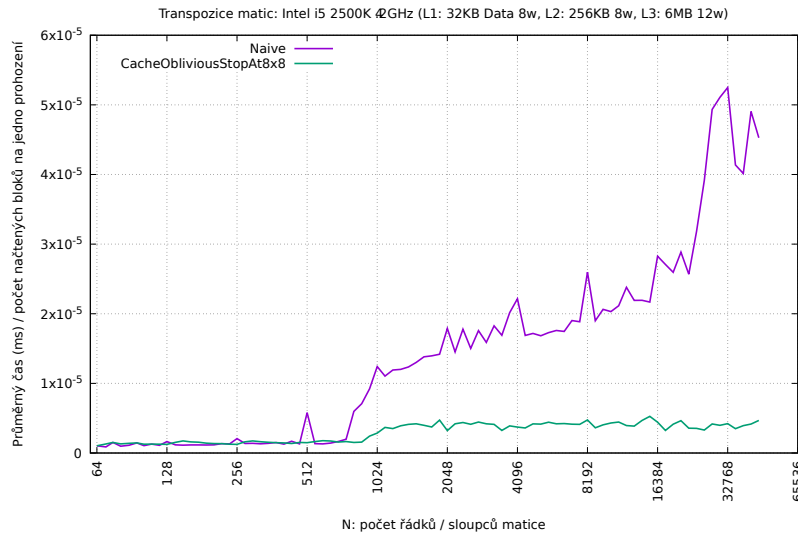


0.1 Reálné CPU



Na prvním grafu vidíme porovnání průměrného času na prohození dvou prvků pro naivní algoritmus a rekurzivní cache oblivious algoritmus, který se zastaví na podmatici velikosti 8x8 a dál pokračuje naivně.

0.1.1 Cache oblivious

Pro cache oblivious algoritmus jsou zřejmé dva segmenty, s rozhraním okolo cirká 800^2 prvků. Skok mezi těmito dvěma segmenty není nikterak výrazný, z $1.5 \cdot 10^{-6}$ na $4.0 \cdot 10^{-6}$.

Tento skok přibližně odpovídá tomu, kdy se celá matice (640000 prvků, 3MB) ještě vleze do L3 cache. Skok tedy začíná trochu dříve než by čistě matematicky měl. I pro $n == 1000$ (4MB) by se do L3 matice měla vejít celá, nicméně kvůli asociativitě cache a komplikovanosti dnešního HW není úplně překvapivé, že ke zhoršování průměrného času dochází před úplným naplněním.

Byť to z grafu není úplně vidět, tak čistá data naznačují, že podobný skok by mohl být i u hranice velikosti L2 a L3 cache. Konkrétně pro $n \approx 256$, kdy se celá matice ještě vejde do L2, se průměrný čas přístupu zvedne z $1.2 \cdot 10^{-6}$ na $1.4 \cdot 10^{-6}$. Kvůli šumu a skokům k vyšším hodnotám pro určitá n (téměř jistě kvůli asociativitě) se ovšem nedá říct nic s jistotou.

Na rozhraní n pro matice, které se těsně vejdu do L1 cache žádný pozorovatelný rozdíl není.

0.1.2 Naive

U naivního algoritmu jsou jasně pozorovatelné segmenty 3. Od začátku do 1000^2 prvků, kde se průměrný čas pohybuje okolo $1.5 \cdot 10^{-6}$, tedy stejně jako u cache oblivious algoritmu. Následně do 20000^2 prvků, kdy je čas okolo $1 - 2 \cdot 10^{-5}$, a nakonec poslední segment, ve kterém čas roste až k $5 \cdot 10^{-5}$ milisekund na prohození.

První skok odpovídá již dříve popsánému přechodu z toho, kdy se celá matice ještě vleze do L3 cache a kdy už ne. Dokud se vleze, tak je naivní algoritmus, s výjimkou $n == 512$, naprosto srovnatelný s cache aware variantou. Pro $n == 512$ se téměř určitě projevuje asociativita cache. Ta má která má, vzhledem k access patternu, daleko větší dopad v případě naivního přístupu (kde přistupujeme postupě na všechny řádky pod sebou, které jsou od sebe přesně o $4 \cdot 512B$) než v případě cache oblivious.

