Algoritmy a jejich implementace

2. ledna 2013

Obsah

1 Účel přednášky

Jak převést efektivní algoritmus do praxe tak, aby opravdu běžel rychle.

1.1 Příklad – sčítání matice

Paměť je pomalejší než cache. Vyplatí se po řádcích, aby nenačítal tak často nové stránky.

Kopírování paměti je rychlejší, než forcyklové vyplňování. Lze použít i mmap (je spožděný).

2 Jazyk C

Používáme normu `C99`.

`switch` je jen `goto` na proměnlivé návěští. Lze prokládat jinými věcmi, tedy např. `while` cyklus.

2.1 Typy

Základní:

- Celočíselné typy, jako `int`, `char`, různě velké, znaménkové či neznaménkové, `_Bool`. `C99` už říká i požadované velikosti (`char` ≥ 8, `short, int` ≥ 16, `long` ≥ 32, `long long` ≥ 64). Dále `unsigned char` je nejmenší adresovatelná buňka paměti.

  Na `unsigned` typech funguje aritmetika modulo nějaké číslo. Existují tam bity, které jsou součástí binárního zápisu, smí tam být i něco navíc, mohou mít libovolné permutace bitů.

- Typy na plovoucí čárky (`float`, `double`, `long double`). Mimo to zavádí klíčové slovo `_Complex`, které když se napíše za jméno typu, tak se stane komplexním.

- Divný typ `void` – neobsahuje žádnou hodnotu, např. když funkce nic nevrací, ukazatel do nikam.

- Výčtové typy se tváří také být základní, je to celočíselný typ.

Odvozené:

- Ukazatel. Nelze přetypovávat libovolně. Smí se přetypovávat na `unsigned chary`.

Je zde speciální `void *`, je typově kompatibilní s libovolným ukazatelem.

- Pole, v paměti jsou naskládané souvisle za sebou. Chová se k nim skoro stejně, jako k ukazateli na jeho začátek.
- Struktury a uniony. Jsou to nějaké různé položky, struktura je má za sebou, union přes sebe. Může mít paddingy mezi prvky. Dodržuje pořadí a padding závisí pouze na věcech nad tím a nikdy nesmí být před prvním prvkem. Lze použít při dědičnosti.

Typy mohou být kompletní a nekompletní. U nekompletních lze vytvářet ukazatele, nesmí se deklarovat proměnná toho typu, nesmí se `sizeof`.

Typ lze upřesňovat, i postupně (např. velikosti polí).

2.1.1 Jména typů

Jsou výrazy. `int *f(void *x[])` – když vezmu f, zavolám ho s parametrem x (který, když vezmu, zaindexuji a zdereferencuji, dostanu void), to zdereferencuju, tak dostanu int. Tedy to musí být funkce, která bere pole ukazatelů na void a vrací ukazatel na int.

2.1.2 Obskurdnosti

- **Bitová pole** – intové typy, které mají danou velikost v bitech, slučují se dohromady.

```
struct {
  int x:5;
  int y:3;
};
```

- **Flexibilní pole** – na konci struktury může být pole neznámé délky.

2.1.3 Kvalifikátory

- `const` – Objekt, o kterém se mluví, je kontstantní. Záleží na jeho poloze, čímž určuje, co je konstantní.
- `volatile` – Zakazuje optimalizace na objektu.
- `restrict` – Říká o něčem, že je to jediný ukazatel na daný objekt (nebo, jediný aktivní), aby překladač mohl optimalizovat.

2.1.4 Storage class

Určují, jak je to uložené.

- `auto` – Žijí lokálně a vidět jsou také lokálně. Tak se chovají lokální proměnné, když nic nemají. Nesmí se použít na globální.
- `extern` – Nežije to tady, je to viditelné globálně, žije stále.
- `static` – Je lokální pro daný blok/modul a žije celou dobu.
- `register` – Doporučuje žít v registru. Nejde na to vzít ukazatel a podobně, jinak je stejná jako auto.
- `typedef` – Je to typ.

Na parametrech smí být pouze `register`.

