pdf)

Literatúra

[A+94] L. Victor Allis et al. *Proof-number Search.* Artificial Intelligence vol. 66, issue 1, 1994.

[Bre98] Dennis Breuker. *Memory versus Search in Games.* Universiteit Maastricht, 1998.

[Die00] Reinhard Diestel. *Graph Theory.* Springer-Verlag, 2000.

[Elo78] Arpád E. Elo. *The Rating of Chess Players, Past and Present.* Arco Pub, 1978.

[Fel93] Rainer Feldman. *Game Tree Search on Massively Parallel Systems.* University of Paderborn, 1993.

[Fel97] Rainer Feldman. *Computer Chess: Algorithms and Heuristics for a Deep Look into the Future.* Sofsem '97: Theory and Practice of Informatics, 1997.

[FMM91] Rainer Feldman, Peter Mysliwietz, and Burkhard Monien. *A Fully Distributed Chess Program.* University of Paderborn, 1991.

[GB+94] Al Geist, Adam Beguelin, et al. *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[Gre05] Amy Greenwald. *Artificial Intelligence: Adversial Search.* lecture notes, 2005.

[Gus90] John Gustafson. *Fixed Time, Tiered Memory, and Superlinear Speedup.* Proceedings of the Fifth Distributed Memory Computing Conference, 1990.

[Hei98] Ernst Heinz. *DarkThought Goes Deep.* University of Karlsruhe, 1998.

[Hsu90] Feng-Hsiung Hsu. *Large Scale Parallelisation of Alpha-Beta search: An Algorithmic and Architectural Study with Computer Chess.* Carnegie Mellon University, 1990.

[JS97] Andreas Junghanns and Jonathan Schaeffer. *Search versus Knowledge in Game-Playing Programs Revisited.* University of Alberta, 1997.

[KM75] Donald E. Knuth and Ronald W. Moore. *An Analysis of Alpha - Beta Pruning.* Artificial Intelligence 4/75, 1975.

[MP84] Tony Marsland and Fred Popowich. *Parallel Game-tree Search.* University of Alberta, 1984.

[Pla96] Aske Plaat. *Research, Re: Search and Re-Search.* Erasmus University Rotterdam, 1996.

[RN02] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2002.

[SL96] Jonathan Schaeffer and Robert Lake. *Solving the Game of Checkers.* Game of No Chance, vol. 29, 1996.

[Sla96] Branislav Slantchev. *Game Theory: Elements of Basic Models.* University of California - San Diego, 1996.

[Upt99] Robin Upton. *Dynamic Stochastic Control - A New Approach to Game Tree Searching.* University of Warwick, 1999.

[Wik05] Wikipedia.org. *ELO rating system.* Wikipedia, The Free Encyclopedia, 2005.

[Zob90] Albert Zobrist. *A New Hashing Method with Applications for Game Playing.* University of Wisconsin, 1990.