To že je naivní přístup až do n , kdy se matice nevejde do L3 cache, srovnatelný s cache oblivious algoritmem jen dále podporuje tezi, že mezi tím, jestli se matice vejde do L1, L2 nebo jen L3 není žádný významně pozorovatelný rozdíl, který by nebyl z větší části vynahrazen cache pre-fetchem a zamaskován ostatními vlivy. Pokud by totiž přesouvání z L3 cache do L1 a L2 mělo výrazný overhead, tak by se tento overhead nutně více projevil v naivním algoritmu (cache oblivious přístup tyto přesuny minimalizuje).

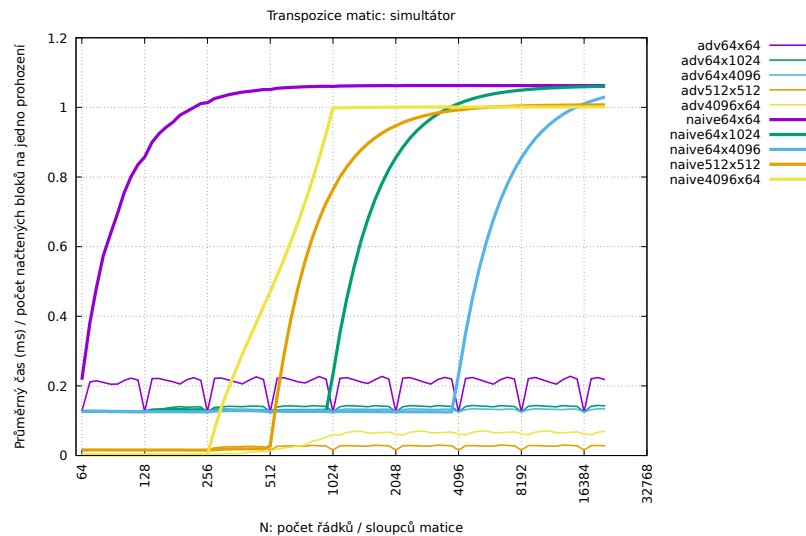
Od n , kdy se matice už nevleze do L3 cache roste průměrná doba swapu postupně až do $n \approx 22000$ (matice velikosti 2GB), kdy začne raketově růst z $2 \cdot 10^{-5}$ k $5 \cdot 10^{-5}$ milisekundám. Tam se následně, byť v velkém rozptylem, drží až do maximální testované velikostí $n == 44591$ (7.9 GB).

Tento růst a obecně velký rozptyl v druhém i třetím segmentu si vysvětlují tím, jak je paměť RAM strukturovaná. Za zmínku stojí ještě vyšší průměrný čas vždy když je n násobkem mocin 2. Z části to bude způsobené tím, že u poloviny přístupů (ty co jdou v rámci řádku) se bude více projevovat cache aliasing (viz výše) způsobený asociativností cache. Z části to může být způsobené také strukturou paměti RAM, nicméně na posouzení toho co má konkrétně jaký vliv nemám dostatečné znalosti.