U funkce, `static` znamená, že žije jen lokálně. Pokud se definuje, tak je to globálně viditelné, jen při deklaraci se chová jako `extern`. Smí se použít explicitně.

O funkci lze říct ještě `inline`, doporučuje nevolání, ale vkládání.

2.1.5 Inicializátory

Kompletuje datové typy (např. určuje délku pole, u kterého se to zatím nevědělo).

Umí být pojmenované, jsou neúplné výrazy.

```
struct i item = {
  .a.i = 0,
  .x = 2
};
```

Co je neinicializované, tak se defaultuje na nulu, což je rozdíl oproti bez inicializované, kde to nula být nemusí.

2.2 Výrazy

Většina věcí jsou výrazy. Kromě výsledky mohou mít i vedlejší účinky.

Při pořadí počítání je to definované, tedy `a+b+c`, uzávorkuje se zleva a např. u floatů nemusí být asociativní.

Ale side effecty nemají dané pořadí, jen říká, že vše proběhne do nejbližšího sequence pointu.

Sequence pointy jsou:

- Konec příkazu
- Čárka jako operátor (ne parametry!)
- Booleovské `&&`, `||`
- `?`
- Vstup a výstup z funkce

U některých slibuje, že se něco nevyhodnotí (u booleovských podmínek, otazníku).

2.2.1 Literály

Např. integerové (čísla v různých soustavách, mohou mít datové typy).

Znakové konstanty jsou typu `int`. Stringové se dají spojovat, např. `"str""ing"`. Široké znaky (`wchar_t`), např. `L'ž'` či `L"kůň"`. Unicodové escapy (`L'\u2302'`) se mohou vyskytovat i v identifikátorech.

Literál daného typu se dá udělat syntakticky přetypováním iniciátorů (`(const char []){ 1, 2, 3 }`).

2.2.2 Operátory

Pár poznámek:

- `~` je definován jen na unsigned.
- `&*` je identita, i kdyby ten ukazatel nechtěl fungovat (byl by `NULL` či podobně).
- Pointerová aritmetika funguje po velikosti toho, na co se ukazuje. Smí se ukazovat na prvky pole a jeden za.
- `a[b]` je totéž jako `*(a+b)`.
- Bitové posuny se smí dělat jen do velikosti typu (např. o 31 na 32 bitovém integeru).
- Přiřazovací operátory se závorkují zprava.

Chytáky:

- `x&1 == y&1` je totéž jako `x&(1 == y)&1`.
- `1 << 4 + 1 << 5` je totéž jako `1 << (4 + 1) << 5`.

2.2.3 Konverze

- Pole → ukazatel (mimo `sizeof` a `&`).
- Funkce → ukazatel na fci (mimo `sizeof` a `&`).
- 0 a `NULL` jsou zaměnitelné.
- Integery menší než `int` na `int` (pro `unsigned` obdobně).

Konverze podle počítání, z integeru na float, když je alespoň jeden float a tak podobně.

Konverze při přetypování nebo při přiřazení.

2.2.4 Deklarace funkcí

Vynechané parametry – neříkám o parametrech nic. `...` – dále kolik chce libovolných parametrů.

Pozor, pole se předávají odkazem, i když se zadají velikosti a nedá se vracet.

Pokud mám vícerozměrné pole, pak jde i toto:
`void sum(int n, int matice[n][n]);`, aby bylo vědět, jak velký je řádek toho pole.

Kvalifikátory mohou být uvedeny v hranatých závorkách:
`int cmp(int x[restrict], int y[restrict]);`

2.3 Preprocesor

Pozor, existují trigraphy, např. ??/ je backslash. Nahrazuje se všude, dá se nachytat.

Dělá podmíněné překlady, nahrazuje definice.

V includu se provádí expanze maker.

Ochrana proti násobnému vložení:

```
ifndef BLABLA_H
define BLABLA_H
```

```
...

endif
```

Direktiva `#pragma` je implementačně závislá, ovládá překladač. Podobně se chová i `_Pragma`, ale lze se použít kdekoliv.

V makrech se občas hodí konstrukce `do { ... } while(0)`.

