

Datové struktury I

4. přednáška: Paměťová hierarchie

Jirka Fink

<https://ktiml.mff.cuni.cz/~fink/>

Katedra teoretické informatiky a matematické logiky
Matematicko-fyzikální fakulta
Univerzita Karlova v Praze

Zimní semestr 2024/25

Licence: Creative Commons BY-NC-SA 4.0

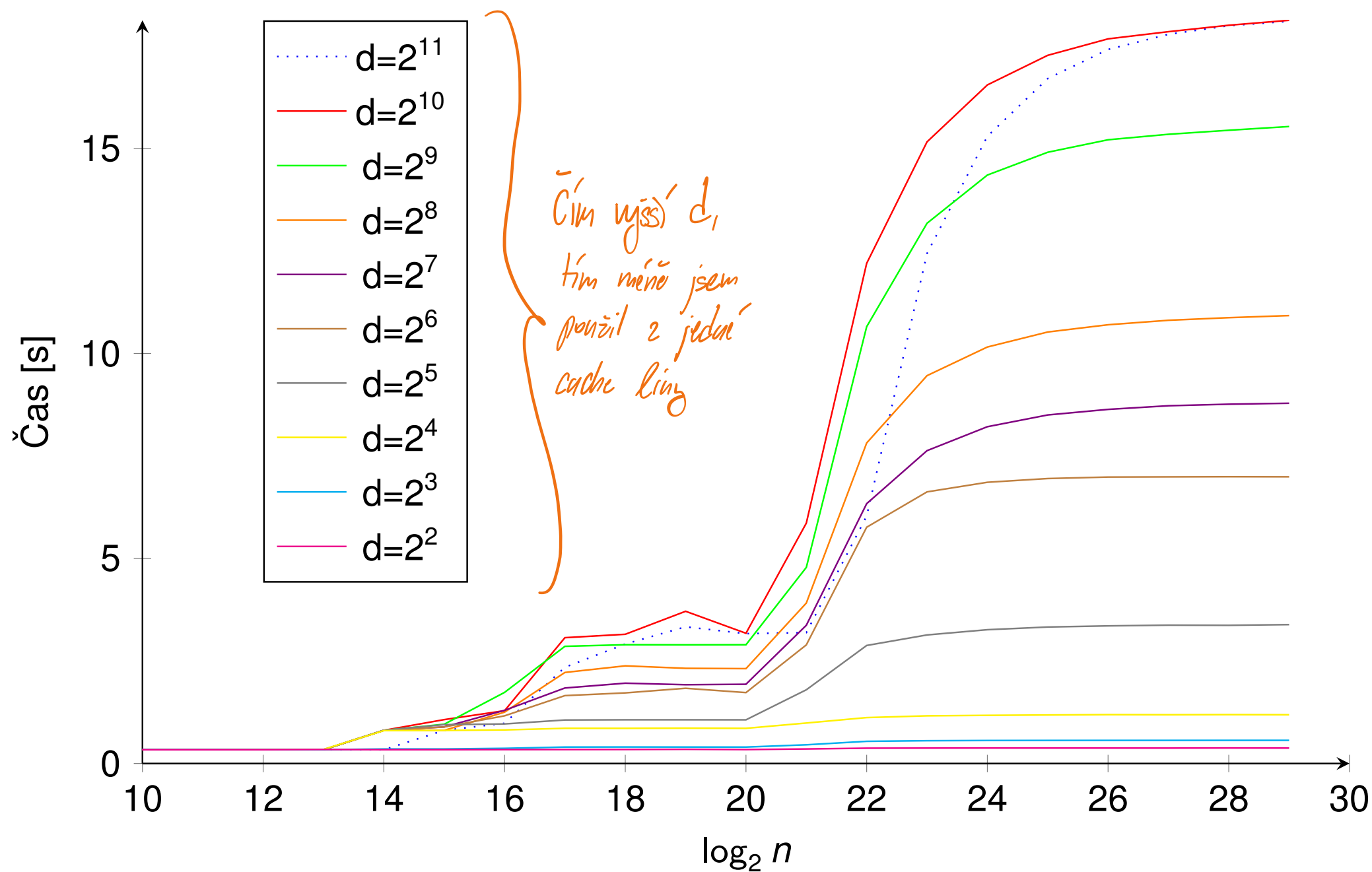
Příklad velikostí a rychlostí různých typů paměti

	velikost	rychlost
L1 cache	32 KB	223 GB/s
L2 cache	256 KB	96 GB/s
L3 cache	8 MB	62 GB/s
RAM	32 GB	23 GB/s
SDD	112 GB	448 MB/s
HDD	2 TB	112 MB/s

