

Pamäťový model

Pri časovej analýze algoritmov sa zvyčajne používa takzvaný *RAM model* (skratka z anglického *Random-Access Machine*, stroj s náhodným prístupom k pamäti), v ktorom sa predpokladá možnosť prístupovať k ľubovoľnému úseku pamäte v konštantnom čase. To znamená, že vo výslednej asymptotickej analýze počítame len počet vykonaných operácií.

V skutočnosti však moderné počítače využívajú niekoľko úrovňovú pamäťovú hierarchiu. Tá sa typicky skladá z registrov a troch úrovní *cache* (vyrovnávacej pamäte) priamo na procesore, následne z hlavnej operačnej pamäte a disku. V tomto poradí sú tieto úrovne zoradené od najrýchlejšej a najmensej (odozva rádovo 1 ns, kapacita 128 KiB) až po najpomalšiu ale najväčšiu (odozva od 100 μs po 10 ms podľa typu¹, kapacita rádovo 1 TiB). Podrobné hodnoty pre všetky úrovne sú v tabulke ??.

referencie

tabulka

Dôsledkom tejto hierarchie je závislosť výslednej rýchlosti algoritmu od jeho prístupu k pamäti. Operácie, ktoré využívajú dáta uložené na disku potrvávajú dlhšie ako tie, ktoré využívajú iba dáta v registroch. V skutočnosti sa tieto dáta postupne presunú z disku do hlavnej pamäte, do *cache* na procesore a napokon do registrov. Následne sa môže požadovaná operácia vykonať rovnako rýchlo ako v druhom prípade, avšak nejaký čas bol algoritmus nečinný a čakalo sa na presun dát. Pre všetky susedné dvojice pamäťových úrovní teda slúži tá menšia a rýchlejšia ako vyrovnávacia pamäť pre tú väčšiu a pomalšiu.

1.1 External-memory model

Spôsobom ako zohľadniť tieto skutočnosti pri analýze algoritmov je takzvaný *external-memory model* (model externej pamäte), nazývaný tiež *I/O model* alebo *cache-aware model*. Ten popisuje pamäť skladajúcu sa z dvoch častí, ktoré voláme *cache* a *disk*.

¹Klasické pevné disky (HDD) alebo disky bez pohyblivých častí (SSD)

Všetky výpočty prebiehajú nad dátami v *cache*, ktorá má obmedzenú veľkosť. Ostatné dáta sú uložené na disku neobmedzenej veľkosti, no nemôžeme s nimi priamo manipulovať a je ich potrebné najskôr preniesť do *cache*. V samotnej analýze algoritmov potom počítame počet týchto prenosov z disku do *cache* a naopak.

Samotné presuny sú realizované v *blokoch* pamäte veľkosti B . Disk aj *cache* sa skladajú z takýchto blokov a za jednu operáciu považujeme presun jedného bloku medzi nimi. *Cache* má obmedzenú veľkosť M a skladá sa teda z $\frac{M}{B}$ blokov.

obrazok

1.1.1 Cache-aware algoritmy

Pokiaľ poznáme parametre B a M , môžeme skonštruovať algoritmus, ktorý bude túto dvojicu pamätí využívať efektívne. Takéto algoritmy voláme *cache-aware* (uvedomujúci si *cache*). Súčasťou tohto algoritmu by bolo explicitné spravovanie presunov pamäte - je potrebné riešiť čítanie blokov z disku a ich umiestňovanie do *cache*, nahrádzanie blokov v *cache* pri zaplnení a spätný zápis blokov na disk.

Tento model tiež popisuje len dve úrovne pamäte a teda funguje efektívne len pre danú susednú dvojicu, pre ktorú ho na základe znalosti parametrov optimalizujeme. V moderných systémoch ale máme takýchto dvojíc niekoľko. Keby sme poznali parametre pre všetky tieto dvojice môžeme zovšeobecniť tento model pre viac úrovní a explicitne riešiť presun blokov medzi nimi. Takto sa však samotná správa pamäte potenciálne stáva komplikovanejším problémom ako pôvodný algoritmus.

