Semestrální projekt PAR 2006/2007:

Paralelní algoritmus pro řešení problému Othello

Michal Augustýn Michal Trs

9. prosince 2006

5. ročník, obor počítače, K336 FEL CVUT, Karlovo nám. 13, 121 35 Praha 2

1 Zadání - pravidla a cíl hry

Na počátku je na čtvercové šachovnici S o straně k rozmístěno q bílých kamenů. Tomuto rozmístění budeme říkat počáteční konfigurace X. V počáteční konfiguraci jsou na šachovnici umístěny jen bílé kameny a to tak, že nejsou na hranách šachovnice (tedy ani v rozích šachovnice). Jeden tah je umístění jednoho černého kamene na volné pole šachovnice S vedle libovolného bílého kamene. Všechny bílé kameny, které leží mezi přidávaným černým kamenem a již existujícími černými kameny ve směru os nebo úhlopříček, se změní na černé. Cílem hry je minimálním počtem tahů očernit všechny bílé kameny. Při řešení záleží na pořadí pokládání černých kamenů.

2 Řešení

2.1 Společné části řešení

Horní mez: V nejhorším případě bude řešení nalezeno v *min(2q,neighborhood(q)))* tazích, kde *neighborhood(q)* je počet volných políček sousedících s bílými kameny.
Důkaz: buď budeme na očernění jednoho bílého kamene potřebovat maximálně 2 černé kameny, nebo položíme tolik černých kamenů, kolik je volných políček sousedících s bílými kameny.

Těsná dolní mez na počet tahů je rovna výrazu pocet_komponent(F) kde vyraz pocet_komponent(F) představuje počet komponent. Komponenta je zde definována jako maximální (tzn. nedá se zvětšit) množina bílých kamenu, kde každý její prvek je (vertikálním / horizontálním / diagonálním) sousedem jiného prvku.
Důkaz: Jedna komponenta vyžaduje alespoň 2 černé kameny a 2 komponenty mohou sdílet nejvýše 1 černý kámen.

2.2 Popis sekvenčního algoritmu

Řešení vždy existuje. Sekvenční algoritmus je typu BB-DFS s hloubkou prohledávaného prostoru omezenou na hodnotu horní meze. Při implementaci sekvenčního algoritmu ukládáme na zásobník následníky aktuálního stavu v pořadí klesající funkce úspěšnosti položení (počet přebarvených kamenů). Za následníky vybíráme ty tahy, při jejichž zahrání dojde k položení černého kamene na místo, které přímo sousedí s místem, kde je umístěn nějaký bílý kámen. Při každé expanzi ukládáme následníky na zásobník neseřazené a teprve při konci expanze ohodnotíme a seřadíme nově expandované tahy. Tímto přednostně pokládáme černé kameny na místa, kde lze v příštím tahu obarvit co nejvíce bílých kamenů. Cenu, kterou minimalizujeme je počet tahů, tedy počet

černých kamenů nutných k obarvení všech bílých kamenů. V průběhu výpočtu si neustále udržujeme nejlepší aktuálně nalezené řešení. Expanze se neprovádí, pokud by expanzí vznikly řešení, která by měla více tahů než nejlepší dosud nalezené řešení. Nejlepší nalezené řešení je tedy na začátku inicializováno na hodnotu horní meze. V průběhu běhu algoritmu se pracuje stále jen s jednou šachovnicí, jejíž obsah se mění. Při zahrání tahu se umístí černý kámen a provede se přebarvení příslušných bílých kamenů. K tahu se přitom vedle polohy pokládaného kamene uloží i seznam těchto přebarvených kamenů, což umožní snadno provádět zpětné tahy. Toto řešení je sice náročnější na implementaci, ale je paměťově méně náročné a lépe použitelné pro paralelní algoritmus.

