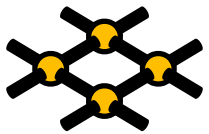
Vláknové programování
část IX

Lukáš Hejmánek, Petr Holub
{xhejtman,hopet}@ics.muni.cz



Laboratoř pokročilých síťových technologií

PV192
2014–04–29

Přehled přednášky

Hierarchie pamětí

- Oproti původnímu von Neumannovu konceptu existuje více druhů pamětí
- Různé druhy paměti se liší rychlostí, kapacitou, cenou
- Nelze mít veškerou paměť pouze rychlou a s dostatečnou kapacitou
- Čím má paměť vetší kapacitu, tím je obvykle pomalejší
  - Ať už z ekonomických důvodů
  - Nebo z fyzikálních omezení
- Pevný disk $\Rightarrow$ Flash paměť $\Rightarrow$ Paměť DRAM $\Rightarrow$ Cache $\Rightarrow$ Registry

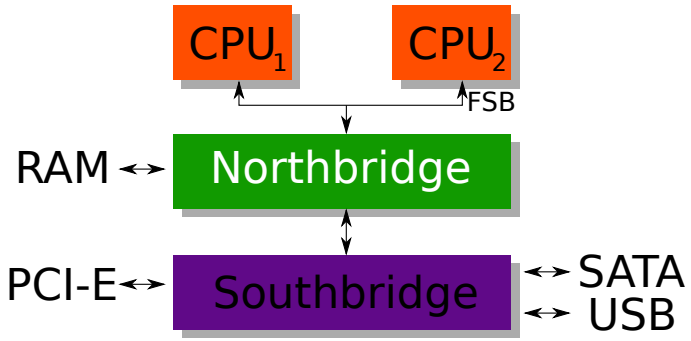
Hierarchie pamětí

Odkud	Cyklů
Registr	≤ 1
L1d	~ 3
L2	~ 14
DRAM	~ 240

Údaje pro Pentium M

Architektura paměti

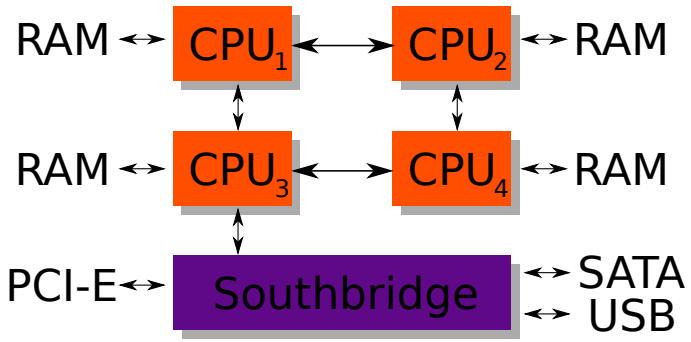
- Tradiční schéma pamětí RAM a CPU



- Problémy takového přístupu
  - Všechny procesory sdílí jednu sběrnici (quadpumped 800–1333MHz)
  - Komunikace s pamětí všech procesorů probíhá přes jeden Northbridge
  - Paměť má jen jeden přístupový port
  - Komunikace mezi procesory a zařízeními probíhá přes jeden Northbridge

Architektura paměti

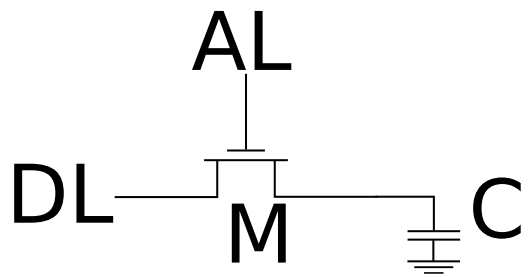
- NUMA schéma pamětí RAM a CPU



- Výhody takového přístupu
  - Komunikace s pamětí nepřetěžuje sběrnici s Northbridge
  - Více připojitelné paměti
- Nevýhody
  - Neuniformní přístup do paměti
  - Vysoký NUMA faktor je problém
  - Ne každý procesor může přímo komunikovat se zařízením

Principy DRAM

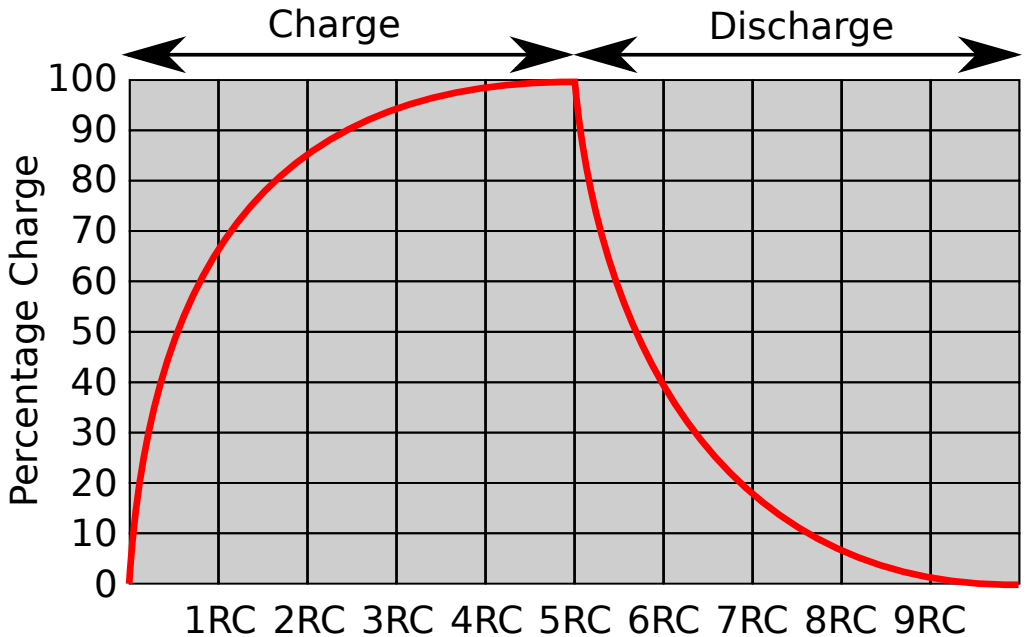
- Buňka paměti DRAM



- Kondenzátor C svým nabitím uchovává bitovou informaci 1 nebo 0
- Čtení/zápis bitu je řízen AL (adresní linka)
- Čtení/zápis bitu je provedeno na DL (datová linka)

