# Semestrální projekt MI-PAR 2010/2011:

Paralelní algoritmus pro řešení problému **Jeskyně pokladů (JSP)**

Daniel Kavan <kavanda1@fit.cvut.cz>
Michal Včelička <vcelimic@fit.cvut.cz>

magisterské studium, FIT ČVUT, Kolejní 550/2, 160 00 Praha 6
8. prosinec 2010

# 1 Definice problému a popis sekvenčního algoritmu

## 1.1 Zadání úlohy Jeskyně pokladů (JSP)

### 1.1.1 Úkol

Náhodou se shodou příznivých okolností ocitnete v jeskyni pokladů. Je zde mnoho (přesněji n) předmětů. U každého předmětu znáte jeho cenu a objem. Chcete si z této jeskyně odnést předměty v co největší celkové hodnotě, předměty ale můžete odnést jen ve svém batohu. Máte batoh, kam se vejde neomezený počet předmětů, ale má maximální objem $V$.

Úkolem je nalézt charakteristický vektor $a$ množiny odnesených věcí, tak aby max($C.a$), kde $O.a \leq V$.

### 1.1.2 Vstupní data:

| Proměnná | Matematická interpretace | Popis |
|---|---|---|
| $n$ | konstanta $n$ | počet předmětů |
| $C[1..n]$ | cenový vektor $C$ | ceny jednotlivých předmětů, $0.9 \leq C[i] \leq 1.1$ |
| $O[1..n]$ | vektor objemu $O$ | objemy jednotlivých předmětů, $0.9 \leq O[i] \leq 1.1$ |
| $V$ | konstanta $V$ | maximální objem předmětů, co lze odnést |

**Tabulka 1 - Vstupní data úkolu**

### 1.1.3 Výstup algoritmu:

Výpis množiny předmětů v batohu včetně celkové ceny a celkového objemu.

## 1.2 Popis sekvenčního algoritmu

Sekvenční algoritmus je typu **BB-DFS**[1], s hloubkou stromu stavů omezenou na $n$. Řešení vždy existuje. Cena, kterou maximalizujeme, je součet cen věcí v batohu. Algoritmus končí, když je cena rovna horní mezi nebo když prohledá celý stavový prostor do hloubky dané $n$.

Těsná horní mez není známa. Odhad horní meze ceny lze vypočítat jako $c\_max_i = V(\max C_i/O_i)$.

[1] Branch-and-Bound Depth-First Search

### 1.2.1 Nástin algoritmu:

Stav je hodnota charakteristického vektoru *a*. Procházení stromu se tedy rovná postupnému nastavování složek vektoru *a* zleva čísly 0 nebo 1. Při prohledávání větve stromu (ohodnoceno je prvních i položek vektoru *a*) se lze vrátit, pokud

1. cena předmětů v batohu + cena zbývajících dosud neuvažovaných věcí ≤ průběžné maximum
2. cena předmětů v batohu + maximální odhad ceny nejlepšího doplnění do batohu ≤ průběžné maximum

### 1.2.2 Odchylky od zadání

V zadání byla specifikována omezení pro hodnoty ve vektorech *O* a *C*:

$$0.9 \leq C[i] \leq 1.1$$
$$0.9 \leq O[i] \leq 1.1.$$

Vstupy s takto omezenými hodnotami bez problémů zpracováváme, nicméně naše implementace si poradí i s hodnotami mimo tento interval. Problémy nám nečiní ani

- předměty těžší než 1.1 (smysl úlohy nezměněn)
- předměty hodnotnější než 1.1 (smysl úlohy nezměněn)
- předměty lehčí než 0.9 (předměty se zápornou hmotností mohou batoh dokonce nadlehčovat)
- předměty lacinější než 0.9 (předměty se zápornou cenou mohou negativně cenově působit na „ostatní obsah batohu“).

### 1.2.3 Naměřené časy pro sekvenční algoritmus různě velká data

| Počet předmětu v batohu (*n*) | Čas potřebný k řešení úlohy[2] [s] |
|---:|---:|
| 4 | 0.000593901 |
| 10 | 0.0027411 |
| 15 | 0.116669 |
| 20 | 3.42538 |
| 22 | 15.9661 |
| 25 | 112.343 |
| 27 | 550.386 |

**Tabulka 2 - Naměřené časy pro různé velikosti batohu**

## 1.3 Implementace sekvenčního algoritmu

Implementujeme algoritmus typu DFS pro stromové prohledávání do hloubky s vlastním zásobníkem. Jasně neperspektivní stavy a jejich podstromy (např. aktuální váha batohu s danými předměty by překročila povolené maximum) v rámci implementace kroku **pruning**[3] vyřazujeme.

