

UNIVERZITA KOMENSKÉHO V BRATISLAVE  
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

*Cache-oblivious algoritmy a ich vizualizácia*  
Bakalárska práca

UNIVERZITA KOMENSKÉHO V BRATISLAVE  
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

*Cache-oblivious algoritmy a ich vizualizácia*  
Bakalárska práca

Študijný program: Informatika  
Študijný odbor: 2508?? Informatika  
Školiace pracovisko: Katedra Informatiky FMFI  
Vedúci práce: Mgr. Jakub Kováč, PhD.



Univerzita Komenského v Bratislave  
Fakulta matematiky, fyziky a informatiky

## ZADANIE ZÁVEREČNEJ PRÁCE

**Meno a priezvisko študenta:** Ladislav Pápay  
**Študijný program:** informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)  
**Študijný odbor:** 9.2.1. informatika  
**Typ záverečnej práce:** bakalárska  
**Jazyk záverečnej práce:** slovenský

**Názov:** Cache-oblivious algoritmy a ich vizualizácia / *Cache-oblivious algorithms and their visualization*

**Cieľ:** Cieľom práce je uviesť prehľad a implementovať vizualizácie vybraných cache-oblivious algoritmov a dátových štruktúr.

**Kľúčové slová:** cache-oblivious algoritmy, vizualizácia

**Vedúci:** Mgr. Jakub Kováč, PhD.  
**Katedra:** FMFI.KI - Katedra informatiky  
**Vedúci katedry:** doc. RNDr. Daniel Olejár, PhD.  
**Dátum zadania:** 30.10.2013

**Dátum schválenia:** 30.10.2013  
doc. RNDr. Daniel Olejár, PhD.  
garant študijného programu

.....  
študent

.....  
vedúci práce

## Podakovanie

Nemáte za čo.

# Abstrakt

Slovenský abstrakt.

# Abstract

English abstract.

# Obsah

<b>Úvod</b>	<b>1</b>
<b>1 Cache-oblivious algoritmy a dátové štruktúry</b>	<b>2</b>
1.1 Model . . . . .	2
1.2 Výmena stránok . . . . .	3
1.3 Základné algoritmy . . . . .	4
1.4 Matice . . . . .	5
1.5 Stromy . . . . .	5
1.6 Triedenie . . . . .	5

## Zoznam obrázkov



# Zoznam tabuliek

# Úvod

Pri tvorbe efektívnych algoritmov sa najčastejšie zaujíname o časovú zložitosť a prípadne aj pamäťovú zložitosť. Dôležitým faktorom však môže byť aj množstvo pamäťových operácií. Tie majú priamy vplyv na výslednú časovú zložitosť avšak pri asymptotickej analýze sa často považujú za operáciu vykonateľnú v konštantnom čase. V prípadoch keď pracujeme s veľkým objemom dát, ktoré sú uložené na médiu s veľkou kapacitou ale nízkou prístupovou rýchlosťou, môže byť výhodné minimalizovať množstvo zápisov a čítaní z tohto média.

Klasickým prístupom v týchto prípadoch boli takzvané *cache-aware* algoritmy, ktoré potrebujú poznať presné parametre pamäťového systému pre dosiahnutie požadovanej efektivity. V tejto práci sa budeme venovať *cache-oblivious* algoritmom, ktoré sú asymptoticky rovnako efektívne ako *cache-aware*, avšak bez nutnosti poznať parametre pamäte.

Okrem samozrejmej výhody, kedy nám stačí jedna univerzálna implementácia algoritmu pre ľubovoľné množstvo rôznych systémov, prinášajú tieto algoritmy aj iné zlepšenia. V takmer každom systéme je pamäť hierarchická, zložená z viacerých úrovní, pričom každá je väčšia ale pomalšia ako predošlá. Pri *cache-aware* by pre optimálne využitie každej z úrovní pamäte bolo potrebné poznať parametre každej úrovne, a rekurzívne vnoriť do seba mnoho inštancií, každú optimalizovanú pre jednu úroveň. No v *cache-oblivious* nám stačí jedna inštancia - keďže nepozná parametre žiadnej úrovne, bude rovnako dobre fungovať na každej z nich.

Súčasťou práce je vysvetliť problematiku *cache-oblivious* pamäťového modelu, uviesť prehľad algoritmov a dátových štruktúr a vytvoriť vizualizácie vybraných z nich. Vizualizácie sú implementované ako súčasť programu *Alg-Vis*. Cieľom týchto vizualizácií je umožniť užívateľom experimentovať s implementáciou týchto algoritmov a dátových štruktúr, sledovať ich prácu krok po kroku a uľahčiť ich porozumeniu.