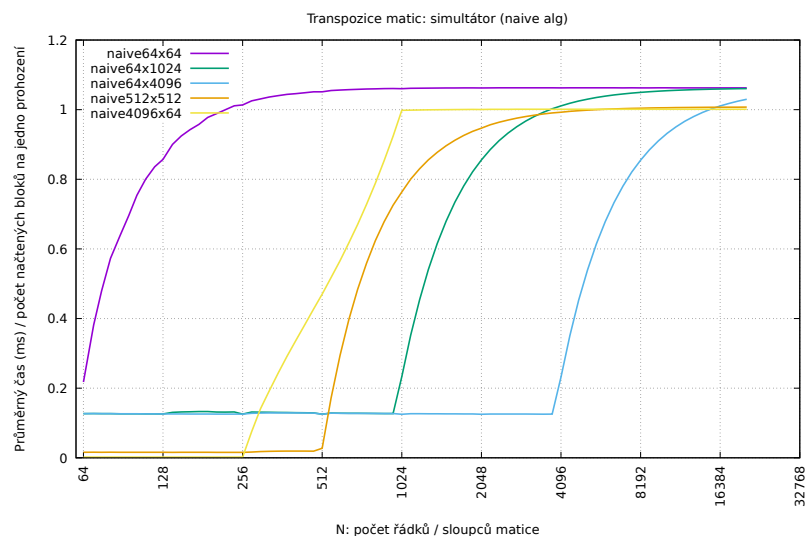
0.2 Simulátor

Na grafu ze simulátoru je hned na první pohled pozorovatelný jeden fakt. Zatímco cache oblivious algoritmus drží průměrný počet načtených bloků vůči velikosti matice (nebo jejího řádku) konstantní, tak pro naivní začne vždy od nějakého n průměrný počet načtených bloků růst k číslu okolo 1. To odpovídá tomu, že jeden ze dvou přístupů swapu (ten co jde ve sloupci) způsobí vždy načtení nového bloku.

Naopak pro cache oblivious přístup je zjevná periodicitu. Pro n mocniny dvojky je průměrný počet načtených bloků vždy nižší než pro ostatní n . To je způsobeno tím, že pro n mocniny dvojky jsou všechny rekurzivní podmatice vždy zarovnané na začátku cache řádků tím nejlepším způsobem, jak jen mohou být.



0.2.1 Naivní přístup



Jak bylo řečeno již výše, tak v naivním přístupu pro každou konfiguraci cachí platí následující. Průměrný počet načtených bloků je konstantní až do velikosti, kdy má matice více řádků než kolik má cache bloků, v ten moment začne růst k 1. Zlomová velikost vždy odpovídá tomu, kdy začne mít matice více řádků než kolik je cache bloků.

Jakmile má totiž matice více řádků než je cache bloků, tak naivní přístup z cache vyhodí v rámci přístupů po řádcích kvůli $radku > cacheBloku$ první řádek. A v rámci dalšího průchodu tak první řádek musí znovu načíst. To ale vyhodí druhý řádek, a tak dále. Přístup po řádcích tedy vždy vyústí v načtení jednoho bloku.

Tento efekt se nezačne projevovat úplně ostře (zlom není ostrý z malých hodnot rovnou k 1) kvůli tomu, že počet efektivních řádků z pohledu algoritmu se lineárně mění z 1 do n . U přístupů v rámci řádků totiž jedeme v horním trojúhelníku (nad diagonálou) a u přístupů po řádcích v dolním trojúhelníku (pod diagonálou). Díky tomu po zlomu roste průměrný počet načtených bloků postupně (v relaci k tomu jakou část běhu algoritmu je počet řádků větší než počet bloků cache). Čím je počet bloků větší tím postupnější je tento přechod.

Pro přístupy v rámci řádku (v naivním algoritmu je vždy jedne po řádcích a jeden v rámci řádku) se pak s n nic nemění. Konfigurace cache jen ovlivňuje, jaký je průměrný počet načtených bloků na jeden přístup. Různost těchto čísel pro různé konfigurace způsobuje, proč po zlomovém bodě nekonvergují všechny průměrné počty přístupů k jednomu číslu, ale proč jdou konfigurace s delšími cache bloky k 1 a ty s kratšími jdou k hodnotě > 1 . Konkrétně pro cache bloky o 64B k $1 + 1/(64/4) = 1.062$

Co se týče průběhu před zlomem, tak tam se výše popsany průměrný počet bloků na přístup v rámci řádku projevuje také, což vysvětluje různé výšky ze začátku. Například pro konfiguraci 64x1024 a 64x4096 je to $2 * 1.062 = 0.124$. Načtení nového bloku je totiž třeba každých 64/4 bloků pro přístupy v rámci řádku a to samé v rámci sloupců (každých 64/4 sloupců je potřeba načíst pro každý prvek ve sloupci, pro zbylé sloupce se jede z již načtených bloků). Obdobně pro ostatní konfigurace.

0.2.2 Cache oblivious přístup