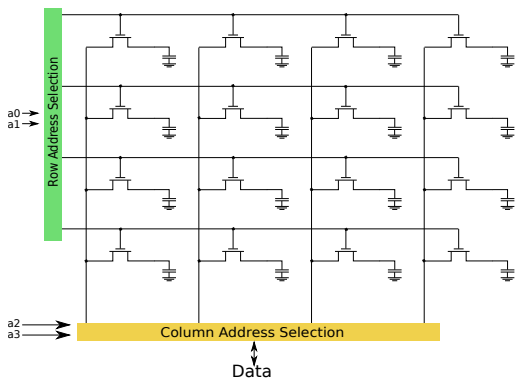
Principy DRAM

- Čtení a zápis do/z kondenzátoru není instantní



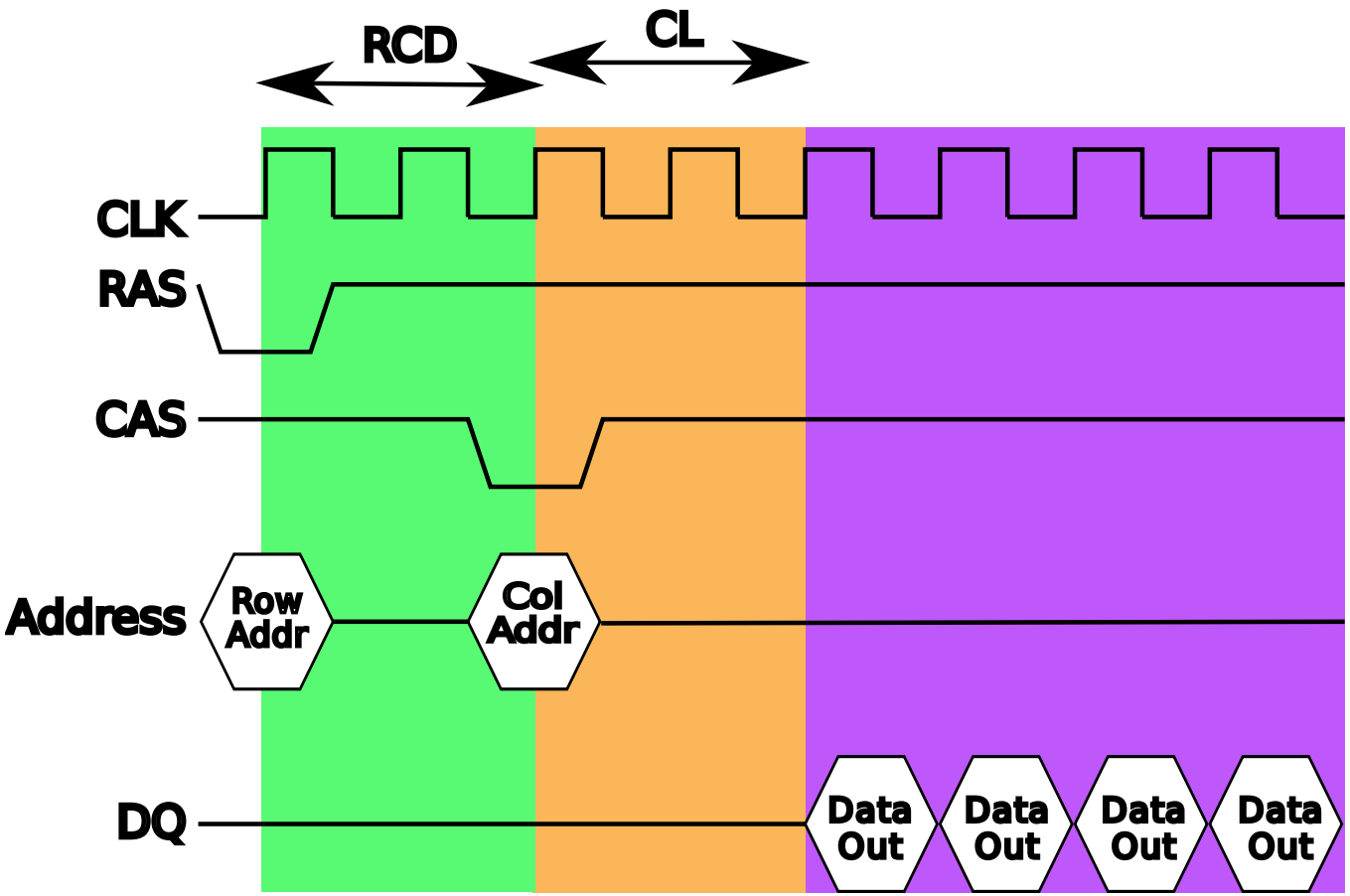
Principy DRAM

- AL a DL samostatně ke každému bitu by vyžadovalo extrémní množství propojů (Kolik je 32 Gb?)
- Buňky jsou v matici, adresujeme sloupce, řádky

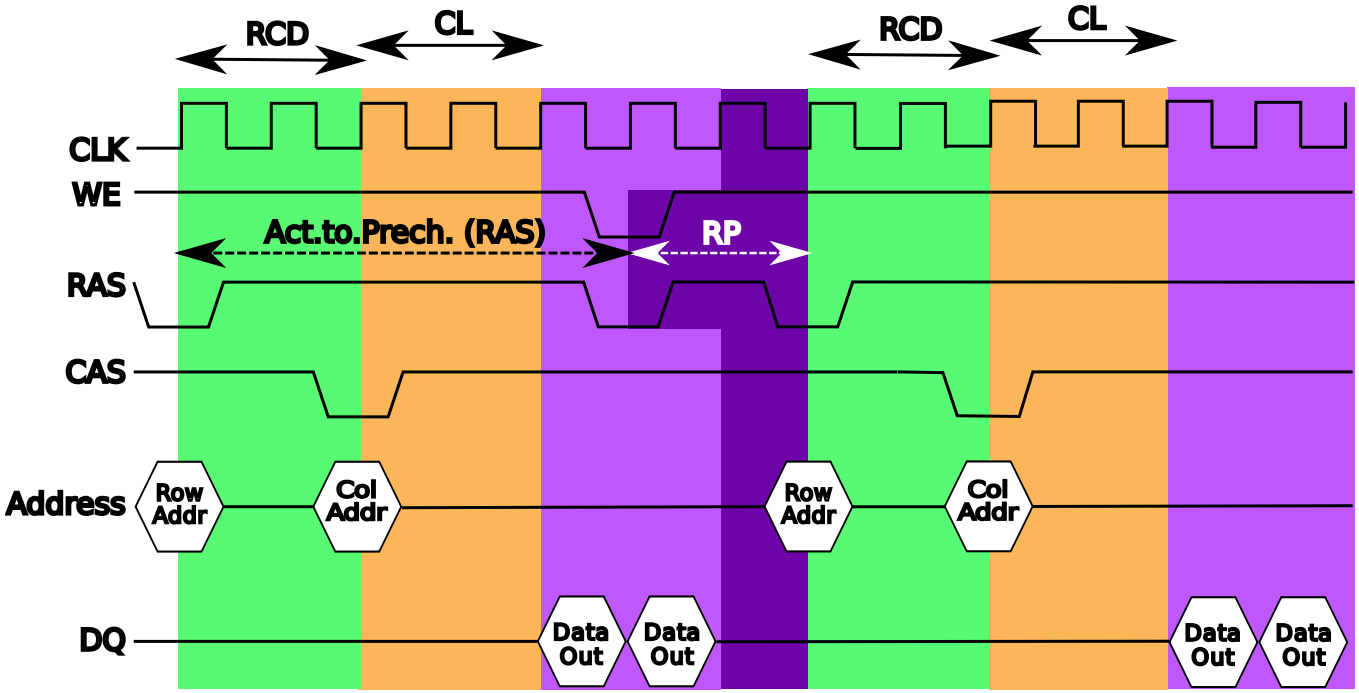


- Pinů pro zvlášť pro sloupce a zvlášť pro řádky je pořád příliš
- Probíhá adresace řádků a následně (v dalším cyklu) adresace sloupců

Čtení z paměti



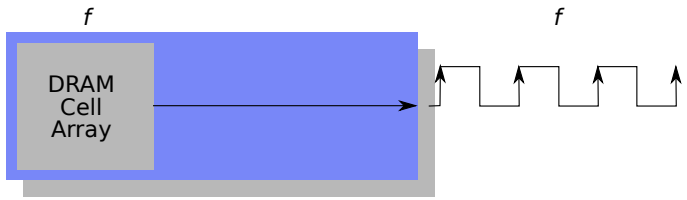
Čtení z paměti



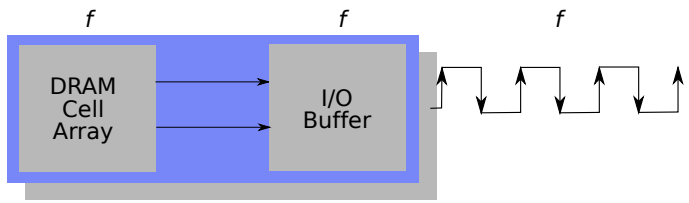
- CAS Latence (CL)
- RAS to CAS delay (RCD)
- RAS Precharge (RP)
- Active to Precharge delay (RAS)
- Command rate
- 2-3-2-8-T1