Makra s proměnlivým počtem argumentů se dělají tak, že se do závorek napíše trojtečka a použije se `__VA_ARGS__`.

Makro pracuje na tokenech, tedy pokud mám makro `E`, pak `12E3` je něco jiného než `12 E 3`.

2.4 Standardní knihovna

Věci jako `stddef.h`, `stdlib.h` a podobně.

2.4.1 `limits.h`

Např. `CHAR_BIT` je počet bitů, různé `INT_MAX`.

2.4.2 `stddef.h`

`ptrdiff_t` je rozdíl dvou ukazatele, `size_t` je výsledek `sizeof`, `NULL`, `offsetof()` zjišťuje offset prvku ve struktuře.

2.4.3 `stdbool.h`, `stdint.h`

Např. typy `int16_t` (přesně 16 bitů), `int_least16_t` (nejmenší alespoň 16 bitový), `int_fast16_t`. Dále `intmax_t` (největší integerový) a `intptr_t` (vejde se tam ukazatel).

2.4.4 `inttypes.h`

Formátovací sekvence pro `printf` – např. PRId16, případně `SCN*` pro scanf.

2.4.5 `stdlib.h`

Funkce `atexit()`, registrují „cleanup handlery“.

Také makra `EXIT_SUCCESS` a `EXIT_FAILURE`.

`qsort()`, `bsearch()`.

2.5 `string.h`

Např. `strcspn()`.

3 Paměti

3.1 Základní model

Má procesor připojený ke sběrnici. Na ní žijí i další zařízení a řadič paměti. K němu je připojená paměť. Také je možné, aby zařízení komunikovaly s pamětí přímo, ale to lze zanedbat.

Paměti existují paměti statické a dynamické. Statická je klopný obvod, je velká, ale velmi rychlá. Dynamická je založená na kondenzátorech, je menší, ale potřebuje nabíjení (je třeba znovu zapsat) a je pomalejší (asi o 2 řády než procesor).

Proto se dělají hlavní (velké) paměti z dynamických a přidávají se cache z statických.

3.1.1 Plně asociativní cache

Několik řádků, v každém adresa a data, umí to přímo odpovědět, kde je v cache daná adresa. Obvykle se na vyhazování používá LRU (Least Recently Used), nejstarší vyhazovat.

Pokud je cache typu write-through (zapíšu do obou), nebo častěji, write-back cache. V cache si pamatuji, že některá data byla změněná, když není co dělat nebo vyhazuji změněnou položku, tak ji zapíšu.

3.1.2 Optimalizace řádků

Obvykle se cachuje po celých řádcích, jeden řádek má velikost většinou 32 nebo 64 bajtů. Nepotřebuji si potom v cache pamatovat všechny bity z adresy. Paměti jsou na toto optimalizované a umí číst celé řádky skoro stejně rychle.

Pro zápis osamostatněného bajtu je třeba napřed řádek načíst, zápisy jsou proto pomalejší.

3.1.3 Přímo mapovaná cache

Asociativní paměti se vyrábějí těžko, proto se každá adresa ukládá na předem dané místo. Většinou se používá daná část adresy.

Dochází ke kolizím, říká se tomu cache aliasing.

3.1.4 Množinově asociativní cache

Má hashovací funkci, v každé přihrádce je jedna malá plně asociativní cache – k-cestně asociativní (mám v každé přihrádce k řádků).

Protože je problém udělat cache, která je zároveň dostatečně velká a dostatečně velká (i kvůli přenosu k procesoru), používá se víceúrovňová hierarchie (čím blíže, tím menší ale rychlejší).

Jsou hierarchie inkluzivní (co je v některé, je i ve všech větších) nebo exkluzivní (co je v jedné nesmí být jinde).

Občas bývají 2 `L1` cache (na instrukce a na data). V instrukčí mohou být nápovědy.

3.1.5 MMU

Na jedné straně je adresní prostor každého procesu a na druhé se nachází fyzický adresní prostor. Obvykle překládá po stránkách (většinou pevné velikosti). Dnes například `4kB`. Každá stránka může mít práva, vlastníka, flagy a podobně.