# Cache-oblivious algoritmy a dátové štruktúry

## 1.1 Model

Klasický model pre analýzu pamäťovej hierarchie s dvoma úrovňami je *external-memory* model. Ten popisuje procesor, ktorý ma k dispozícii pracovnú pamäť (*cache*) obmedzenej veľkosti  $M$ . Všetky dáta, s ktorými algoritmus pracuje sa musia nachádzať niekde v tejto pamäti, ako bloky veľkosti  $B$  (spolu teda  $M/B$  blokov). Ostatné dáta sa nachádzajú na disku, ktorý ma prakticky neobmedzenú veľkosť, no s jeho blokmi nie je možné pracovať priamo. Je potrebné najprv pomocou operácií čítania a zápisu na disk presunúť blok z disku do cache a naopak.

Pre tento model je možné vyrobiť algoritmy, ktoré budú tieto dve úrovne pamäte používať efektívne, za predpokladu, že poznajú parametre  $B$  a  $M$ . Nevýhodou je nutnosť algoritmu explicitne spravovať pamäť a presúvať jednotlivé bloky medzi diskom a cache.

Na rozdiel od klasického modelu, *cache-oblivious* model používa tieto parametre iba pri analýze a nie pri samotnom návrhu. To znamená, že keď funguje efektívne pre neznáme parametre, musí fungovať rovnako efektívne pre ľubovoľné také parametre. Vďaka tomu bude fungovať pre systémy s rôznou konfiguráciou pamäte bez nutnosti úprav. Taktiež, pre systémy s viacerými úrovňami pamäte bude fungovať medzi každou dvojicou v hierarchii. A keďže bez znalosti  $B$  nie je možné explicitne spravovať pamäť a presúvať bloky, *cache-oblivious* algoritmy prenechávajú túto činnosť na iné úrovne (operačný systém, hardware).

## 1.2 Výmena stránok

V prípade, že sa *cache-oblivious* algoritmus pokúsi o čítanie z bloku pamäte, ktorý sa momentálne nenachádza v cache, je potrebné ho tam najskôr skopírovať. Ak cache nie je plná tak to nie je problém. V opačnom prípade je však potrebné uvoľniť miesto tým, že sa vyberie blok z cache, jeho obsah sa zapíše na disk a následne sa požadovaný blok uloží na jeho miesto v cache. Tento proces sa nazýva výmena stránok (*page replacement*), a algoritmus rozhodujúci, ktorý blok z cache odstrániť, voláme stratégiou výmeny stránok (*page-replacement strategy*)

Ak by sa táto stratégia správala tak, že vždy odstráni blok, ktorý bude potrebný v ďalšom kroku algoritmu, tak by bolo zakaždým presúvať bloky medzi diskom a cache. To by znamenalo, že počet blokov, s ktorými cache môžeme pracovať je prakticky  $1 = M/B$ . Ďalším problémom je asociatívnosť cache - z praktických dôvodov je často možné daný blok z disku uložiť len na niekoľko pozícií v cache. Inak by bolo potrebné ukladať spolu s každým blokom jeho adresu, čo by redukovalo celkový počet blokov, ktoré sa do cache zmestia. V prípade nízkej asociativity môžu opäť nastať situácie, kedy je algoritmus schopný využiť iba malý počet blokov v cache.

Tieto problémy *cache-oblivious* model obchádza predpokladom ideálnej cache - cache, ktorá je plne asociatívna (každý blok disku je možné uložiť v každom bloku cache) a používa optimálnu stratégiu výmeny stránok, ktorá vždy odstráni blok, ktorý bude potrebný najneskôr. Prvý predpoklad je síce v reálnych systémoch nepraktický, no z teoretického hľadiska je v poriadku. Čo však s druhým predpokladom, kedy stratégia výmeny stránok je schopná predpovedať budúcnosť? Ako ukazujú nasledovné lemy, bez týchto predpokladov na reálnom systéme s nízkou asociativitou a jednoduchou stratégiou výmeny stránok sa algoritmus zhorší len o konštantný faktor.

**Lema 1.2.1.** *Algoritmus, ktorý v ideálnej cache veľkosti  $M$  s blokmi veľkosti  $B$  vykoná  $T$  pamäťových operácií, vykoná najviac  $2T$  pamäťových operácií v cache veľkosti  $2M$  s blokmi veľkosti  $B$  pri použití stratégie LRU alebo FIFO. [?, Lemma 12]*

Stratégia *LRU* (least recently used) vyberá vždy blok, ktorý bol najdlhšie nepoužitý. Implementácia vyžaduje udržiavať si ku každej položke počítadlo, ktoré sa pri prístupe nastaví na nulu a pri prístupe k iným položkám zvýši o jedna. Potom stačí odstrániť položku s najväčšou hodnotou počítadla.