Triviální program

```
# Inicializace pole 32-bitových čísel velikosti  $n$ 
1 for ( $i=0; i+d < n; i+=d$ ) do
2   |  $A[i] = i+d$  # Vezmeme každou  $d$ -tou pozici a vytvoříme cyklus
3  $A[i]=0, i=0$ 
# Měříme dobu průběhu cyklu v závislosti na parametrech  $n$  a  $d$ 
# Počet operací je nezávislý na  $n$  a  $d$ 
4 for ( $j=0; j < 2^{28}; j++$ ) do
5   |  $i = A[i]$  # Dokola procházíme cyklus  $d$ -tých pozic
```

Paměťová hierarchie: Triviální program



Zjednodušený model paměti

- Uvažujme pouze na dvě úrovně paměti: pomalý disk a rychlá cache
- Paměť je rozdělená na bloky (stránky) velikosti B ①
- Velikost cache je M , takže cache má $P = \frac{M}{B}$ bloků
- Procesor může přistupovat pouze k datům uložených v cache
- Paměť je plně asociativní ② *→ data může ležet na libovolném místě*
- Data se mezi diskem a cache přesouvají po celých blocích a cílem je určit počet bloků načtených do cache

Cache-aware algoritmus

Algoritmus zná hodnoty M a B a podle nich nastavuje parametry (např. velikost vrcholu B-stromu při ukládání dat na disk).

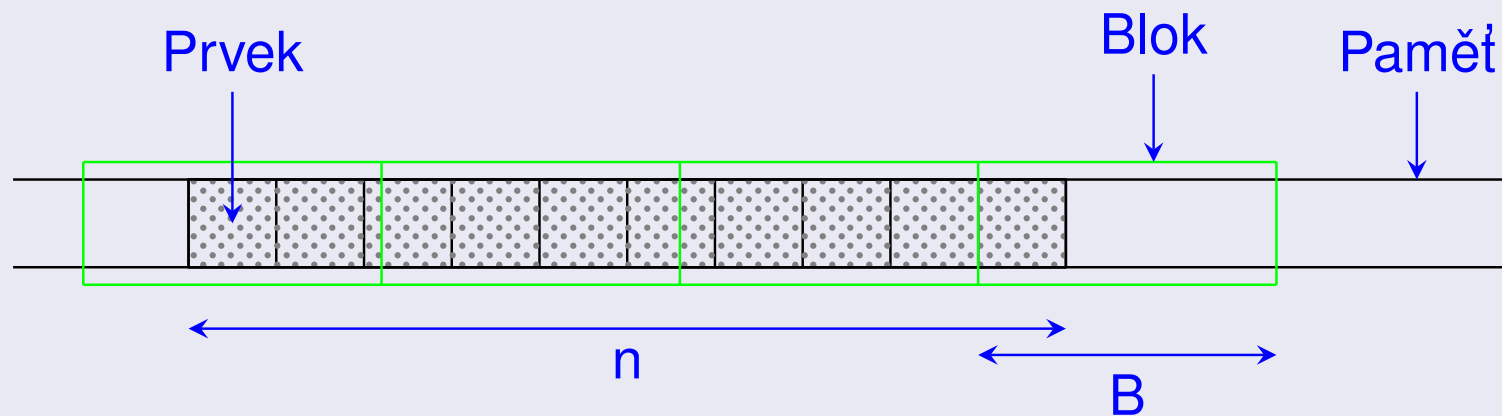
Cache-oblivious algoritmus *→ tohle je ideální, protože bude fungovat stejně (dobře) na všech zařízeních*

Algoritmus musí efektivně fungovat bez znalostí hodnot M a B . Důsledky:

- Není třeba nastavovat parametry programu, který je tak přenositelnější
- Algoritmus dobře funguje mezi libovolnými úrovněmi paměti (L1 – L2 – L3 – RAM)

- ① Pro zjednodušení předpokládáme, že jeden prvek zabírá jednotkový prostor, takže do jednoho bloku se vejde B prvků.
- ② Předpokládáme, že každý blok z disku může být uložený na libovolné pozici v cache. Tento předpoklad výrazně zjednodušuje analýzu, i když na reálných počítačích moc neplatí, viz https://en.wikipedia.org/wiki/CPU_cache#Associativity.