1.2 Cache-oblivious model

Riešením týchto problémov je *cache-oblivious model* (na *cache* nedbajúci), v ktorom uvažujeme rovnakú dvoj-úrovňovú pamäť zloženú z disku a *cache*. Na rozdiel od *cache-aware* modelu však algoritmus nepozná parametre B a M . Pokiaľ sa nám napriek tomu podarí navrhnúť algoritmus, ktorý vykonáva (asymptoticky) rovnaký počet pamäťových presunov ako *cache-aware* algoritmus, bude bežať efektívne pre ľubovoľné takéto parametre.

Takéto algoritmy majú na rozdiel od *cache-aware* algoritmov v *external-memory* modeli mnohé výhody. Samotná implementácia algoritmu nemôže explicitne riešiť presun blokov pokiaľ nepozná veľkosť bloku ani koľko blokov môže do *cache* uložiť. Táto úloha zostane ponechaná na nižšiu vrstvu (operačný systém resp. hardware v prípade *cache* na procesore) - algoritmus bude pristupovať k pamäti priamo bez ohľadu na to, či sa nachádza v *cache* alebo nie a v prípade potreby prebehnú nutné prenosy na nižšej úrovni (z pohľadu algoritmu) automaticky.

Ďalšou výhodou je automatická optimalizácia pre dané parametre. V prípade *cache-aware* algoritmov môže byť problémom získať presné hodnoty týchto parametrov a

potrebné pri ich zmene upraviť algoritmus. Vývoj algoritmu, ktorý bude fungovať na rozličných architektúrach, môže byť problematický.

V neposlednom rade bude takýto *cache-oblivious* algoritmus efektívny medzi každou dvojicou susedných úrovní. Vzhľadom na to, že hodnoty parametrov nepozná, musí pre ľubovoľnú takú dvojicu pracovať rovnako efektívne.

1.2.1 Správa pamäte

V momente keď sa *cache-oblivious* algoritmus pokúsi o vykonanie operácie, ktorá potrebuje dáta mimo cache je potrebné ich najskôr z disku skopírovať. V prípade, že je v cache voľný blok tak je možné presunúť dáta bez nutnosti nahradenia. V opačnom prípade je však potrebné uvoľniť miesto tým, že sa vyberie blok z cache, v prípade, že bol upravený sa jeho obsah zapíše späť na disk a následne sa požadovaný blok z disku zapíše na jeho miesto v cache. Tento proces sa nazýva výmena stránok (*page replacement*), a algoritmus rozhodujúci, ktorý blok z cache odstrániť, voláme stratégia 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 v cache môžeme pracovať je $M/B = 1$. 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 plnú adresu na disku, čo by redukovalo celkový počet blokov, ktoré sa do cache zmestia. Znížením asociativity je možné ukladať iba časť adresy, pričom zvyšok je implicitne určený pozíciou v cache. V prípade nízkej asociativity však 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. Druhý predpoklad je však nerealizovateľný, keďže by stratégia výmeny stránok musela predpovedať budúce kroky algoritmu. Nasledovné lemy však ukazujú, že 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]*

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]*

Stratégia *LRU* (least recently used) vyberá vždy blok, ktorý bol najdlhšie nepoužitý. Implementácia vyžaduje udržiavať si ku každému bloku počítadlo, ktoré sa pri prístupe nastaví na nulu a pri prístupe k iným blokom zvýši o jedna. Pri potrebe uvoľniť miesto v cache vyberieme blok s najväčšou hodnotou počítadla - ten, ku ktorému najdlhšie nebol prístup.

Stratégia *FIFO* (first in, first out) je ešte jednoduchšia - bloky sú udržiavame zoradené podľa poradia ich vloženia do cache. Keď vyberáme blok na odstránenie tak vezmeme ten, ktorý bol pridaný najskôr a teda je na začiatku tohto usporiadania.

Cache-oblivious algoritmy a dátové štruktúry

Intro text - obsah kapitoly (alg/ds, analiza, ...)

2.1 Základné algoritmy

Na demonštráciu *cache-oblivious* algoritmov a ich analýzy v *external-memory* modeli použijeme jednoduchý algoritmus, ktorý počíta agregáčnú funkciu nad hodnotami uloženými v poli.