SDR, DDR, DDR2, DDR3

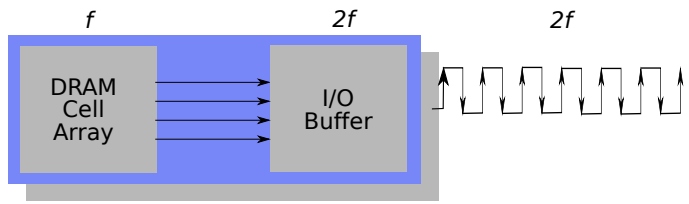
- Princip SDR paměti



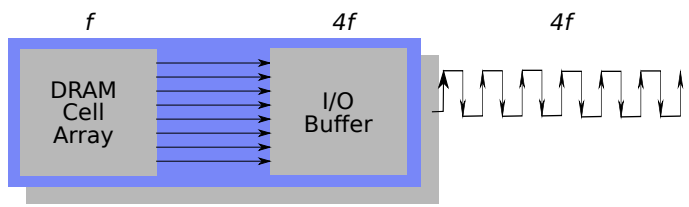
- Princip DDR paměti



- Princip DDR2 paměti



- Princip DDR3 paměti

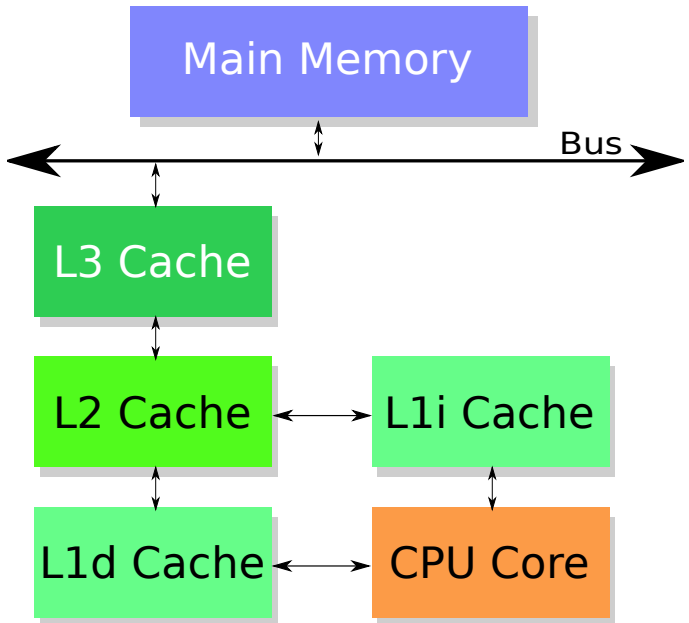


Problémy DDR pamětí

- Zvyšující se frekvence klade nároky na paralelní sběrnice
- DDR2 modul má 240 pinů, každý by měl být zhruba stejné délky
- Nejvýše dva DDR2 moduly lze osadit na jeden kanál
- DDR3 na rychlosti 1600MB/s podporovala pouze 1 modul na kanále
- Možná řešení
  - Paměťový řadič přímo v CPU (AMD Opteron, Intel Nehalem)
  - Výměna DDR3 za FB-DRAM – sériové moduly, 69 pinů

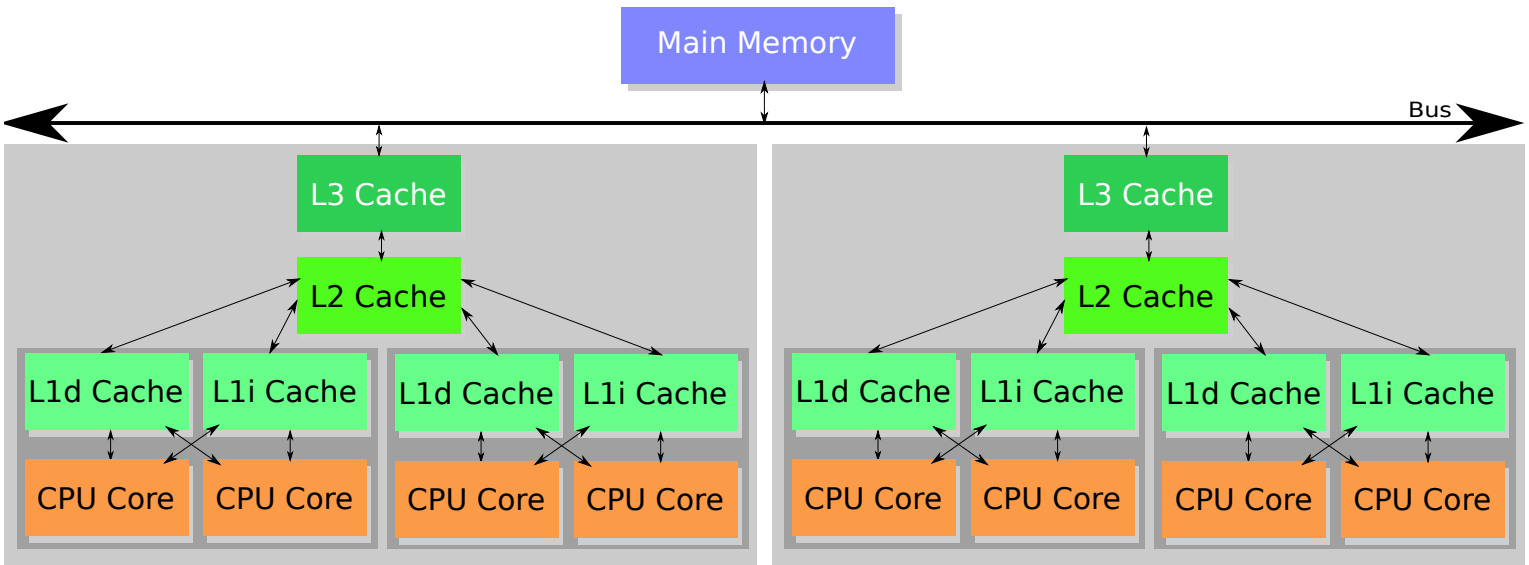
Cache paměť

- Zavádí hierarchii podobně jako paměti obecně
- Zde opět platí, že nelze mít velkou a zároveň rychlou paměť



Cache paměť

- Ve světě multijaderných, hyperthreadovaných CPU jsou cache paměti různým způsobem sdíleny



Organizace cache

- Cache je organizována po řádcích (cacheline)
- Velikost řádku cache je obvykle 64bytů (někdy 32 nebo 128)
- Při přístupu do paměti je do cache načten zpravidla celý řádek
- Hierarchii cache rozlišujeme exkluzivní a inkluzivní
- Exkluzivní
  - Oblast paměti pokrývá vždy pouze jedna úroveň cache (např. L2)
  - Případ AMD, VIA
- Inkluzivní
  - Vyšší úroveň cache pokrývá celou nižší úroveň cache
  - Případ Intelu

Organizace cache

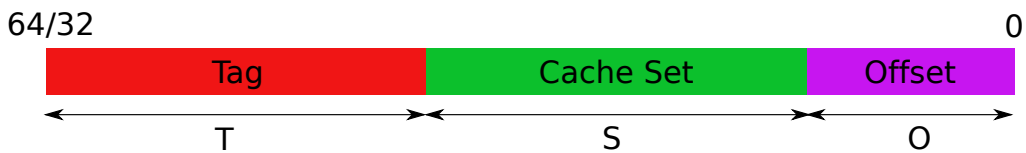
- Paměť RAM je zpravidla rozdělena do:
  - Cachovatelná oblast
  - Necachovatelná oblast
  - Write combining oblast
- Cachovatelná oblast
  - Pro čtení dat použita cache (postupně všechny úrovně)
  - Data jsou do cache přednačítána
  - Data jsou zapisována do cache a rovnou do paměti RAM (write through režim) nebo do paměti RAM až později (write back režim)

Organizace cache

- Přednačítání do plné cache
  - Nějaký řádek je nutné uvolnit (eviction)
  - Exkluzivní cache – řádek lze uvolnit zrušením nemodifikovaného řádku nebo zápisem modifikovaného řádku do cache vyšší úrovně případně do RAM
  - Inkluzivní cache – řádek lze uvolnit zrušením nemodifikovaného řádku nebo zrušením modifikovaného řádku (pokud je řádek cachován ve vyšší úrovni)