Obvykle se používá strom na vyhledávání překladu, každá ukládá, kde je další tabulka nebo vlastní stránka. Je třeba mít v procesoru uloženou kořene.

Některé procesory umí i velké stránky.

Například 32bit vypadá tak, že adresa je rozdělená na 10, 10, 12. Napřed se podívá do kořene, ve kterém je 1024 položek. Tam může být buď odkaz na $4MB$ stránku, nebo na stránku s dalšími 1024 položkami.

Dále máme `TLB` (Translation Lookup Buffer) – cache pro už resolvované adresy stránky. Obvykle jsou množinově asociativní a 2-úrovňové.

Někdy se používá jen `TLB`, zbytek se dělá softwarově.

Když by byla cache s fyzickými adresami, je mnohem méně problémů. Problém je, že v době, kdy chci hledat, tak ještě nemám hotový překlad. Proto se používá hybrid, index (ta hashovaná část) je virtuální, tag (ten asociativní prefix) je fyzický. Navíc se starám o to, aby tam nebyla jedna fyzická stránka dvakrát.

`L2` už bývají adresované čistě fyzicky.

3.2 Parametry RAM

Dram je matice, má řádkový buffer ve statické paměti. Vždy se přečte řádek, tím se smaže, když už není potřeba, tak ho vrátí (a obnoví) a přečte jiný.

Obvykle se používá hodinový signál na komunikaci. V každém tiku se posílá data tam i zpět. Můžu poslat adresu a příkaz, můžu vyměňovat data. Příkazy jsou:

- **NOP** – Některé příkazy trvají déle, vyplňování.
- **Active** – Načtení řádku do bufferu. Předává adresu řádku.
- **Read** – Přečtení dat z aktuálního řádku. Může přečíst i více, může být nakonfigurovaná na čtení po dávkách v různě velkých kusech.
- **Write** – Opak read.
- **Precharge** – Vrátí zpátky řádek.
- **Setmode** – Např. nastavení velikosti dávky.
- **Refresh** – Obnoví jeden, nejstarší, řádek.

Obvykle jsou omezení na minimální čas čekání po Active, reakce na Read, časy mezi Active a Precharge oběma směry.

Dnes se používá něco vylepšeného, DDR.

DDR umí přenést dvě slova za jeden takt (na náběžné i shazovací hraně signálu). DDR2 má dvakrát rychlejší sběrnici, než paměť. DDR3 má čtyřikrát rychlejší sběrnici, než paměť.

Rozšíření jsou, že lze říct, kterým slovem začít dávku (cyklicky).

Jsou tam oddělené banky, při čekání jednoho můžu dělat něco s jiným. Dále umí automatický precharge, je parametr readu. Také umí delayed read.

4 Rozsíření GCC

Existuje jich hodně, tady jsou některé často používané:

- **Příkazové výrazy** – define, který má proměnné, začíná `({`, pak je blok, poslední výraz je návratová hodnota a ukončené pomocí `})`.

```
define MIN(a,b) ({ \
  int _a = (a), _b = (b); _a < _b ? _a : _b; })
```

- Vnořené funkce, vidí lokální proměnné. Jdou dokonce dávat ukazatele na ni.
- `a ?: b`
- Binárně zapisované literály (`0b00100010`).
- `case 1...5:`
- Vkládání assembleru: `asm("KUS ASM":Výstup:Vstup)`.
- Atributy funkcí – `__attribute__((noreturn))`.
- Zabudované funkce, např. `__builtin_expect` – říká, že očekávaná hodnota výrazu je nějaká.

5 Profilování

Procesor obsahuje registry. `EAX` až `EDX` jsou na obecné použití, jsou menší varianty a poté nějaké speciální.

Má také flagy, udávají stav procesoru. Například:

- Sign flag (znaménko)
- Zero flag (poslední vyšla nula)
- Carry flag (přetečení)

Máme také zásobník, jsou na manipulaci instrukce a registry (`ESP` ukazuje nakonec, `EBP` za návratovou adresu). Ukládají se sem funkce a jejich lokální proměnné. Na intelu roste směrem dolů.