Průchod stromem v naší implementaci probíhá zjednodušeně následovně:

1. Na zásobník je uložen kořenový prvek stromu.

[2] Čas vybrané úlohy pro zvolený počet předmětů *n*, přesný čas vždy závisí na datech, zde klademe důraz na řád výsledků.
[3] Krok BB algoritmu, kdy je neperspektivní podstrom vyřazen z prohledávání. Viz zdroj [4a].

2. Na zásobníku existuje minimálně jeden stav. Je vyjmut a otestován, zda je možné ho expandovat
    a. Expanze možná → expandováno → test, zda jsou nové stavy perspektivní
        i. Každý perspektivní stav je vrácen na zásobník
        ii. Každý neperspektivní stav je zahozen
    b. Expanze není možná (list stromu) → test, zda je list lepším řešením než „zatím nejlepší“
        i. List je lepším řešením → list se stává „zatím nejlepším“
        ii. List není zlepšujícím řešením → je zahozen
3. Jakmile je zásobník prázdný → „zatím nejlepší“ je řešením problému.

Ve zdrojovém kódu aplikace je tato implementace sekvenčního algoritmu zpracována ve funkci `procedeNode`.

### 1.3.1 Formát vstupních dat

Vstupní data pro naši implementaci předpokládáme v textovém souboru, který obsahuje:

- počet předmětů
- objem batohu
- dvojice hodnot *value*, *volume* jednotlivých předmětů v počtu právě 2*n*

oddělených znaky typu *whitespace* (mezery, tabulátory, CR, LF, …)

```
3   2.7     1.05  1      1.04  1     1.03  1

```

**Ukázka kódu 1 - Formát vstupních dat**

Ukázka kódu 1 demonstruje formát vstupních dat na 3 předmětech, jejichž hodnoty jsou po řadě 1.05, 1.04, 1.03 a objemy vždy roven 1 a batohu o velikosti 2.7.

### 1.3.2 Formát výstupních dat

Výstupní data jsou opět v textové formě, hlavními částmi jsou

- oznámení počtu procesorových jader podílejících se na výpočtu
- zvolený typ zásobníku
- průběžná lokální maxima jednotlivých vláken
- výsledné optimální naplnění batohu (tj. max. cena, hmotnost, binární vektor obsahu batohu)
- spotřebovaný čas pro běh úlohy [s]

```
P0:There are 2 processes.
StackType: STACK
... případná lokální maxima a dodatečné informace ...
P0:solution with the value of 3538 is:
---Node Content:--- (0)
P0:Vector content: <1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 0
0 0 1 >
P0:Volume: 348
Spotrebovany cas je: 267.985
```

**Ukázka kódu 2 - Formát výstupních dat**

# 2 Popis paralelního algoritmu a jeho implementace v MPI

## 2.1 Použité algoritmy

### 2.1.1 PBB-DFS-V

Pro naši úlohu jsme využili paralelní algoritmus je typu PBB-DFS-V[4]. Tento algoritmus se nejlépe hodí k povaze problému. Při hledání řešení problému typu „knapsack" nelze předem odhadnout v jaké části stromu řešení se nejlepší řešení nachází. Musíme proto v nejhorším případě projít všechny stavy. Dále není potřeba, aby spolu jednotlivé procesy komunikovaly ohledně průběžného stavu řešení, každý proces si pouze ukládá své nejlepší lokální řešení, které je na konci odešle hlavnímu procesu.

### 2.1.2 Výběr dárce

Používáme asynchronní cyklické žádosti. Každý proces si drží vlastní cyklický čítač, jehož hodnota odpovídá počtu procesorů. Na začátku běhu programu je hodnota čítače nastavena na pid + 1. Při každé odeslané žádosti o práci si procesor tento čítač zvětšuje o 1.

### 2.1.3 Algoritmus dělení zásobníku

V programu jsme naimplementovali 2 možné způsoby dělení zásobníku.

- Uzly jsou odebírané shora zásobníku
- Uzly jsou odebírány zdola zásobníku

Pro náš problém je výhodnější druhá použitá implementace, protože stavy poblíž dna zásobníku obsahují pravděpodobně větší část prohledávaného prostoru řešení, než stavy u vrcholu zásobníku.