Mapování cache na RAM

- Statické v blocích
  - Např. řádka 1 pokrývá RAM 0-999B, řádka 2 pokrývá RAM 1000-1999B, atd.
  - Cache moc nepomůže u kódu a dat blízko sebe
- Asociativní
  - Podobně jako hash tabulka
  - Každý řádek cache má u sebe adresu, kterou cachuje
  - Vyrobit velkou asociativní paměť skoro nelze
- Řešení
  - Kombinace bloku (cache set) a asociativního mapování (tag)
  - Pojem 8-way cache



Cache paměť

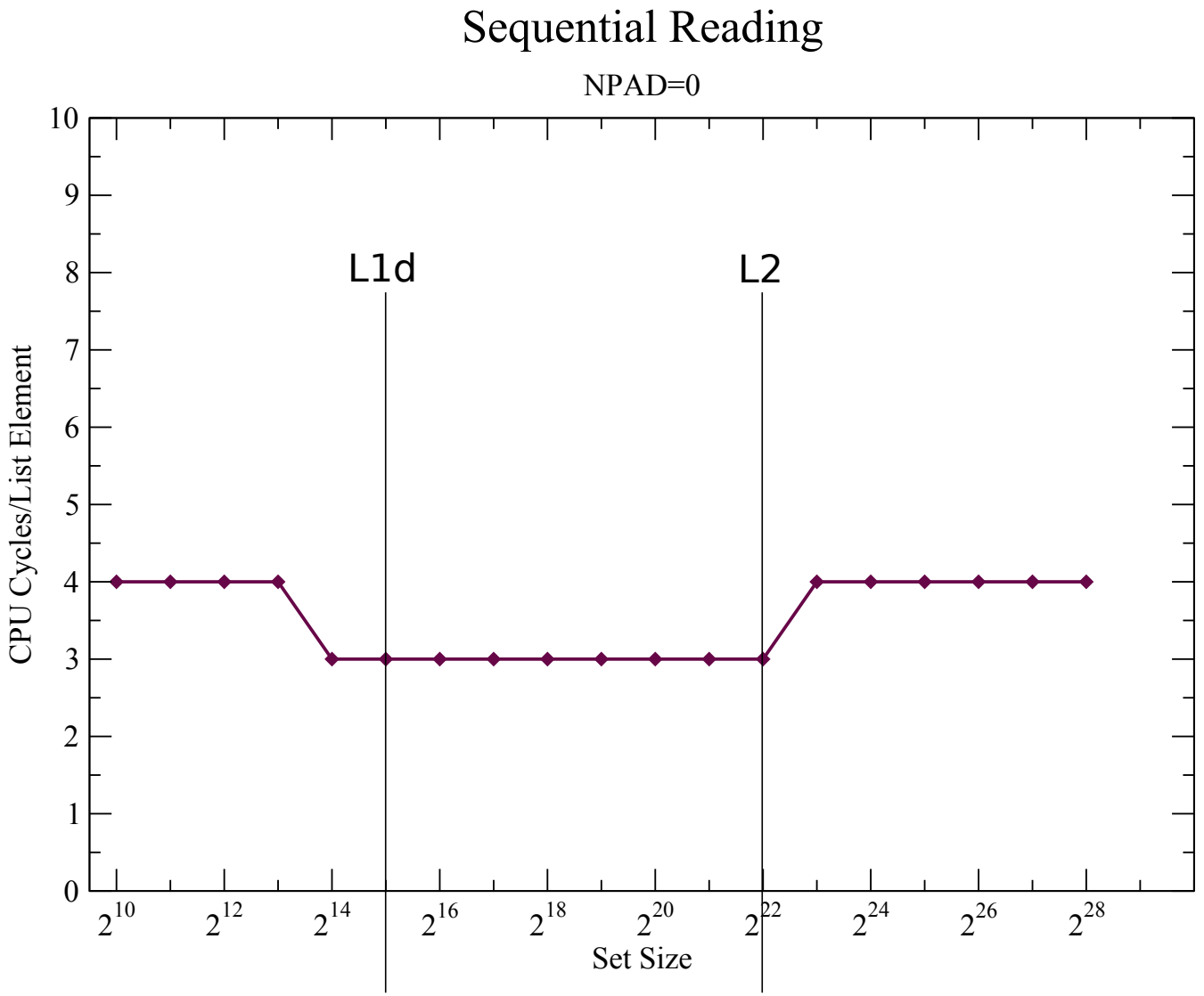
- V případě Intel procesorů, L1 cache neobsahuje x86(64) kód, ale mikrokód, AMD cachuje x86 kód
- L2 cache obsahuje již x86 kód
- L1 cache používá virtuální adresy (překlad virtuální adresy na fyzickou by příliš trval)
- L2 cache používá fyzické adresy

Jednovláknový sekvenční přístup do paměti

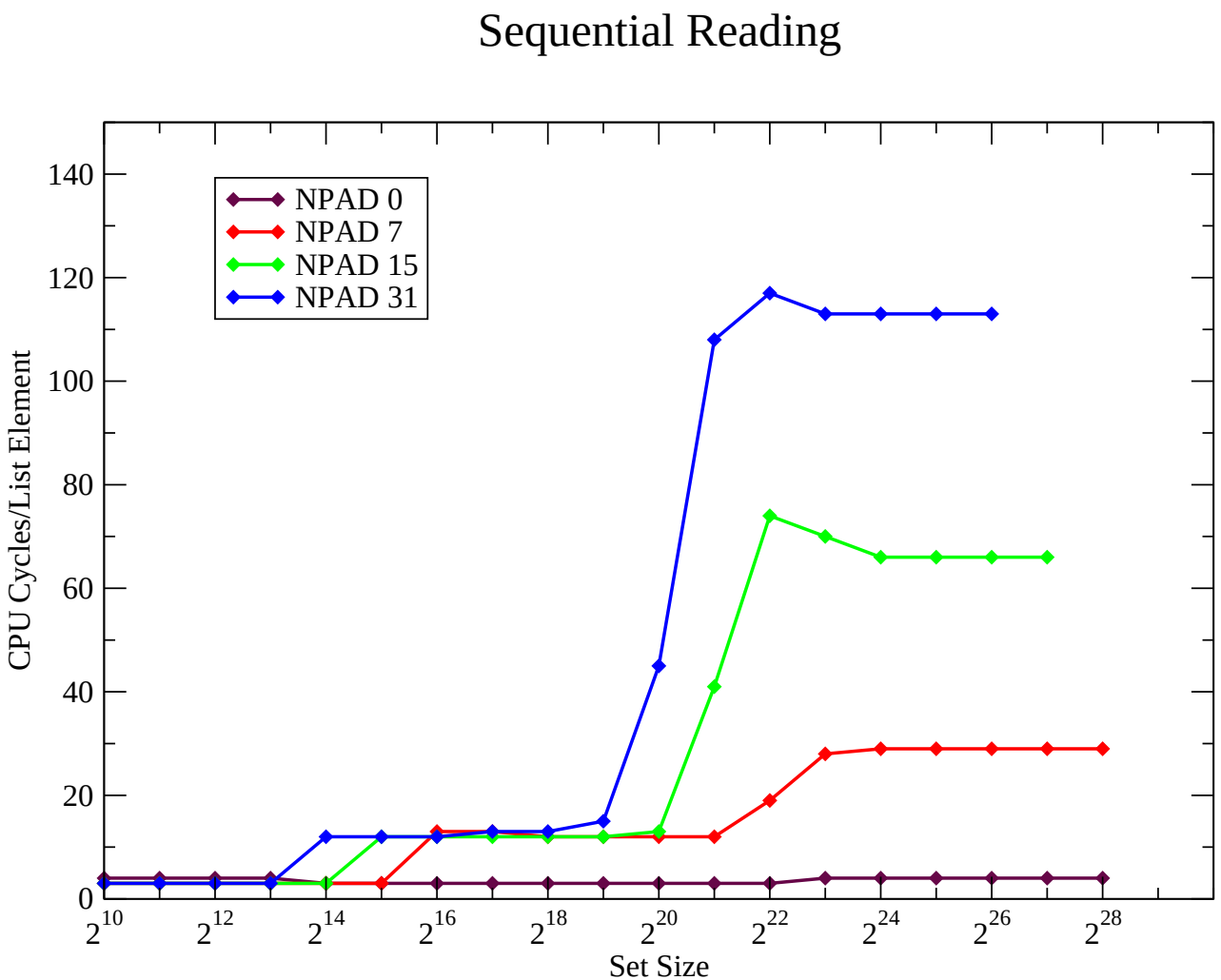
- Mějme program:

```
1  struct l {
2          struct l *next;
3          long int pad[NPAD];
4  } l;
5
6  void
7  iterate_list(struct l *head)
8  {
9          struct l *e = head;
10         long i;
11         for(i = 0; i < nelem*200; i++) {
12                 e = e->next;
13                 asm volatile("" : : "m" (*e));
14         }
15 }
```

