KAPITOLA 1

Cache-oblivius algoritmy a dátové štruktúry

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

1.1 Základné algoritmy

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

1.1.1 Popis algoritmu

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

$$f_g(\{a_1, \ldots, a_k\}) = g(a_k, f(\{a_1, \ldots, a_{k-1}\}))$$
$$f_g(\emptyset) = g_0$$

Spravna notacia pre x infinity?

Túto funkciu je možné implementovať jednoducho ako jeden cyklus. Schematickú verziu implementácie uvádzame v algoritme 1.1.

Algoritmus 1.1 Implementácia agregačnej funkcie f_g

```
1: function AGGREGATE(g, g_0, A)
2:     y ← g_0
3:     for i ← 1, ..., |A| do
4:         y ← g(A[i], y)
5:     return y
```

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:

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

1.1.2 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

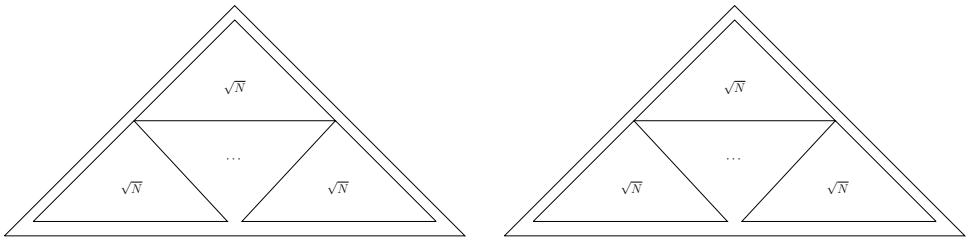
V prípade *cache-aware* algoritmu by sme pole A mali uložené v $\lceil \frac{N}{B} \rceil$ blokoch veľkosti B. Pri výpočte by sme postupne tieto bloky načítali do cache a pracovali s nimi. V rámci jedného bloku počas výpočtu nedochádza k pamäťovým presunom. Zároveň stačí každý prvok spracovať raz a teda 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 1.1, 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.



(a) Abc (b) Def

Obrázok 1.1: Test

1.2 Vyhľadávacie stromy