### 2.1.4 Sběr dat a ukončení výpočtu

Ukončení výpočtu obstarává algoritmus ADUV. Jeho zjednodušený popis je následující:

- Každý proces má svůj dvoustavový vnitřní stav.
- Pokud proces odešle práci procesoru s nižším PID, nastaví svou hodnotu na BLACK.
- Pokud p0 (hlavní proces) dojde práce, odesílá tzv. peška procesu p1 s barvou WHITE.
- Pokud proces pid != 0 obdrží token a má sám barvu BLACK, tak tokenu nastaví barvu BLACK a sám se přebarví na WHITE. Pokud má procesor barvu WHITE, pouze předává token dále.
- Jestliže p0 obdrží token barvy WHITE, může začít sbírat lokální řešení od ostatních procesů a zahájit vyhodnocení. Pokud obdrží token BLACK, odešle znovu další token s barvou WHITE

## 2.2 Popis algoritmu našeho řešení

1. Je vytvořeno $n$ procesů. Všechny procesy načtou vstupní data (formát je stejný jako u sekvenčního řešení)
    a. Všechny P!=0 čekají na zprávu s počátečními daty
    b. P0 expanduje $n$ prvních stavů na zásobník a rozešle je všem procesorům – výpočet je zahájen
2. Pokud dojde procesu práce, zažádá o další (dárce je zvolen na základě vnitřního čítače každého procesu). Pokud dojde práce P0, před zažádáním o novou ještě odešle token.
    a. Každý proces má pouze $n/2$ pokusů na získání další práce. Pokud není ani na $n/2$ žádost úspěšný, čeká už pouze na ukončení výpočtu
3. Pokud proces obdrží žádost o práci, rozdělí svůj zásobník (pokud má dost práce sám) a odešle práci s následujícím počtem uzlů ve zprávě:
    a. Pokusí se odeslat zprávu s [*počet předmětů*/2] uzlů

[4] Paralelní BB-DFS s vždy úplným prohledáváním stavového prostoru

b. Pokud sám proces nemá tolik uzlů k dispozici, pošle [*velikost zásobníku*/2]
c. Pokud se tento počet uzlů nevejde do zprávy (zpráva má omezený počet byte), odešle proces pouze tolik uzlů, kolik se do zprávy vejde

4. Poté, co jsou zpracována všechna data, dochází k ukončení výpočtu pomocí algoritmu ADUV.
   a. Poté, co P0 obdrží zpět token barvy WHITE, rozešle všem ostatním procesorům žádost o jejich nejlepší lokální řešení.
   b. Když p!=0 odešle svá data, ukončí se.
   c. P0 posbírá výsledky a najde ten nejlepší.

# 3 Měření, výsledky a vyhodnocení

## 3.1 Naměřené časy pro jednotlivé počty procesorů:

| # CPU's | 1 - Infiband | 2 - Infiband | 3 - Infiband | 1 - LAN | 2 - LAN | 3 - LAN |
|---|---|---|---|---|---|---|
| 1 | 550,39 | 562,66 | 562,99 | 550,97 | 573,27 | 540,66 |
| 2 | 267,99 | 280,16 | 272,84 | 271,59 | 280,61 | 271,35 |
| 4 | 131,54 | 133,15 | 131,16 | 130,91 | 134,77 | 136,78 |
| 8 | 66,05 | 67,62 | 66,31 | 65,86 | 67,18 | 64,30 |
| 16 | 32,26 | 32,57 | 32,04 | 32,67 | 31,82 | 31,59 |
| 24 | 17,53 | 17,31 | 15,94 | 16,84 | 17,43 | 16,50 |
| 32 | 16,60 | 15,89 | 16,60 | 16,70 | 16,20 | 19,28 |

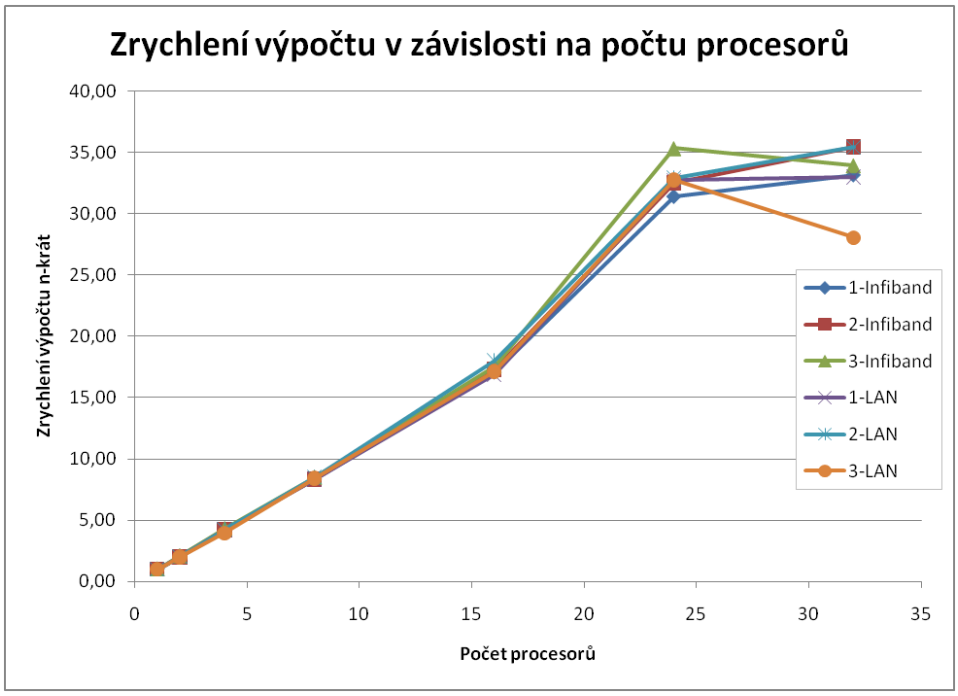
**Tabulka 3 - Délky běhu výpočtů podle počtu procesorů**