Stratégia *FIFO* (first in, first out) je ešte jednoduchšia - položky sú udržiavané vo fronte, pričom nové vkladáme na koniec a v prípade, že sa minie miesto v cache a je potrebné nejaké uvoľniť, tak vezmeme prvok na začiatku fronty a odstránime ho.

**Lema 1.2.2.** *Plne asociatívna cache veľkosti  $M$  sa dá simulovať s použitím  $\mathcal{O}(M)$*

*pamäte tak, že prístup ku každému bloku v cache zaberie v priemernom prípade  $\mathcal{O}(1)$  času. [?, Lemma 16]*

## 1.3 Základné algoritmy

Najjednoduchším *cache-oblivious* algoritmom je prechod poľa a výpočet agregáčnej funkcie. Majme pole  $A$  veľkosti  $|A| = N$  a označme jeho prvky  $A_1, \dots, A_n$ . Chceme vypočítať hodnotu  $f(g, A)$ , kde  $g$  je agregáčna funkcia,  $g_0$  je počiatočná hodnota a  $f$  je definovaná nasledovne:

$$\begin{aligned} f(A_1, \dots, A_n) &= g(A_1, f(A_2, \dots, A_n)) \\ f(\emptyset) &= g_0 \end{aligned}$$

Tento algoritmus s použitím vhodnej funkcie  $g$  a hodnoty  $g_0$  je možné použiť na rôzne, často užitočné výpočty, ako napríklad maximum ( $g(x, y) = \max(x, y)$ ,  $g_0 = -\infty$ ), minimum, suma ( $g(x, y) = x + y$ ,  $g_0 = 0$ ) a podobne.

V prípade *cache-aware* algoritmu by sme pole  $A$  mali uložené v  $\lceil N/B \rceil$  blokoch veľkosti  $B$ . Pri výpočte  $f$  by sme postupne tieto bloky načítali do cache a pracovali s nimi. V rámci jedného bloku počas výpočtu nedochádza k pamäťovým presunom. Zároveň stačí každý prvok spracovať raz a teda celkový počet pamäťových operácií bude presne rovný počtu blokov,  $\lceil N/B \rceil$ . Tento algoritmus však požaduje znalosť parametra  $B$ .

Jednoducho však vieme dosiahnuť (takmer) rovnakú zložitosť aj v prípade *cache-oblivious*. Uložíme si pole  $A$  do súvislého úseku pamäte - k tomu nepotrebujeme poznať hodnotu  $B$  ani žiadne iné parametre. Zvyšok algoritmu prebieha rovnako ako v predchádzajúcom prípade. Každý blok obsahujúci nejaký prvok poľa  $A$  bude teda presunutý do cache práve raz, a žiadne iné presuny nenastanú. Ostáva zistiť, koľko takých blokov môže byť.

Keďže nepoznáme veľkosti blokov v pamäti, nevieme pri ukladaní prvkov poľa zaručiť zarovnanie so začiatkom bloku. V najhoršom prípade uložíme do prvého bloku iba jeden prvok. Potom bude nasledovať  $\lfloor N/B \rfloor$  plných blokov a nakoniec ešte najviac jeden blok, ktorý opäť nie je plný. Spolu máme teda  $\lfloor N/B \rfloor + 2$ . Pokiaľ  $\lfloor N/B \rfloor < \lceil N/B \rceil$  a teda najviac  $\lceil N/B \rceil + 1$  blokov. V opačnom prípade  $B$  delí  $N$ , teda v prvom a poslednom bloku je spolu presne  $B$  prvkov a medzi nimi sa nachádza najviac  $\frac{N-B}{B} = N/B - 1$  plných. Teda blokov je vždy najviac  $\lceil N/B \rceil + 1$ , čo je až aditívnu konštantu rovnaký výsledok ako uvedený *cache-aware* algoritmus.

## 1.4 Matice

### Násobenie matíc

Ďalším jednoduchým *cache-oblivious* algoritmom je násobenie matíc. Majme dve matice  $A, B$  typu  $N \times N$  a chceme vypočítať ich súčin  $S = A \cdot B$ . Klasický algoritmus bude pri výpočte každého prvku  $S$  postupne prechádzať maticu  $A$  po riadkoch a maticu  $B$  po stĺpoch. Za predpokladu, že sa do cache súčasne zmestia aspoň tri bloky - po jednom z matíc  $A$  a  $B$ , a jeden blok  $S$  obsahujúci prvok, ktorý práve počítame - budeme na každý prvok  $S$  potrebovať najviac  $\mathcal{O}(1 + N/B)$  presunov. Celkovo teda vykonáme najviac  $\mathcal{O}(N^2 + N^3/B)$  pamäťových presunov.