Přečtení souvislého pole (výpočet maxima, součtu a podobně)



- Minimální možný počet přenesených bloků je $\lceil n/B \rceil$
- Skutečný počet přenesených bloků je nejvýše $\lceil n/B \rceil + 1$
- Předpokládáme, že máme k dispozici $\mathcal{O}(1)$ registrů k uložení iterátoru a maxima

Obrácení pole

Počet přenesených bloků je stejný za předpokladu, že $P \geq 2$.

↳ kdybych měl jen jeden blok, budu mít zbytečný výsledek —> musím to mít kam „vrazi-uklidnit“

Binární vyvážený strom

- Jeden vrchol je uložen v nejvýše 2 blocích ①
- Výška stromu je $\mathcal{O}(\log n)$
- Na cestě z kořene do listu načteme $\mathcal{O}(\log n)$ bloků

(a,b)-strom

- Předpokládejme, že můžeme zvolit $b = \Theta(B)$
- Jeden vrchol je uložen v nejvýše 2 blocích a má $\Theta(B)$ synů
- Výška stromu je $\Theta(\log_B n)$
- Na cestě z kořene do listu načteme $\Theta(1 + \log_B n)$ bloků
- Toto je cache-aware přístup

Reprezentace binárního stromu pomocí van Emde Boas

- Cache-oblivious reprezentace
- Na cestě z kořene do listu načteme $\Theta(1 + \log_B n)$ bloků
- Podrobnosti na Datových strukturách II

- 1 Předpokládáme, že prvky jsou dost malé, aby se vrchol vešel do jednoho bloku a druhý blok máme pro případ, že nemůžeme alokovat paměť tak, aby se paměť pro vrcholy byla zarovnána se začátky bloků.

Binární halda v poli: Průchod od listu ke kořeni



- Cesta má $\Theta(\log n)$ vrcholů
- Posledních $\Theta(\log B)$ vrcholů leží v nejvýše dvou blocích
- Ostatní vrcholy jsou uloženy v po dvou různých blocích
- $\Theta(1 + \log n - \log B) = \Theta(1 + \log \frac{n}{B})$ přenesených bloků

B-regulární halda v poli: Průchod od listu ke kořeni

- Opět předpokládáme, že do jednoho bloku se vejde B prvků
- Cesta má $\Theta(1 + \log_B n)$ vrcholů
- Na cestě z kořene do listu načteme $\Theta(1 + \log_B n)$ bloků

*Na začátku stačí hodně (o půlku vždy zmenšujíc podstranu),
takže hlavně na začátku dochází k výpadkům při hledání prvků*

Binární vyhledávání

- Uvažujeme neúspěšné vyhledávání, protože úspěšné může skončit dříve
- Porovnáváme $\Theta(\log n)$ prvků s hledaným prvkem
- Posledních $\Theta(\log B)$ prvků je uloženo v nejvýše dvou blocích
- Ostatní prvky jsou uloženy v po dvou různých blocích
- $\Theta(1 + \log \frac{n}{B})$ přenesených bloků