Příkaz `objdump` umí například disassemblovat. Assembler lze získat také přímo z gcc.

Proměnná `MALLOC_CACHE_` – potom kontroluje, že se nevolá free dvakrát a podobně.

Program ElectricFence odchytává přístup za alokovanou paměť a podobně.

Nejmocnější je asi valgrind.

- memcheck – kontrola paměti
- helgrind – zamykání paměti
- massif – grafy, kolik je paměti spotřebované
- cachegrind – cachemissy

- callgrind – využití cache, profilování kódu

Dále lze přidat flagy `-fprofile-generate` a `-fprofile-use`.

5.0.1 Profilovací nástroje

Např. `gprof` (jednou za nějakou dobu se podívá, která funkce zrovna běží), `oprofile`, který k tomu používá kernel a HW podporu.

AMD umí něco zvané Instruction-Based Sampling. Sleduje vždy jednu instrukci při celém jejím zpracování a zaznamenávají se její události (např. cache-miss, počet tiků).

6 Více procesorů

Model: procesory a paměťový řadič připojený ke sběrnici (SMP). Každý má nějakou cache. Problém je, že je potřeba cache synchronizace. Lze řešit pomocí toho, že procesory odposlouchávají sběrnici.

Write-back cache lze vyřešit tak, že max. 1 procesor má data nacachovaná, v případě, že jich potřebuje více, tak je write-trough.

Stavy řádků:

- Invalid (nic)
- Shared (aktuální data, má je i někdo jiný)
- Exclusive (aktuální data, mám je já)
- Modified (aktuální data, která nejsou v RAM, mám je jen já)

Toto je třeba udržovat v průběhu výpočtu u každého řádku, v případě čtení může jiný zasáhnout a říct, že to mám já. Pokud se sdílí řádek při zápisech, je to pomalé, pokud mají každé svoje a nelezou si do zelí. Druhý problém je, že je to pomalé tak, jak je pomalá paměť – např. když jeden čte změněný řádek, musí se počkat, až předchozí zapíše.

Vylepšeno – nový stav „Owned“ – data nejsou v hlavní paměti a patří aktuálnímu procesoru. Rozšířený stav „Shared“, takhle někomu patří a musí je na konci zapsat.

6.1 Reálná situace

Procesor je připojen přes `FSB` (Front Side Bus) k NorthBridge, ten už se stará o komunikaci s rychlými zařízeními (`DRAM`, `PCIe`). Dále je k němu připojen ještě SouthBridge, ten komunikuje s pomalými zařízeními (porty, karty).

AMD má místo `FSB` něco, čemu říká HyperTransport, paměti přímo na procesoru. HyperTransport umí větvení do stromu.

V případě, že je více procesorů, tak v prvním případě jsou připojeny k NorthBridgi. Umí se to až se 4 procesory.

U AMD má každý procesor více hypertransportů, propojují se mezi sebou. Paměti jsou lokální pro procesor, tedy jsou různě rychlé (záleží, jestli je lokální).

Vyrábí se také vícejádrové procesory (více procesorů na jednom čipu). Každé jádro má vlastní L1 cache, ostatní cache již někdy bývají společné. Z pohledu programu se chová podobně, jako oddělené procesory, jen je trochu komplikovaný čas na přístup k datům jiných procesorů. Paměťový řadič mívají společný.

Existuje také HyperThreading – řídící část je tam dvakrát, ale výkonné jednotky sdílí. Bohužel to nefunguje, vyhazují si data z L1 cache.

6.2 Programování paralelizmů

- Více procesů. Jednoduché, samo se to vyváží, ale málo spolu komunikují. Nehodí se, pokud je potřeba přidělovat dynamicky nebo když mají nějaká sdílená data.

  Možnost vyřešit sdílenou pamětí.

- Vlákna, což jsou v podstatě jen procesy se sdíleným adresním procesem. Problém napsat správně, je potřeba synchronizovat.

6.2.1 Synchronizační primitiva

První je Mutex (Mutable Execution). Lze to zamknout a zamknuté to smí mít max. jedno vlákno. Zamknutí čeká, dokud není odemčeno.