2.1.1 Popis algoritmu

Majme pole A veľkosti $|A| = N$ a označme jeho prvky $A = \{a_1, \dots, a_N\} \in X^N$. Chceme vypočítať hodnotu $f_g(A)$, kde $g : X \times Y \rightarrow Y$ je agregáčná funkcia, $g_0 \in Y$ je počiatočná hodnota a $f_g : X^\infty \rightarrow Y$ je rozšírenie agregáčnej funkcie definované nasledovne:

$$\begin{aligned} f_g(\{a_1, \dots, a_k\}) &= g(a_k, f(\{a_1, \dots, a_{k-1}\})) \\ f_g(\emptyset) &= g_0 \end{aligned}$$

Túto funkciu je možné implementovať jednoducho ako jeden cyklus. Schematickú verziu implementácie uvádzame v algoritme 2.1.

Algoritmus 2.1 Implementácia agregáčnej funkcie f_g

```

1: function AGGREGATE( $g, g_0, A$ )
2:    $y \leftarrow g_0$ 
3:   for  $i \leftarrow 1, \dots, |A|$  do
4:      $y \leftarrow g(A[i], y)$ 
5:   return  $y$ 

```

Správna notácia pre x infinity?

Tento algoritmus s použitím vhodnej funkcie g a hodnoty g_0 je možné použiť na rôzne, často užitočné výpočty, ako napríklad maximum, minimum, suma a podobne:

$$\begin{aligned} g^{\max}(x, y) &= \max(x, y) & g_0^{\max} &= -\infty \\ g^{\text{sum}}(x, y) &= x + y & g_0^{\text{sum}} &= 0 \end{aligned}$$

2.1.2 Analýza zložitosti

Časová analýza

Klasická časová analýza tohto algoritmu je triviálna ak uvažujeme *RAM model*. Keďže prístup ku každému prvku $A[i]$ zaberie konštantný čas a za predpokladu, že čas na výpočet funkcie g , T_g , je nezávislý na vstupe, bude výsledný čas na výpočet tejto funkcie

$$T(N) = \mathcal{O}(1) + N[\mathcal{O}(1) + T_g + \mathcal{O}(1)] = T_g \cdot \mathcal{O}(N)$$

Pamäťová analýza

V prípade *cache-aware* algoritmu by sme pole A mali uložené v $\lceil \frac{N}{B} \rceil$ blokoch veľkosti B . Pri výpočte 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, $\lceil \frac{N}{B} \rceil$. Tento algoritmus však požaduje znalosť parametra B a explicitný presun blokov.

Jednoducho však vieme dosiahnuť (takmer) rovnakú zložitosť aj v prípade *cache-oblivious* algoritmu 2.1, ktorý žiadne parametre pamäte zjavne nevyužíva a nepozná. Budeme predpokladať, že pole A je uložené v súvislom úseku pamäte - to je možné dosiahnuť aj bez znalosti parametrov pamäte. 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 $\lfloor \frac{N}{B} \rfloor$ plných blokov a nakoniec ešte najviac jeden blok, ktorý opäť nie je plný. Spolu máme teda $\lfloor \frac{N}{B} \rfloor + 2$ blokov.

Pokiaľ $\lfloor \frac{N}{B} \rfloor < \lceil \frac{N}{B} \rceil$ máme spolu najviac $\lceil \frac{N}{B} \rceil + 1$ blokov. V opačnom prípade B delí N , teda v prvom a poslednom bloku je spolu presne B prvkov a medzi nimi sa nachádza najviac $\frac{N-B}{B} = \frac{N}{B} - 1$ plných. Teda blokov je vždy najviac $\lceil \frac{N}{B} \rceil + 1$.

Zostrojili sme teda *cache-oblivious* algoritmus s asymptoticky rovnakou zložitosťou $\mathcal{O}(\frac{N}{B})$ ako optimálny *cache-aware* algoritmus, ktorého implementácia je však jednoduchšia, keďže nemusí explicitne spravovať presun blokov do cache.