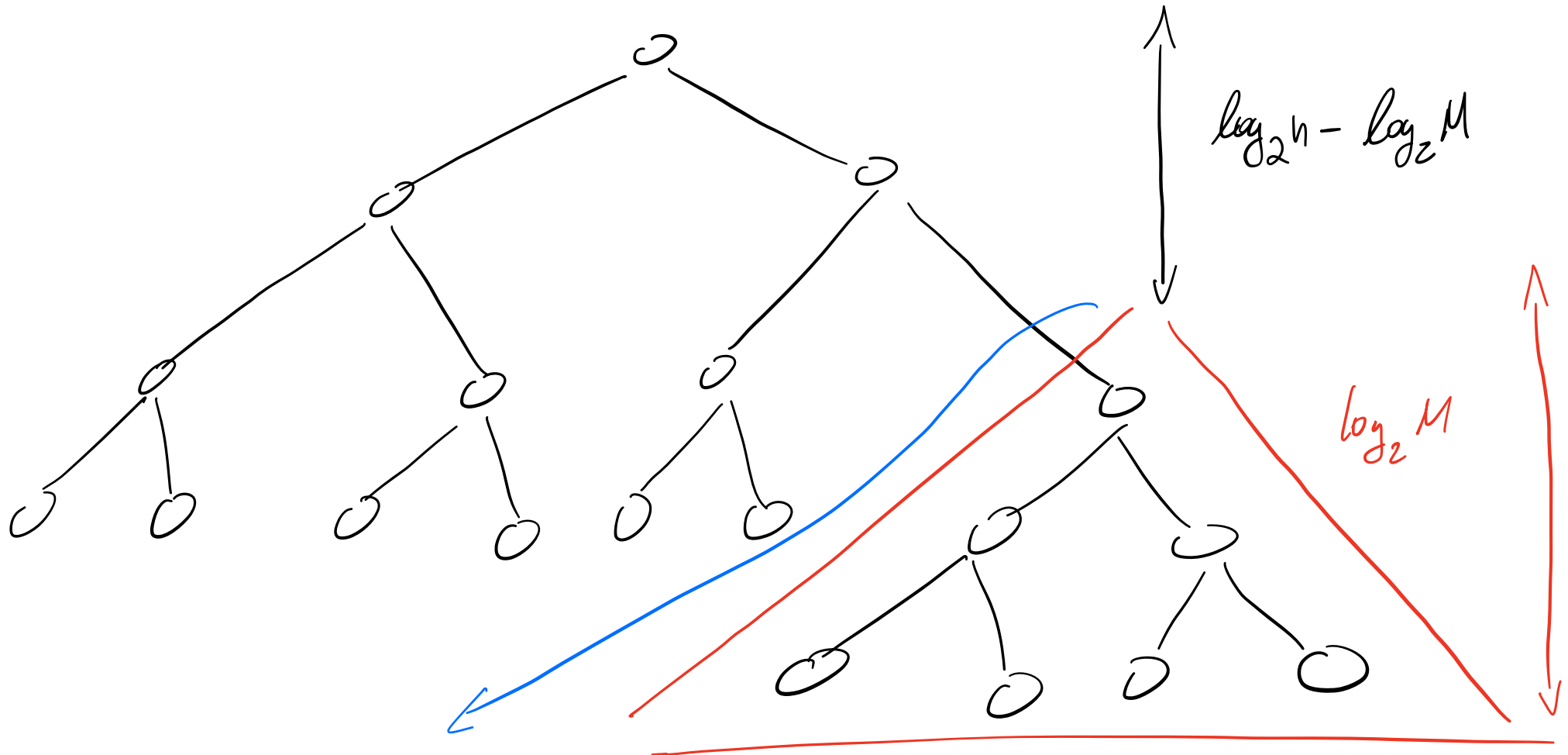
Mergesort

- 1 Slití polí velikosti n_1 a n_2 potřebuje $\frac{2}{B}(n_1 + n_2) + \mathcal{O}(1)$ přenosů
- 2 Uvažujme (binární) strom rekurzivního volání
- 3 Na jedné hladině stromu potřebujeme $\mathcal{O}(n/B + 1)$ přenosů
- 4 Počet hladin stromu je $\mathcal{O}(\log n)$
- 5 Celkový počet přenosů je $\mathcal{O}((n/B + 1) \log n)$

k-cestný mergesort

- 1 Slití polí velikostí $n_1, \dots, n_k$ potřebuje $\frac{2}{B}(n_1 + \dots + n_k) + \mathcal{O}(1)$ přenosů
- 2 Musíme volit $k < P$
protože procesor pracuje jen s cache, takže musím mít přenos do a z cache
- 3 Uvažujme k -ární strom rekurzivního volání
- 4 Na jedné hladině stromu potřebujeme $\mathcal{O}(n/B + 1)$ přenosů
- 5 Počet hladin stromu je $\mathcal{O}(\log_k n)$
- 6 Celkový počet přenosů je $\mathcal{O}((n/B + 1) \log_k n)$ *to P má ale limitu...*
- 7 Volbou $k = \Theta(P)$ dostáváme $\mathcal{O}((n/B + 1) \log_P(n))$ přenosů ①

- 1 Tento algoritmus je cache-aware a tento počet přenosů je teoreticky optimální i v cache-oblivious modelu. Funnelsort je cache-oblivious algoritmus mající tento počet přenosů a časovou složitost $\mathcal{O}(n \log n)$



dívám se do nejvyšší možné úrovně, kde se mi celý podstrom vejde do cache, další výpadky už nenastanou

ve skutečnosti se mi takhle může všechno vejít do cache a čím méně počet přístupů mít širší tyto bloky