Jednovláknový sekvenční přístup do paměti



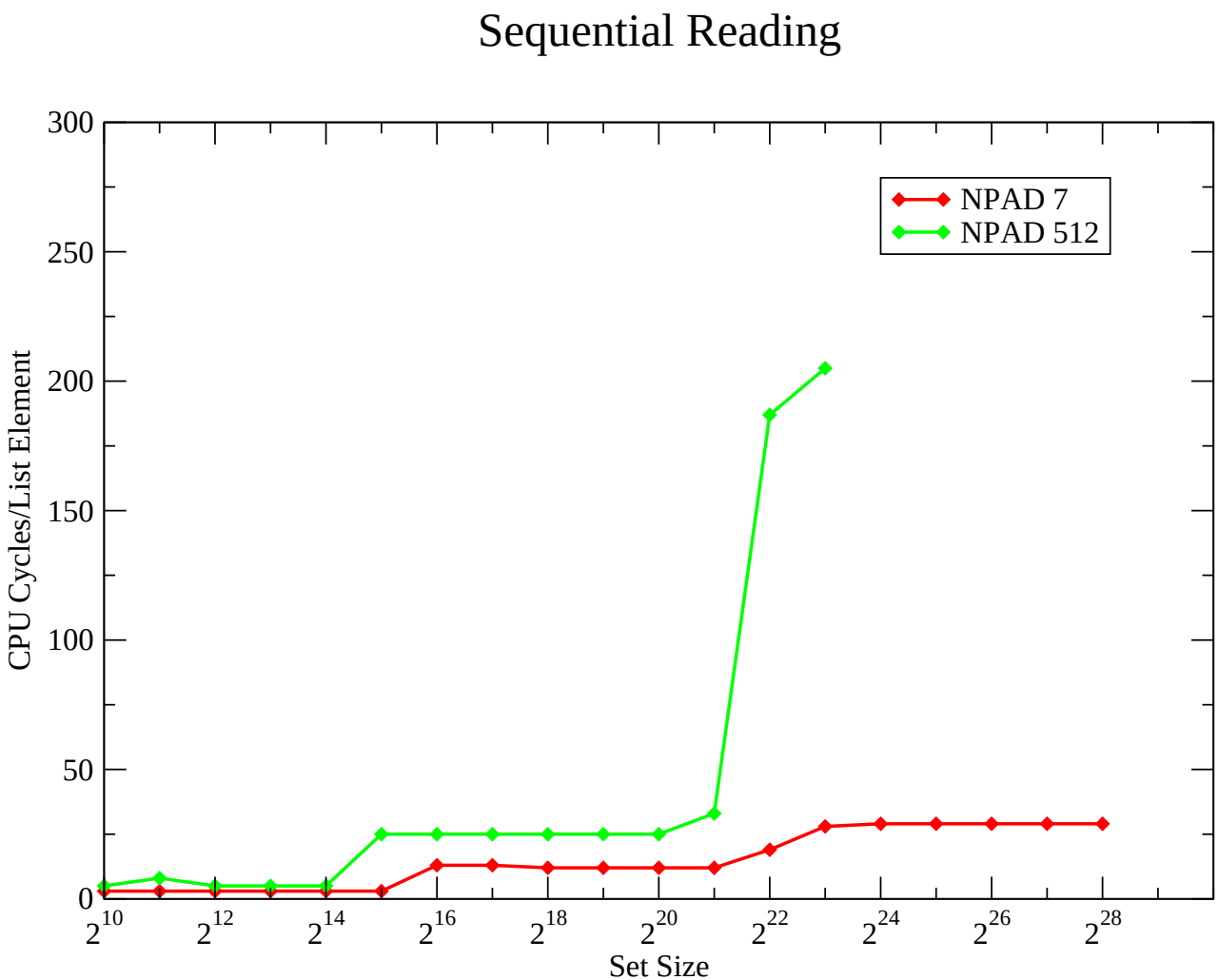
Jednovláknový sekvenční přístup do paměti



Jednovláknový sekvenční přístup do paměti

- Virtuální paměť přináší drobnou komplikaci
- Překlad virtuální adresy na fyzickou znamená nutnost projít tabulky stránek
- Procházení tabulek stránek = náhodný přístup do paměti
- Cache na překlad adres – TLB
- C2D – 256 TLB položek pro data, 128 TLB položek pro instrukce
- Opteron 10. generace, 1024 TLB položek

Jednovláknový sekvenční přístup do paměti



Jednovláknový sekvenční přístup do paměti

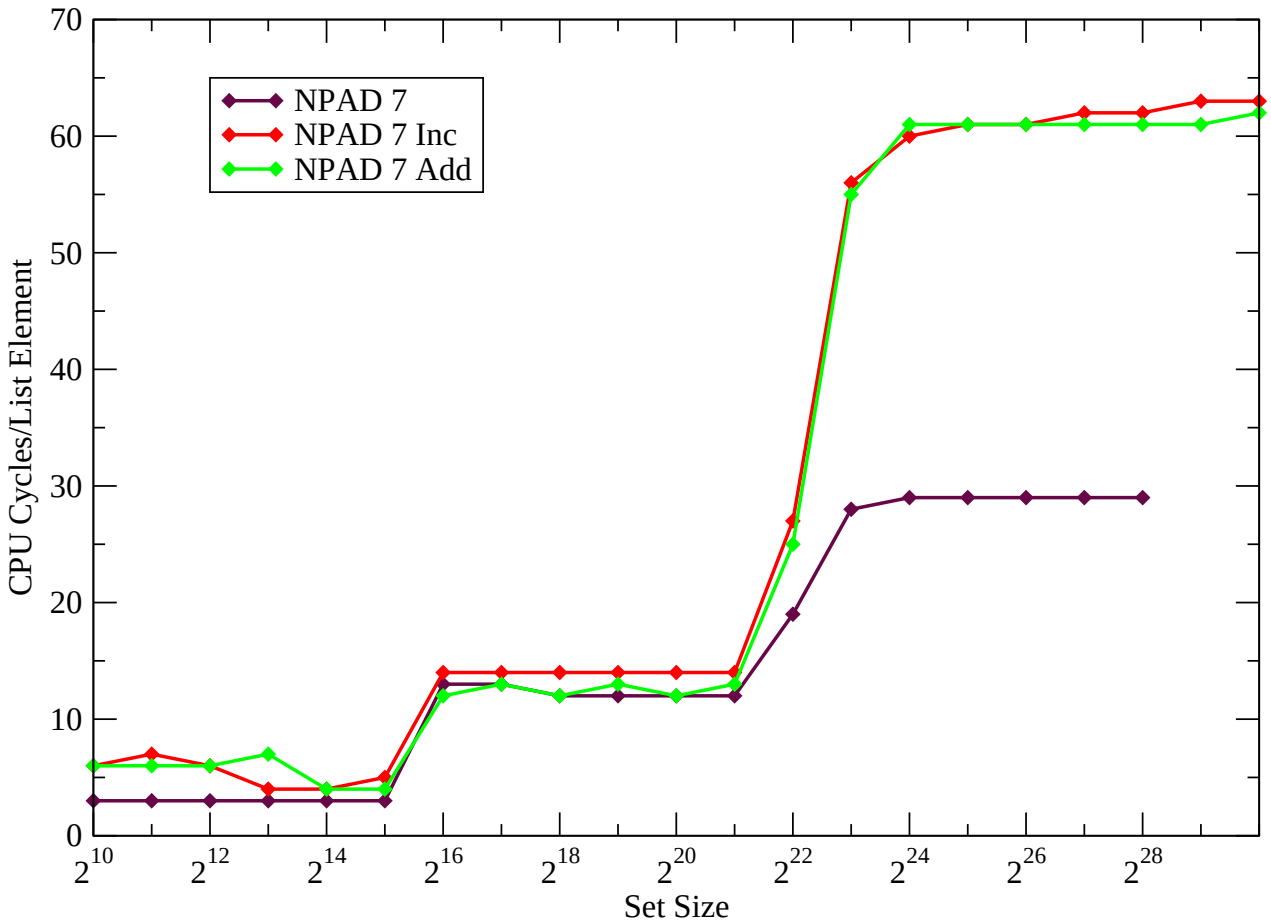
- TLB jsou vázány na konkrétní stránkovací tabulky
- Přepnutí adresního prostoru = zahození všech položek TLB
- Důsledek:
  TLB má málo položek, protože se adresní prostor přepíná velmi často
- Použití větších stránek (2 MB místo 4 kB) redukuje nutný počet TLB položek