Verze je RW-mutex, který dovoluje mít zamčeno pro čtení u více procesorů.

Pak jsou semafory – podobně, jako mutex, ale umí počítat a jeho hodnota je vždy nezáporné číslo. Up zvýší o 1, down sníží o 1, čeká, pokud to nejde.

Tyto operace jsou drahé, protože nejdou cachovat.

Lze implementovat nějaké rozdělování práce např. pomocí fronty. Ta jde implementovat pomocí mutexu a semaforu.

6.2.2 Vyhýbání se synchronizačním primitivům

Existují atomické operace, které umožňují vyhnout se takovýmto věcem. Je to nebezpečné, mohou se prohazovat zápisy a podobně.

6.2.3 Atomicita syscallů

Některá systemová volání jsou atomická. Například zápisy do souboru.

6.2.4 Nástroje

Například lze použít OpenMP, které přidává některá `#pragma` instrukce. Ty pak dovolí paralelizovat kusy kódu.

6.3 Paralelizace mezi počítači

Lze paralelizovat ještě přes více počítačů. Je třeba zaručit ještě rozumnou komunikační složitost – kolik dat se přenáší.

6.4 Vektorové instrukce

Například `SSE` (Streaming SIMD Extension). Pracují se 128 bitovými registry. Dá se s nimi poté pracovat jako s horou čísel za sebou.

Například `PADDW` – sčítání 16-bitových integerů, `PMINB` – minimum z bajtů, či porovnávání (`PCMPEQB`). Dále tu jsou permutační instrukce.

Při programování lze používat buď GCC buildiny, tedy např: `int __attribute__(vector(16))`, pak je to 128 bitů rozdělených na 16. bitové integery.

Případně to může `<emmintrin.h>` dává lepší jména.

7 Procesory

Kroky:

- Načtení instrukce
- Dekódování instrukce
- Získání parametrů
- Vykonání
- Zapsání výsledků

Snaží se to paralelizovat.

Od 486 existuje pipelina, někdy musí počkat na spočítání předchozí instrukce. Od pentia se provádí více instrukcí zároveň (paralelní execuční jednotky). Začíná odhadovat skoky.

Pro hádání skoků se používaly statické heuristiky a dynamické odhady.

Mívají return stack, pro případ skoku návratu, loop predictor.

Používají se virtuální registry a má k dispozici více exekučních jednotek.

8 Paralelní výpočty

8.1 Cell

Základní, kontrolní, jádro je PPC (rozumí instrukcím) včetně rozšíření altivec (vektorové instrukce). Má velmi jednoduché jádro, například nemá out-of-order pipeline, je třeba optimalizovat kód. Normálně skrz cache připojený do paměti.

Má také tzv. „synergic processing elements“, obvykle 8. Ke každému je připojený jeden „local store“ (256 kb), celé připojené přes sběrnici „EIB“.Každé toto SPE má 128 128 bitových registrů, má vlastní (očesanou) instrukční sadu, vektorové instrukce odpovídající altivecu. Local store je explicitní.

8.2 GPU

Grafická karta lze zneužít i k tomu, aby počítala něco, co se přímo nezobrazuje. Lze využít například tak, že framebuffer nemusí být na obrazovku, ale i na texturu. A výpočty lze trochu programovat (tzv. shadery). Nějaká funkce se pustí na každý pixel výsledku, takže lze využít k paralelním výpočtů. Branche je možné simulovat pomocí Z-bufferu.

Je trochu šílená inicializace a podobně. To se snaží vyřešit OpenCL, ale ještě není implementovaný.

9 Cache-oblivious algoritmy

Způsob, jak navrhovat algoritmy tak, aby nezávisely na parametrech cache, ale používaly ji dobře.

Pokud se podíváme na návrh, kdy jsou data ve vnější paměti. Je vnitřní a vnější paměť, vnitřní má velikost M, vstup a výstup probíhá po blocích velikosti B. Přibude vstupně-výstupní složitost (výstup lze zanedbat, protože to někdy budu muset určitě přečíst).