2.3 Popis paralelního algoritmu a jeho implementace v MPI

Paralelní algoritmus je typu L-PBB-DFS-D. Jedná se tedy o paralelní BB-DFS algoritmus s prohledáváním DDE stavového prostoru. To znamená, že to, zda se bude prohledávat celý stavový prostor záleží na vstupních datech. Pokud je nalezeno řešení rovné dolní mezi, je algoritmus ukončen a není tedy třeba prohledávat celý stavový prostor. Z algoritmů pro hledání dárce jsme zvolili a implementovali algoritmus ACZ-AHD, tedy asynchronní cyklické žádosti. V tomto algoritmu si každý procesor udržuje lokální čítač, jehož hodnota označuje potenciálního dárce. Procesor s prázdným zásobníkem (tedy bez práce) pošle žádost o práci procesoru, jenž odpovídá aktuální hodnotě čítače. Pokud není tento dárce schopen práci poskytnout, je čítač cyklicky inkrementován a žádost o práci je poslána dalšímu potenciálnímu dárci. Z algoritmů pro dělení zásobníku jsme vybrali a implementovali algoritmus D-ADZ, tedy půlení zásobníku u dna. V tomto algoritmu žadatel obdrží polovinu prvků zásobníku z úrovně, v níž existuje prvek, který lze darovat. Tedy nejprve dojde k vyhledání úrovně zásobníku (hloubky), v níž existuje alespoň jeden darovatelný prvek. Poté dojde ke spočítání darovatelných prvků v této úrovni a polovina z nich je darována. Začíná se na úrovni 1, tedy u dna zásobníku, tedy u prvků, které byly umístěny na zásobník před nejdelší dobou, tedy prvky, které vedou k prohledávání největšího stavového prostoru. Při dělení zásobníku ignorujeme úrovně zásobníku, které jsou o určitou konstantu nad horní mezí řešení. Tuto řeznou rovinu jsme empiricky stanovili na úroveň o 2 nižší než je horní mez řešení. Z algoritmů pro distribuované ukončení paralelnímu výpočtu jsme zvolili modifikovaný Dijkstrův algoritmus pro ukončení paralelního výpočtu, jenž je popsán a doporučován na stránkách podpory předmětu. Tento algoritmus je založen na posílání černých a bílých pešků v kruhu mezi procesory. Pokud některý z procesorů nalezne řešení odpovídající dolní mezi, pošle tuto informaci všem ostatním procesorům a dojde k ukončení algoritmu.

2.4 Spuštění programu

Program má 3 povinné parametry a je nutné je zadat v tomto pořadí `./othello k q input.txt`. Parametr k - délka strany šachovnice, q - počet bílých kamenů a *input.txt* je textový soubor, kde jednomu řádku odpovídá souřadnice (x,y) jednoho bílého kamene.

2.5 Příklad zadání a výsledku

Příklad zadání pro úlohu 3: `othello 7 6 ../data/15m.txt`
obsah souboru 15m.txt:

```
4,2
4,3
4,4
2,5
6,5
4,6
```

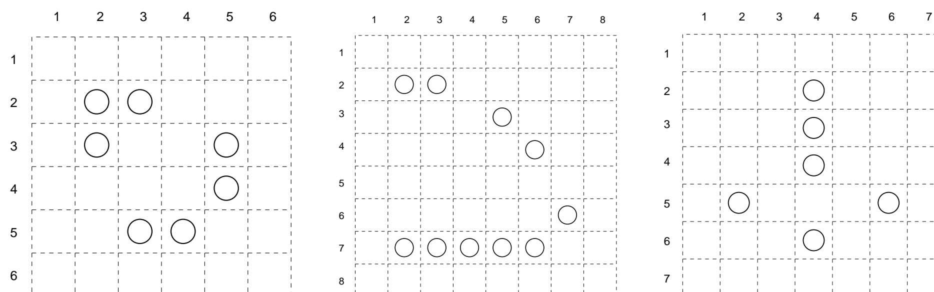
Výstup programu pro úlohu 3:

```
The result is 6.
 x: 7   y: 4
 x: 5   y: 6
 x: 3   y: 6
 x: 1   y: 4
 x: 4   y: 5
 x: 4   y: 1

   6
   B
   B
4  B  1
 B 5 B
  3B2
```

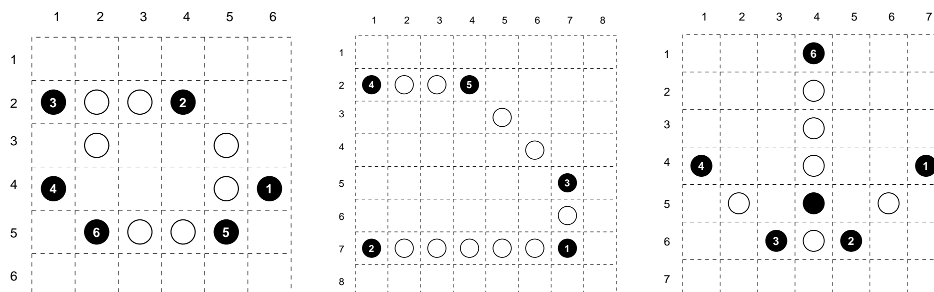
3 Naměřené a vypočtené hodnoty

3.1 Průběh měření



Obrázek 1: Zadání 1, 2, 3

Nejprve jsme nalezli 3 zadání, která na jednom CPU trvají přibližně 5, 10 a 15 minut. Jejich grafické znázornění je na obrázku 1. Dále v souboru *main.c* zakomentujeme direktivu *ENABLE_OUTPUT*, tím ve zkompilovaném souboru nebudou veškeré výpisy během paralelního výpočtu.



Obrázek 2: Řešení 1, 2, 3

Pro vlastní měření jsme napsali / upravili script *par.submit.sh* pro plánovač qsub. Ve scriptu postačí měnit parametr *#PBS -l nodes=i* - počet CPU, kde $i = 2, 4, 8, 12$ a proměnou *TEST_NUM=j*

- číslo zadání (1, 2, 3). Výsledné obarvení je na obrázku 2. Zadání je pevně dané (není generováno náhodně), paralelní algoritmus pracuje deterministicky a proto není nutné měřit vícekrát a hodnotu průměrovat.