Jednovláknový sekvenční přístup do paměti

- Přidáme více práce při procházení seznamu
- Funkce *inc* přičte do prvku jedničku
- Funkce *add* přičte do prvku následující prvek

```
1  void
2  iterate_list_inc(struct l *head)
3  {
4          struct l *e = head;
5          long i;
6          for(i = 0; i < nelem*200; i++) {
7                  e->pad[0]++;
8                  e = e->next;
9                  asm volatile("" : : "m" (*e));
10         }
11 }
12
13 void
14 iterate_list_add(struct l *head)
15 {
16         struct l *e = head;
17         long i;
18         for(i = 0; i < nelem*200; i++) {
19                 e->pad[0]+=e->next->pad[0];
20                 e = e->next;
21                 asm volatile("" : : "m" (*e));
22         }
23 }
```

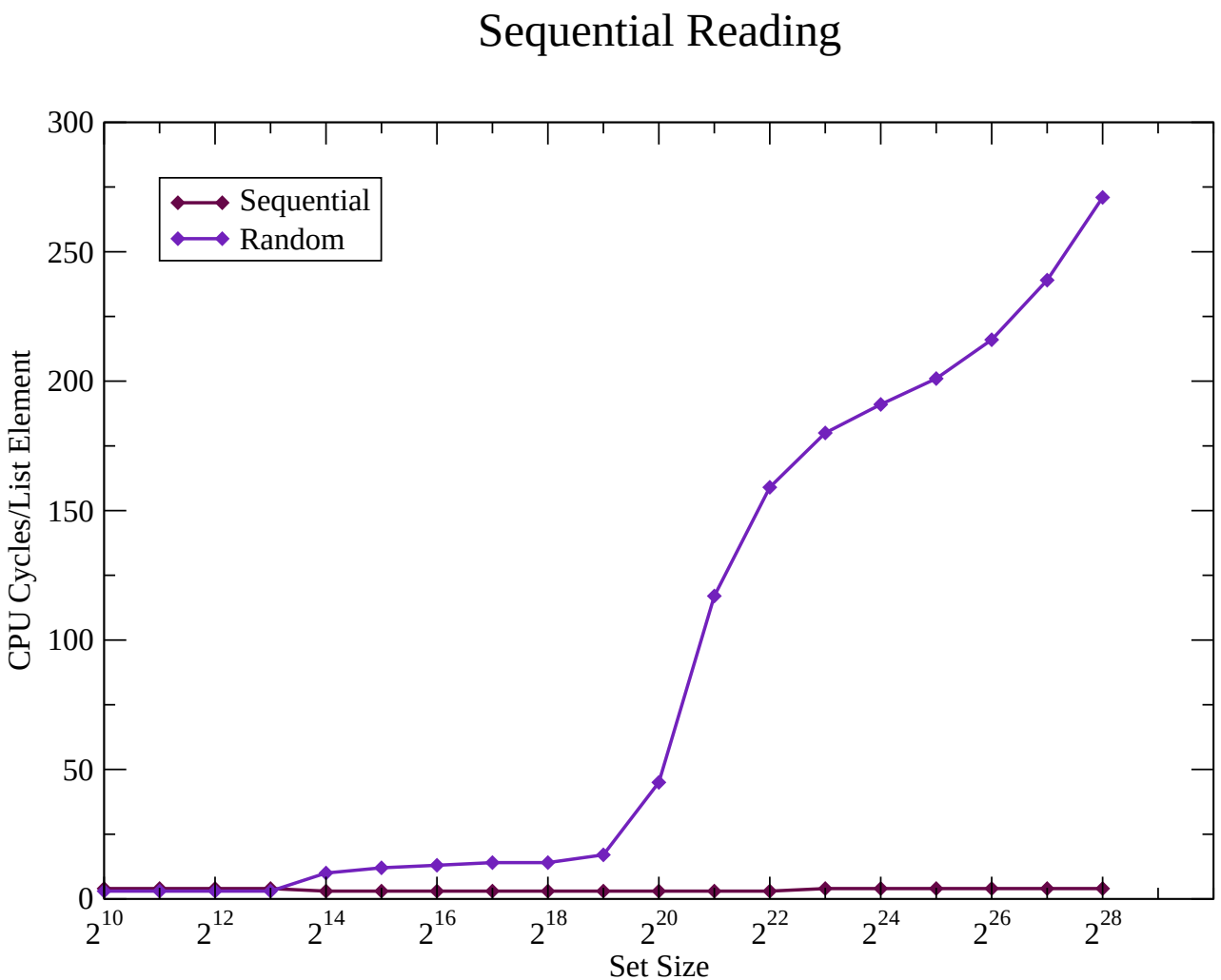
Jednovláknový sekvenční přístup do paměti