Tento model lze používat na modelování cache a paměti, vnější paměť je RAM, vnitřní je cache, bloky jsou řádky cache. Předpokládejme, že cache se chová zcela optimálně. Potom IO model dělá horní odhad, kolik dat proteče.

Pokud algoritmus funguje bez znalosti M a B jen konstantně-krát pomalejší než s jejich znalostmi, pak je cache-oblivious.

Takový algoritmus je například scan pole (každou věc jednou přečteme, čte se postupně, když nezná parametry cache, tak se mu může stát, že bude neúplný blok na obou stranách, takže přečteme o max. 1 blok více). Stejně tak otočení pole (pokud $M \geq 2B$).

9.1 Násobení matic

Když budeme násobit matice a budeme mít matice uložené „obvyklým“ způsobem po řádcích, tak bude z druhé matice vyhazovat data (čtu sloupeček).

Pokud budu mít ale první matici po řádcích a druhou po sloupcích, jsem snížil počet čtení z N^3 na $\Theta\left(\frac{N^3}{B}\right)$.

Často pomáhá metoda „rozděl a panuj“, protože od určité velikosti se už problém vejde do cache. Zde je to velmi podobné strasserovu algoritmu. Hodí se ale, aby se menší problém byl vždy pohromadě. Jednoduše se (rekurentně) ukládá po blocích/čtvrtinách.

$$\begin{pmatrix} AB \\ CD \end{pmatrix}$$

.

Bude to uložené jako $ABCD$. Poté je každá část vždy pohromadě. Celkově to vyjde $O\left(\frac{N^2}{B} + \frac{N^3}{\sqrt{M}\cdot B} + 1\right)$. Pokud se udělá optimalizace ze strasserova algoritmu, tak prostřední člen bude $\frac{N^{\log_2 7}}{\sqrt{M}\cdot B}$.

Pokud by nebylo použito toto uložení, vzniká horší odhad.

Obvykle se uvažuje, že počet řádků cache je alespoň tolik, kolik je velikost cache (říká se tomu ***štíhlá cache***).

Tedy, pokud by bylo normální uložení matice, tak pro štíhlou cache bude stejný.

9.2 Vyhledávání v setříděných datech

Pokud uděláme binární vyhledávání, tak „skáčeme“ a v ničem to nepomůže. Mohli bychom udělat $B-$strom. Problém je, že neznáme B.

Binární strom lze přeskládat tak, aby se dobře cacheoval a nebo můžeme udělat $\sqrt{N}--$strom, ale klíče budou ne lineárně, ale opět v takovémto stromu (říká se tomu van-ende-boasovo uspořádání, něco takového je popsané v grafových algoritmech). V tomto případě potřebujeme tolik přístupů,

kolik je délka cesty až dolů děleno výškou jednoho stromu, který se vejde do B. (Vyjde to $4 \cdot \log_B N$).

9.3 Reálná situace

- Strategie není optimální
- Více úrovní cache
- Není plně asociativní
- Někdy to má ruční obsluhu

Mějme posloupnost požadavků P, C jsou počty čtení a N jsou velikosti cache. Potom:

$$C_{LRU} = O\left(C_{OPT} \cdot \frac{N_{LRU}}{N_{LRU} - N_{OPT} + 1}\right)$$

Pokud vezmeme 2· větší cache bude LRU jen konstanta-krát pomalejší. Vzít dvakrát větší cache není problém, protože všechny tyto algoritmy závisí na velikosti cache „rozumě“ (ztratí se to v O).

Tyto algoritmy se chovají optimálně ke každé cache zvlášť a pokud je cache hierarchie navržená dobře, tak se chová optimálně k celku.

Na zbylé dva body lze postavit redukce, které to převádí, ale potřebují vědět už vlastnosti cache.

Věta 1 *Pro libovolnou posloupnost požadavků je:*

$$\#LRU = O\left(\#OPT \cdot \frac{n_{LRU}}{n_{LRU} - n_{OPT} + 1}\right)$$

Důkaz:
Na prvním požadavku je to nastejno.