Doplniť *cache-oblivious* verziu + analýzu

## 1.5 Stromy

### Statické stromy

...

## 1.6 Triedenie

V *cache-aware* modely pamäte je spodným odhadom na počet pamäťových presunov pri triedení porovnávaním  $\Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ . [??]. Algoritmus, ktorý túto hranicu dosahuje je  $M/B$ -cestný mergesort (triedenie zlučovaním). Na rozdiel od klasického mergesortu si pri zlučovaní pamätá  $B$  prvkov z každého z  $M/B$  zoznamov a pri vyprázdnení načíta opäť celý blok s  $B$  prvkami. Zlúčenie listov celkovej dĺžky  $N$  teda vyžaduje  $\mathcal{O}(N/B)$  pamäťových presunov.

Avšak *cache-oblivious* algoritmy musia fungovať bez znalosti  $M$  a  $B$  a teda bez možnosti vypočítať  $M/B$ , najväčší počet zoznamov, ktoré môžeme súčasne zlučovať a pamätať si z každého  $B$  prvkov v cache. Najlepšie, čo môžeme predpokladať je  $M/B \geq 2$ , teda vieme aspoň dva zoznamy zlučovať. Teda implementácia 2-cestného mergesort algoritmu je *cache-oblivious*, a funguje pre ľubovoľné parametre. Avšak počet pamäťových presunov bude  $\Theta(\frac{N}{B} \log_2 \frac{N}{B})$ , to znamená, že zväčšenie  $M$  tento algoritmus nezrýchli, keďže využívame vždy iba malú časť cache.

Ideálny *cache-oblivious* algoritmus by dosahoval rovnakú, optimálnu hranicu počtu presunov ako  $M/B$ -cestný mergesort, no bez znalosti týchto parametrov. Jedným z takýchto efektívnych *cache-oblivious* algoritmov je takzvaný *funnel sort* - lievnikové

triedenie. Skôr ako ho môžeme popísať však potrebujeme definovať dátovú štruktúru *funnel* (lievik).

## Funnel

$K$ -lievik nazveme štruktúru, ktorá je na vstupe dostane  $K$  usporiadaných zoznamov, s celkovou dĺžkou  $K^3$  a skombinuje tieto prvky do jedného, usporiadaného výstupného zoznamu, pričom použije najviac  $\mathcal{O}(\frac{K^3}{B} \log_{M/B} \frac{K^3}{B} + K)$  pamäťových operácií.

Reprezentácia  $K$ -lievika bude úplný binárny strom s  $K$  listami, uložený v pamäti vo van Emde Boasovom usporiadaní, ako pri statických stromoch (rekurzívne podstromy veľkosti  $\sqrt{K}$ ). Hrany medzi vnútornými rekurzívnymi podstromami si uchovávajú *buffer* (pomocné pole) veľkosti  $K^{3/2}$ , pričom podstromov je  $\sqrt{K}$  a teda spolu potrebujú  $K^2$  pamäte. V podstromoch, ktoré tvoria  $\sqrt{K}$ -lieviky, sú všetky buffery rekurzívne menšie.

Spolu teda  $K$ -lievik potrebuje  $S(K)$  pamäte. Každý sa skladá z  $1 + \sqrt{K}$  podstromov, ktoré reprezentujú  $\sqrt{K}$ -lieviky a teda  $S(K) = (1 + \sqrt{K})S(\sqrt{K}) + K^2$ . Z toho jednoducho dostaneme, že veľkosť  $K$ -lievika v pamäti je  $\mathcal{O}(K^2)$ .

Obrázok + analýza počtu pamäťových presunov

## Funnelsort

Vezmime vstupné pole veľkosti  $N$  a rozdelme ho na  $K = N^{1/3}$  súvislých segmentov. Veľkosť každého bude  $N^2/3$ . Následne rekurzívne utriedime tieto segmenty. Pre ich spojenie použijeme  $K$ -lievik, ktorého výstupom bude usporiadané pole.

Počet pamäťových presunov bude

$$T(N) = N^{1/3}T(N^{2/3}) + \mathcal{O}(\frac{N}{B} \log_{M/B} \frac{N}{B} + N^{1/3})$$

keďže rozdelenie poľa je voči spájaniu zanedbateľné. Táto rekurencia platí pre  $N > M$ . V prípade, že sa celé pole zmestí do cache, teda  $N \leq M$ , a za predpokladu  $M \geq B^2$ , dostávame  $T(N) = T(B^2) = \mathcal{O}(B)$ . Celkové riešenie tejto rekurencie, a teda výsledný počet pamäťových presunov potrebných na usporiadanie poľa veľkosti  $N$  je  $\mathcal{O}(\frac{N}{B} \log_{M/B} \frac{N}{B})$ .