→ 2x předpoklad DFS mergesortu

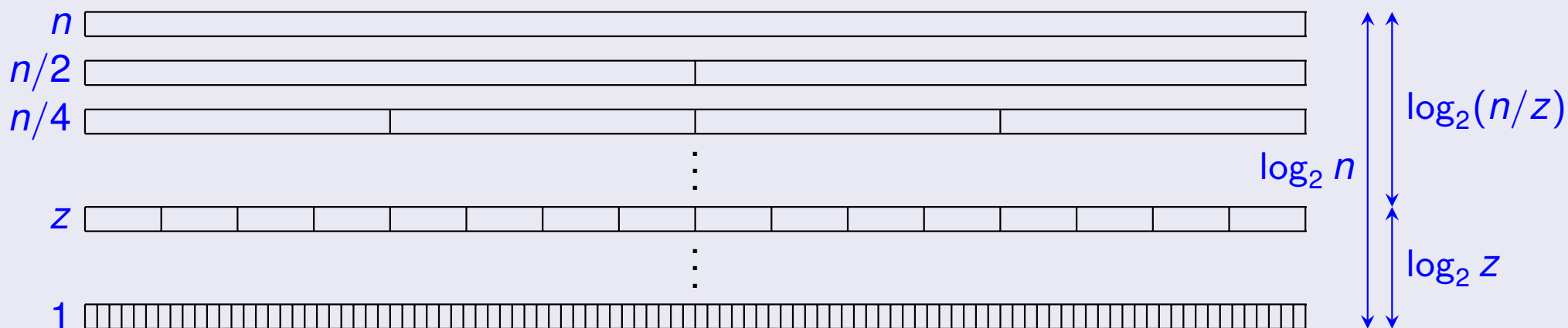
Případ $n \leq M/2$

Celé pole se vejde do cache, takže přenášíme $2n/B + \mathcal{O}(1)$ bloků. ①

Schéma

Délka spojovaných polí

Výška stromu rekurze



Případ $n > M/2$

① Nechť z je maximální velikost pole, která může být seříděná v cache ②

② Platí $z \leq \frac{M}{2} < 2z$

③ Slití jedné úrovně vyžaduje $2 \frac{n}{B} + 2 \frac{n}{z} + \mathcal{O}(1) = \mathcal{O}(\frac{n}{B})$ přenosů. ③

④ Počet přenesených bloků je $\mathcal{O}(\frac{n}{B}) (1 + \log_2 \frac{n}{z}) = \mathcal{O}(\frac{n}{B} \log \frac{n}{M})$. ④

$\log n - \log M$
rozdělení do bloků

- 1 Polovina cache je použita na vstupní pole a druhá polovina na slité pole.
- 2 Pro jednoduchost předpokládáme, že velikosti polí v jedné úrovni rekurze jsou stejné. z odpovídá velikosti pole v úrovni rekurze takové, že dvě pole velikost $z/2$ mohou být slity v jedno pole velikost z .
- 3 Slití všech polí v jedné úrovni do polovičního počtu polí dvojnásobné délky vyžaduje přečtení všech prvků. Navíc je třeba uvažovat nezarovnání polí a bloků, takže hraniční bloky mohou patřit do dvou polí.
- 4 Funnelsort přenese $\mathcal{O}\left(\frac{n}{B} \log_P \frac{n}{B}\right)$ bloků.

Transpozice matic: Triviální přístup

Strategie pro výměnu stránek v cache

OPT: Optimální off-line algoritmus předpokládající znalost všech přístupů do paměti

FIFO: Z cache smažeme stránku, která je ze všech stránek v cachi nejdelší dobu

LRU: Z cache smažeme stránku, která je ze všech stránek v cachi nejdéle nepoužitá

Triviální algoritmus pro transpozici matice A velikost $k \times k$

```
1 for  $i \leftarrow 1$  to  $k$  do
2   for  $j \leftarrow i + 1$  to  $k$  do
3     Swap( $A_{ij}$ ,  $A_{ji}$ )
```

11	12	13	14
21	22	23	24
31	32	33	34
41	42	43	44

$i+1$, protože diagonála
necháme

Předpoklady

Uvažujeme pouze případ

- $B < k$: Do jednoho bloku cache se nevejde celá řádka matice
- $P < k$: Do cache se nevejde celý sloupec matice

Příklad: Representace matice 5×5 v paměti

11	12	13	14	15	21	22	23	24	25	31	32	33	34	35	41	42	43	44	45	51	52	53	54	55
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

LRU a FIFO strategie

Při čtení matice po sloupcích si cache pamatuje posledních P řádků, takže při čtení prvku $A_{3,2}$ již prvek $A_{3,1}$ není v cache. Počet přenesených bloků je $\Omega(k^2)$.