Pokud v S_i `LRU` mine na 2 stejných požadavcích, tak proběhlo alespoň $n_{LRU}+$ 1 různých požadavků. OPT minul alespoň $n_{LRU} - n_{opt} + 1$ krát.

Pokud `LRU` mine na p, v S_i je $\geq n_{LRU}+1$ různých stránek, `OPT` mine alespoň $n_{LRU} - n_{OPT} + 1$.

Jinak LRU mine na n_lru různých stránkách navíc různých od p, `OPT` mine alespoň na $n_{LRU} - n_{OPT} + 1$.

9.4 Třídění

Když uděláme n-cestný merge-sort ($n = \frac{M}{B}$). Potom:

$$R(n) = \Theta\left(\frac{n}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B}\right)$$

To je ale pro případ, kdy známe parametry.

Uděláme si ***k-trychtýř***. Ten bude slévat k posloupností o celkové délce k^3 v čase $O(k^3 \cdot logk)$ a prostoru $O(k^2)$ a počtem přístupů $O\left(\frac{k^3}{\log \frac{M}{B}} \frac{k^3}{B} + k\right)$.

Potom uděláme trychtýřové třídění, což bude rekurzivně třídit za pomoci $n^{\frac{1}{3}}$-cestného slévání pomocí trychtýře.

$$R(n) = n^{\frac{1}{3}} \cdot R(n^{\frac{2}{3}} + O\left(\frac{N}{B} \cdot \log_{\frac{M}{B}} \frac{n}{B} + n^{\frac{1}{3}}\right)$$

Rekurzi zastavíme, když třídíme bloky B^2, které už se vejdou do štíhlé cache, načteme na $O(B)$ přístupů.

Po upočítání vyjde, že pokud máme k dispozici trychtýř, umíme to stejně rychle, jako se znalostí parametrů cache.

Algoritmus 1 (Trychtýř):

Myšlenka je taková, že trychtýř je strom, který má na hranách buffery. Listy načítají z vstupních souborů, výstup padá z kořene. Vrcholy slévají ze svých synů do bufferu nad sebou, když něco dojde, tak zavolají rekurzivně „naplnění“ na synovi.

Uděláme na tom van-ende-boassovu reprezentaci – nahoře je jeden $\sqrt{k}$-trychtýř, pod nimi jsou další $\sqrt{k}$-trychtýře. Spojeno je to buffery velikosti $k^{\frac{3}{2}}$.

Po upočítání zabere $O(k^2)$ paměti, časová složitost je $O(k^3 \cdot \log_k)$.

Pro výpočet počtu přístupů vybereme j takové, že zabírá $\leq \frac{M}{4}$, ale větší už se nevejde. Tedy j bude někdy mezi $\sqrt{M}$ a $M^{\frac{1}{4}}$

Předpokládáme štíhlou cache.

Nahrání jednoho trychtýře bude trvat $O\left(\frac{J^2}{B} + J\right)$ (J za první bloky bufferů v listech). To se udokazuje, že vyjde $O(\frac{J^3}{B})$.

Když dojdou data a spustíme jiný trychtýř, tak přijdeme o tento v cache. Ale do bufferu už přijde dostatek dat (J^3 prvků).

Po upočítání vyjde něco, co je potřeba upočítat, že vyjde stejně, jak chceme.

☺

9.5 Dynamické struktury

Budeme stavět datovou strukturu, která bude umět přidat kamkoliv, odebrat odkudkoliv a projít sekvenčně. Uděláme to jako řídké pole (mohou některé části chybět).

Algoritmus 2 (Dynamická datová vyhledávací struktura):

Potom z toho půjde pomocí van-ende-boass stromu postavit struktura, která umí dělat jak stromové hledání, tak úpravy a sekvenční procházení.

Představíme si strom nad prvky, udává intervaly povolené hustoty pole. Tyto intervaly se „zpřísňují“ směrem ke kořeni.

Invariant může neplatit, když se to nepotká (nikdo si toho nevšimne). Když najdeme nějaký, pokračujeme v hledání nahoru, než narazíme na jeden, který je v pořádku. Potom přebudujeme tu část, kterou potřebujeme.

☺