## 3.2 Hodnoty zrychlení oproti řešení na jednom procesoru:

| # CPU's | 1-Infiband | 2-Infiband | 3-Infiband | 1-LAN | 2-LAN | 3-LAN |
|---|---|---|---|---|---|---|
| 1 | 1,00 | 1,00 | 1,00 | 1,00 | 1,00 | 1,00 |
| 2 | 2,05 | 2,01 | 2,06 | 2,03 | 2,04 | 1,99 |
| 4 | 4,18 | 4,23 | 4,29 | 4,21 | 4,25 | 3,95 |
| 8 | 8,33 | 8,32 | 8,49 | 8,37 | 8,53 | 8,41 |
| 16 | 17,06 | 17,27 | 17,57 | 16,86 | 18,02 | 17,12 |
| 24 | 31,40 | 32,50 | 35,33 | 32,71 | 32,89 | 32,76 |
| 32 | 33,16 | 35,42 | 33,92 | 33,00 | 35,40 | 28,04 |

**Tabulka 4 - Zrychlení výpočtů podle počtu procesorů**

Graf závislosti zrychlení na počtu procesorů (graf jednotlivých časů jsme vynechali, protože pro názornost řešení se nám tento zdál více vypovídající):




**Obrázek 1 - Graf zrychlení výpočtů na počtu procesorů**

### 3.2.1 Vyhodnocení měření

Měření prokázalo velmi dobré vlastnosti škálování výkonu. Při použití 2-24 procesorových jader se nám ve většině případů podařilo dosáhnout superlineárního zrychlení v porovnání se sekvenčním řešením.

V případě 32 jader zrychlení nestoupalo tolik jako v předešlých případech. Složitost komunikace u jedné z instancí dokonce zapříčinilo horší výsledky než u menšího stupně paralelizace.

Dopad pomalejšího komunikačního kanálu (LAN – Infiband) nebyl při menší paralelizaci patrný, postupně, kdy celkový počet jader narůstal, byl dopad rychlosti přenosu zpráv patrný, nikoliv však rozhodující.

# 4 Závěr

Úspěšně se nám podařilo implementovat sekvenční algoritmus pro řešení problému typu batoh („The knapsack problem“ – viz zdroj [4b]). Abychom připravili algoritmus na budoucí paralelizaci, zvolili jsme řešení s využitím vlastního zásobníku na místo systémového, který by byl součástí rekurzivního přístupu.

Sekvenční algoritmus jsme úspěšně paralelizovali s využitím knihovny MPI. Výběr dárce dat pro volné výpočetní jednotky byl realizován pomocí asynchronních cyklických žádostí, ukončení výpočtu se provádí algoritmem typu ADUV.

Následně jsme provedli měření spotřebovaného času pro běh úloh netriviální složitosti sekvenčně a paralelně na 1-32 procesorových jádrech. Měření bylo dále rozděleno podle komunikačních kanálů: LAN a Infiband.

Z výsledků měření vyplynulo, že naše implementace dokáže škálovat výkon velmi dobře – při použití 2 – 24 jader jsme dosáhli dokonce superlineárního zrychlení. Výsledky pro zvolená zadání pro použití 32 jader již ukázaly přínos výrazně menší, u jedné z instancí byly výsledky, zejména kvůli režii komunikace mezi procesy, dokonce horší než při použití 24 jader. Zvolený druh komunikace má výraznější dopad pouze při vysoké míře paralelizace, ale není rozhodující.

# 5 Literatura


[1] C++ Reference
[1a] STL vector <http://www.cplusplus.com/reference/stl/vector/>
[1b] STL deque (double ended queue) <http://www.cplusplus.com/reference/stl/deque/>

[2] Šoch, M.: Programovaní pod MPI (pro MI-PAR) <https://users.fit.cvut.cz/~soch/mi-par/>

[3] Šimeček, I.: Poznámky k implementaci (pro MI-PAR)
<https://edux.fit.cvut.cz/courses/MI-PAR/labs/poznamky_k_implementaci>

[4] Wikipedia.org:
[4a] Branch and Bound <http://en.wikipedia.org/wiki/Branch_and_bound>
[4b] Knapsack problem <http://en.wikipedia.org/wiki/Knapsack_problem>