OPT strategie

- 1 Transpozice prvního řádku/sloupce vyžaduje alespoň $k - 1$ přenosů.
- 2 Nejvýše P prvků z druhého sloupce zůstane v cache.
- 3 Proto transpozice druhého řádku/sloupce vyžaduje alespoň $k - P - 2$ přenosů.
- 4 Transpozice i -tého řádku/sloupce vyžaduje alespoň $\max\{0, k - P - i\}$ přenosů.
- 5 Celkový počet přenosu je alespoň $\sum_{i=1}^{k-P} k - P - i = \Omega((k - P)^2)$.

→ tabule by v czechy mohlo být, protože 12 bylo násobeno

		X	X			X	X		
X									
X	X	X	X						
X	X	X	X						
X	X	X	X						
X	X	X	X						
X	X	X	X						
X	X	X	X						

→ tedy bude lineárně 13 prvků výpočtu

→ tabule už tam proto nebude

→ všude jinde výpočtu za lineární $A_{ij} = A_{ji}$

→ když mám LRU, tak si vždy vyhledám to, co budu za chvíli potřebovat

tedy bude bez výpočtu

tabule ale v czechy násobeno nebylo

Tabulice podle mě to stojí $O(n^2)$ operací a je to to nejhorší, co může nastat

Cache-aware algoritmus pro transpozici matice A velikost $k \times k$

```
# Rozdělíme danou matici na submatice velikosti  $z \times z$ 
1 for ( $i = 0; i < k; i += z$ ) do
2   for ( $j = i; j < k; j += z$ ) do
3     # Transponujeme submatici začínající na pozici  $(i, j)$ 
4     for ( $ii = i; ii < \min(k, i + z); ii ++$ ) do
5       for ( $jj = \max(j, ii + 1); jj < \min(k, j + z); jj ++$ ) do
        Swap( $A_{ii,jj}, A_{jj,ii}$ )
```

Analýza

- Zvolíme $z = B$ a předpokládáme, že $B \leq P$ $\mathcal{O}(B)$
 - K transpozici jedné submatice potřebujeme $\mathcal{O}(z)$ přenosů
 - Počet submatic je $\mathcal{O}((k/z)^2)$
 - K transpozici potřebujeme $\mathcal{O}(k^2/B)$ přenosů, což je optimální
 - Při správně zvolené hodnotě z bývá tento postup nejrychlejší
- maximálně udělat tolik přenosů, kolik má náš řádek*

Rozdělím mntice na podmntice, aby se mi všly do cache,
pak ztmsponujm vždy tyto dva konkrétní bloky.

→ pak si j do cache načtn jen jednou bez budoucích výpočtů

$2 = B \rightarrow$ řádek se vejde do cache-line

Tail-cache = všechny řádky se mi vejdn do cache

Mň než $\mathcal{O}(b^2/B)$ to být nemůže, protože tu mntici celou musím vidět
a v nějakých blocích to vůbec být musí...

Idea

Rekurzivně rozdělíme na submatice *zrcadlo vůči diagonále*

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad A^T = \begin{pmatrix} A_{11}^T & A_{21}^T \\ A_{12}^T & A_{22}^T \end{pmatrix}$$

- Matice A_{11} a A_{22} se transponují podle stejného schématu
- Matice A_{12} a A_{21} se prohazují
- Transpozice a prohození matic A_{12} a A_{21} se provádí najednou