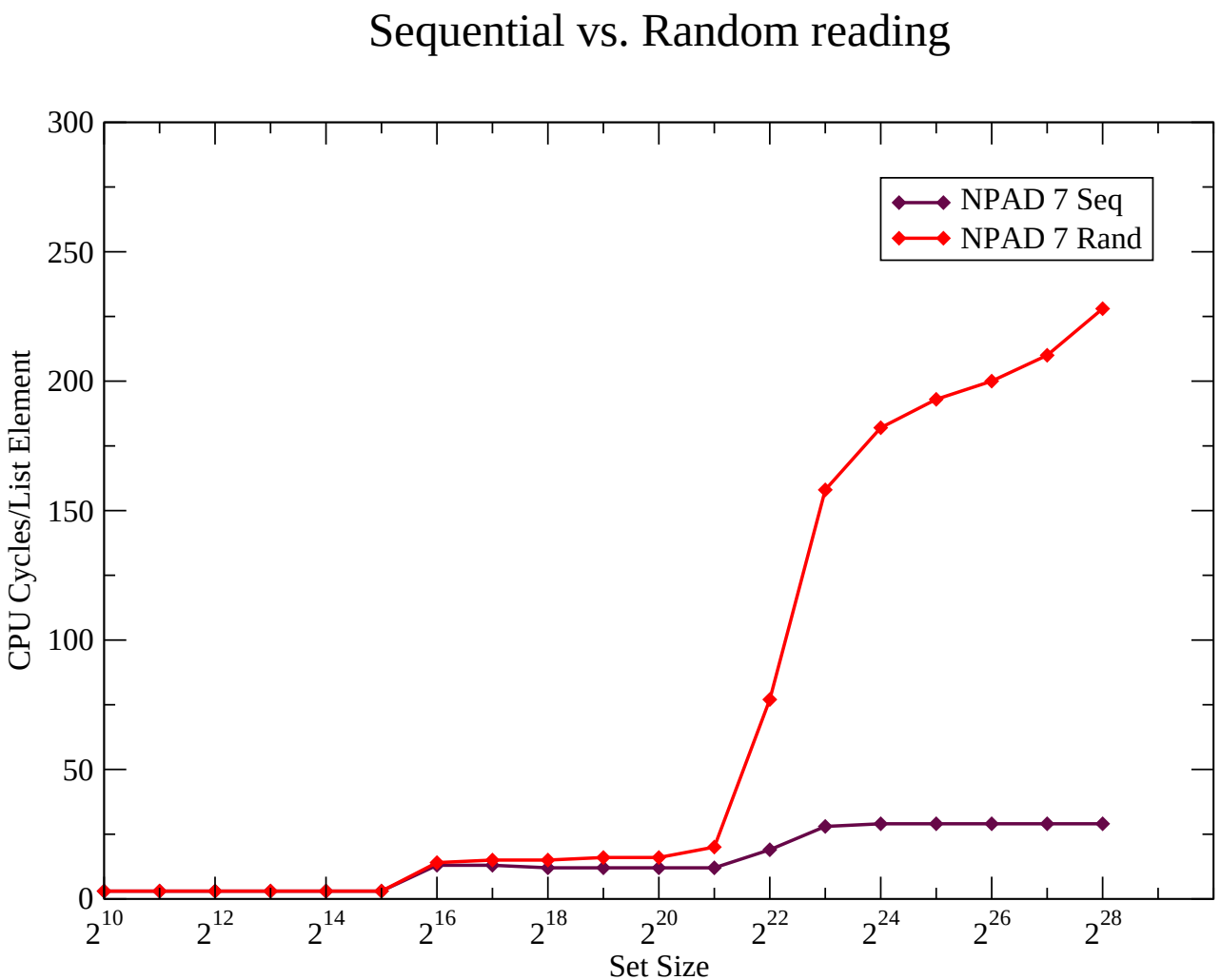
Jednovláknový náhodný přístup do paměti

- Pro sekvenční přístup je schopno CPU sledovat, jaká data budou potřeba v blízké budoucnosti.
- Pro **`NPAD=0`**, paměť v *burst* režimu pošle celou cacheline (64B) = 8 rvků seznamu
- Pro **`NPAD=7`** a další už *burst* režim pošle pouze jeden prvek
- U sekvenčního přístupu je velká šance, že se nebude přeprogramovávat řádek paměti
- U náhodného přístupu je predikce problematická, často je potřeba kompletně naprogramovat přístup k paměti

Jednovláknový náhodný přístup do paměti



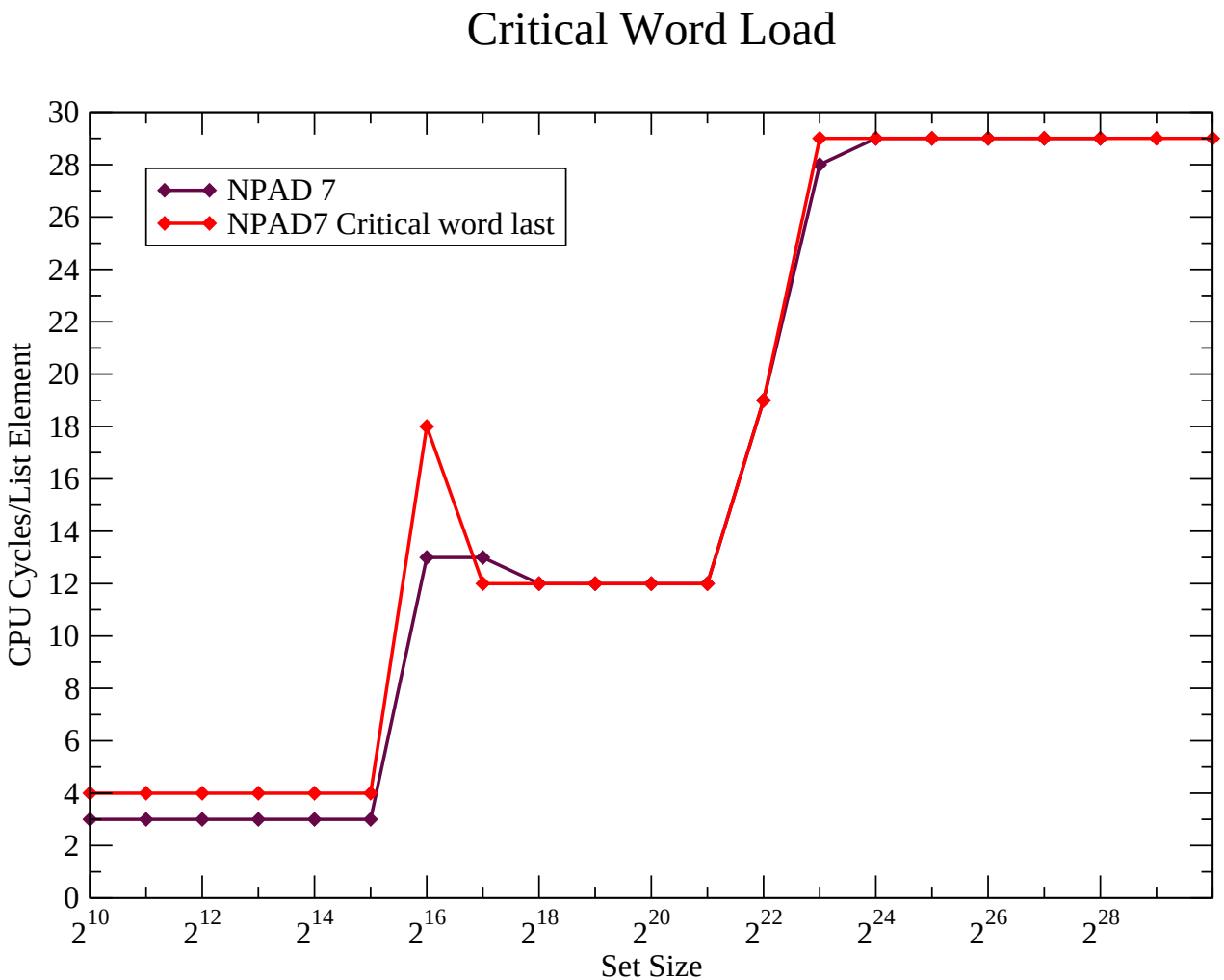
Jednovláknový náhodný přístup do paměti



Critical Word Load

- Cacheline není naplněna instantně ale postupně
- Sběrnicí od paměti nebo cache vyšší úrovně nelze poslat 64 bytů v jednom cyklu
- Procesor je schopen pracovat i s částečně naplněnou cacheline
- Důsledek:
  - Záleží na pozici slova v cache, často používané položky je lepší umísťovat na začátek