3.2 Naměřené hodnoty

Zadání	síť / #CPU	1	2	4	8	12
1	Ethernet	278.87	87.84	72.85	78.82	83.79
2		693.72	435.57	421.60	33.52	35.99
3		914.94	741.59	709.54	161.86	172.67
1	Myrinet	271.64	90.58	66.72	66.55	66.53
2		681.93	418.90	388.89	28.40	28.42
3		899.37	714.10	659.66	138.62	138.56

Tabulka 1: Naměřené časy v sekundách

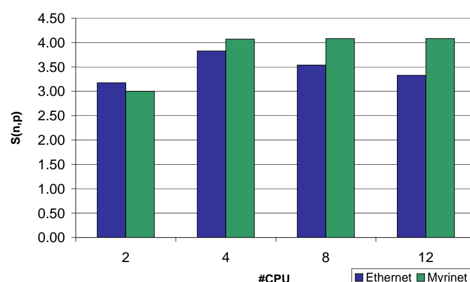
3.3 Vypočtené hodnoty

Z tabulky 1 vypočteme zrychlení $S(n,p)$ podle vztahu $S(n,p) = \frac{SU(n)}{T(n,p)}$. $SU(n)$ je čas sekvenčního řešení. V tabulce 1 odpovídá sloupci #CPU(1). Za $T(n,p)$ postupně dosazujeme hodnoty ze sloupců #CPU(2-12). Výsledné hodnoty jsou v tabulce 2.

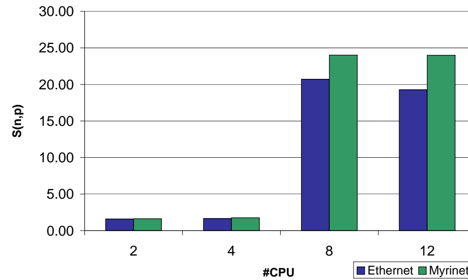
Zadání	síť / #CPU	2	4	8	12
1	Ethernet	3.17	3.83	3.54	3.33
2		1.59	1.65	20.70	19.27
3		1.23	1.29	5.65	5.30
1	Myrinet	3.00	4.07	4.08	4.08
2		1.63	1.75	24.01	23.99
3		1.26	1.36	6.49	6.49

Tabulka 2: Zrychleni $S(n,p)$

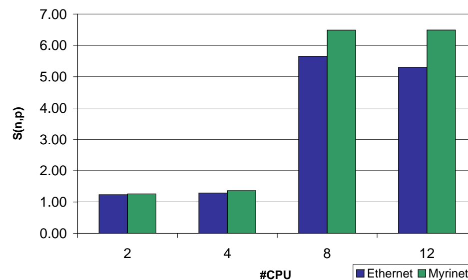
3.4 Grafy



Obrázek 3: Závislost $S(n,p)$ na #CPU pro zadání 1



Obrázek 4: Závislost $S(n, p)$ na #CPU pro zadání 2



Obrázek 5: Závislost $S(n, p)$ na #CPU pro zadání 3

4 Vyhodnocení

Díky zkušenostem získaných během studia se nám podařilo navrhnout sekvenční řešení, které jsme následně snadno paralelizovali.

Při měření implementace bylo nejtěžší nalézt úlohy které trvají cca 5, 10, 15 min na jednom CPU. Úloha 2 má řešení rovné dolní těsné mezi, proto zde můžeme předpokládat superlineární zrychlení. Z tabulky 2 je dobře patrné, že k superlineárním zrychlení došlo i v jiných případech (viz. obrázek 3). Tohoto je dosaženo tím, že procesory udržují globálně aktuálně nejlepší řešení a tím mohou efektivně ořezávat stavový prostor.

Dále je z tabulky 2 (případně obrázků 4 a 5) patrné, že zrychlení není lineární. Je to způsobeno nezanedbatelnou komunikační složitostí. Ta se skládá ze zasílání žádostí o práci, vlastní práci, dosažení nejlepších výsledků a distribuovaného ukončení výpočtu.

Hranice škálovatelnosti není jasně patrná. Velikost stavového prostoru je závislá na délce strany k šachovnice, dále na počtu komponent a velice důležitou roli hraje samotné rozmístění kamenů. Z experimentů při hledání testovacích zadání, jsme zjistili, že zadání které má 1 komponentu a délku strany $k < 10$ je výpočetní čas v řádu sekund a proto paralelní řešení nemá význam. Naproti tomu, pro $k > 6$ a počet komponent > 3 se výpočetní čas pohybuje v řádech minut až desítek minut. Pro tato zadání se vyplatí paralelní řešení.

Při implementaci jsme nenarazili na žádné problémy a zadání je přesně splněno. Proto zde nenavrhujeme žádné zlepšení.

Reference

[1] Tvrdík: *Paralelní systémy a algoritmy* ČVUT 2006

[2] StarCluster: *link 36PAR - Laboratorní cvičení*
http://star.felk.cvut.cz