Transpozice matic: Rekurzivní transpozice

```
1 Procedure transpose_on_diagonal (A)
2   if Matice A je malá then
3     Transponujeme matici A triviálním postupem
4   else
5      $A_{11}, A_{12}, A_{21}, A_{22} \leftarrow$  souřadnice submatic
6     transpose_on_diagonal ( $A_{11}$ )
7     transpose_on_diagonal ( $A_{22}$ )
8     transpose_and_swap ( $A_{12}, A_{21}$ )
```

```
9 Procedure transpose_and_swap (A, B)
10  if Matice A a B jsou malé then
11    Prohodíme a transponujeme matice A a B triviálním postupem
12  else
```

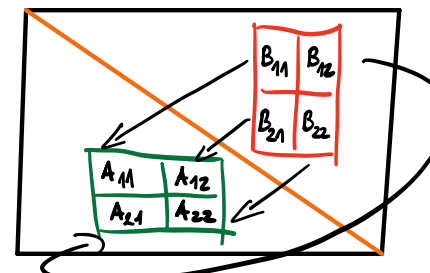
```
13     $A_{11}, A_{12}, A_{21}, A_{22}, B_{11}, B_{12}, B_{21}, B_{22} \leftarrow$  souřadnice submatic
14    transpose_and_swap ( $A_{11}, B_{11}$ )
15    transpose_and_swap ( $A_{12}, B_{21}$ )
16    transpose_and_swap ( $A_{21}, B_{12}$ )
17    transpose_and_swap ( $A_{22}, B_{22}$ )
```

*až tedy, co
se vejde do cache*

→ tedy žádný přístup do paměti není ?

Musím zachovat symetrii vůči diagonále

*→ je důležité, že takhle transponuje
a prohazuje (díky rekurzi) zároveň*

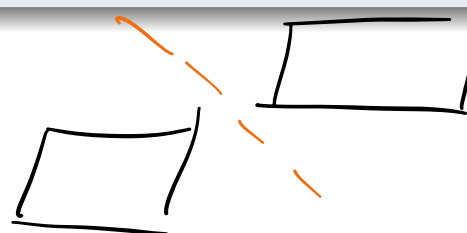


Zachování symetrie

- Všimněme si, že matice A a B musí mít pozice symetrické podle hlavní diagonály původní matice, a proto ve skutečnosti funkci `transpose_and_swap()` stačí předávat pozice matici A .
- Ve funkci `transpose_on_diagonal` musí být matice A čtvercová a ležet na hlavní diagonále, a proto stačí předávat x-ovou souřadnici a řád matice.
- Funkci `transpose_on_diagonal` stačí předat dva argumenty i a m , aby má transponovat matici velikosti $m \times m$ začínající na pozici (i, i)
- Funkci `transpose_and_swap` stačí předat čtyři argumenty i, j, m, n , aby má transponovat a prohodit matici velikosti $m \times n$ začínající na pozici (i, j) s maticí velikosti $n \times m$ začínající na pozici (j, i)

Analýza počtu přenesených bloků

- 1 Předpoklad „Tall cache“: $M \geq 4B^2$, tj. počet bloků je alespoň $4B$ ①
- 2 Necht z je maximální velikost submatice, ve které se jeden řádek vejde do jednoho bloku ②
- 3 Platí: $z \leq B \leq 2z$ *→ všechny matice zmenšujeme o $1/2$...* *slide 13*
- 4 Jedna submatice $z \times z$ je uložena v nejvýše $2z \leq 2B$ blocích
- 5 Dvě submatice $z \times z$ se vejdou do cache ③
- 6 Transpozice matice typu $z \times z$ vyžaduje nejvýše $4z$ přenosů *→ každý řádek může vytěhnout 2 bloky, takže každý potřebuje 2 přenosy, pak transponují navzájem 2 matice*
 $B \leq 2z$
- 7 Máme $(k/z)^2$ submatic velikosti z
- 8 Celkový počet přenesených bloků je nejvýše $\frac{k^2}{z^2} \cdot 4z \leq \frac{8k^2}{B} = O\left(\frac{k^2}{B}\right)$
- 9 Tento postup je optimální až na multiplikativní faktor ④



- 1 Stačilo by předpokládat, že počet bloků je alespoň $\Omega(B)$. Máme-li alespoň $4B$ bloků, pak je postup algebraicky jednodušší.
- 2 Pokud začátek řádky není na začátku bloku, tak je jeden řádek submatice uložen ve dvou blocích.
- 3 Funkce `transpose_and_swap` pracujeme se dvěma submaticemi.
- 4 Celá matice je uložena v alespoň $\frac{k^2}{B}$ blocích paměti.