Critical Word Load



Zápisy do paměti

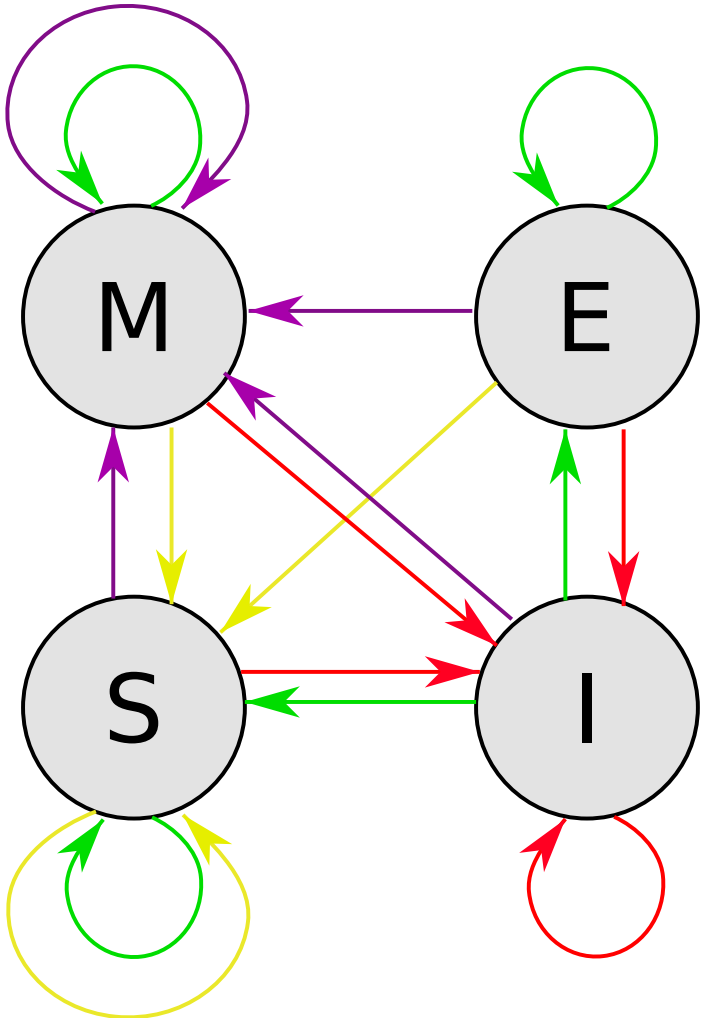
- L1 cache je prakticky vždy nesdílená mezi jádry multiprocesorového systému
- Cachováním položek paměti může dojít k nekonzistencím
- Existují protokoly koherence cache
  - Je nepraktické umožnit procesorům přímý přístup do cache jiného procesoru
  - Procesory odposlouchávají na paměťové sběrnici a mají představu o tom, co ostatní procesory cachují

MESI protokol

- Každý řádek cache je v jednom ze čtyř stavů:
  - Modified: lokální procesor modifikoval řádku cache, která se nenachází v žádné cache jiného procesoru
  - Exclusive: řádka není modifikovaná a není uložena v cache jiného procesoru
  - Shared: řádka není modifikovaná a může existovat v cache jiného procesoru
  - Invalid: řádka není platná

Zápisy do paměti

local read
local write
remote read
remote write

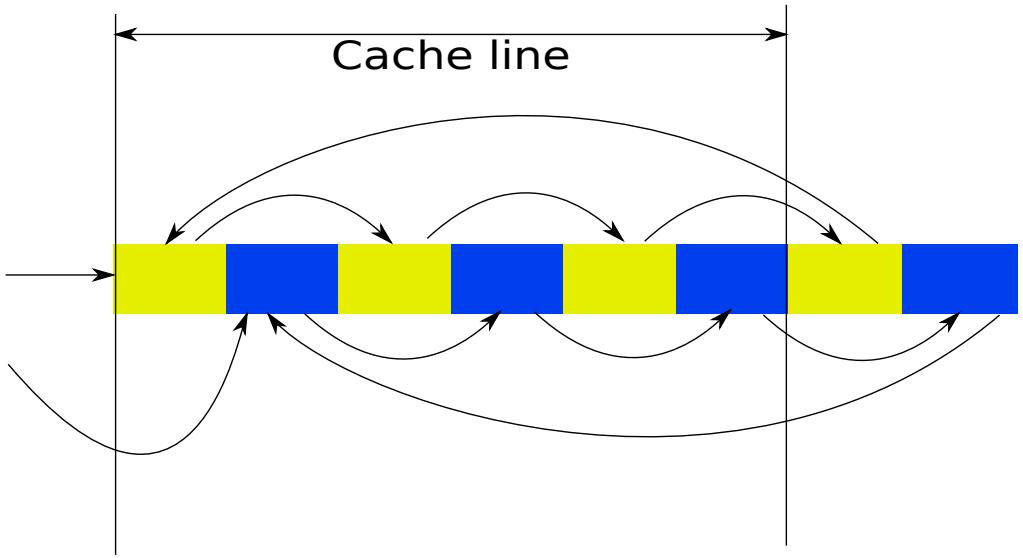


Přechody stavů

- Request for ownership (RFO) zpráva
  - Modified $\Rightarrow$ Invalid
  - Shared $\Rightarrow$ Modified

Zápisy do paměti

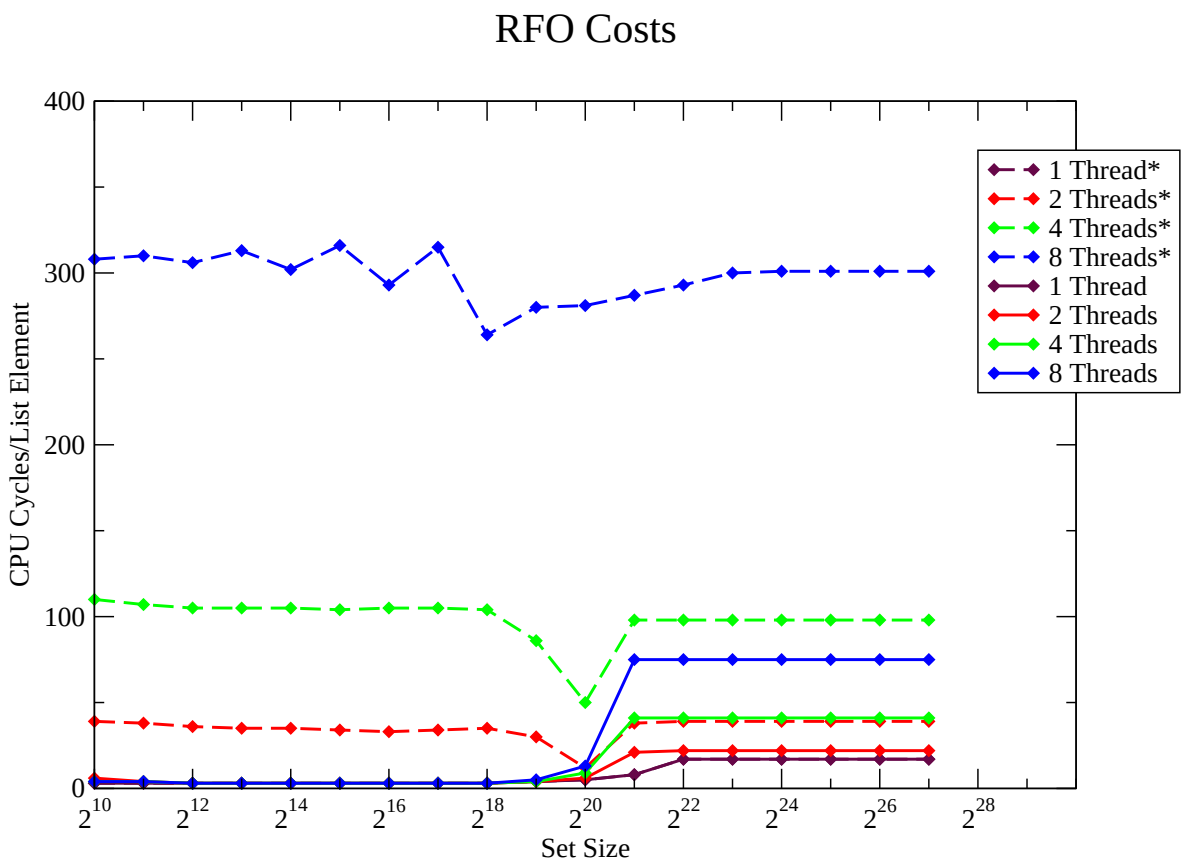
- Mějme seznam, který leží v paměti následovně



- Každý seznam prochází jiné vlákno
- Použijeme *inc* variantu
  tj. položky cache se mění střídavě různými CPU
- Velké množství RFO zpráv (Modified ⇒ Invalid)

Zápisy do paměti

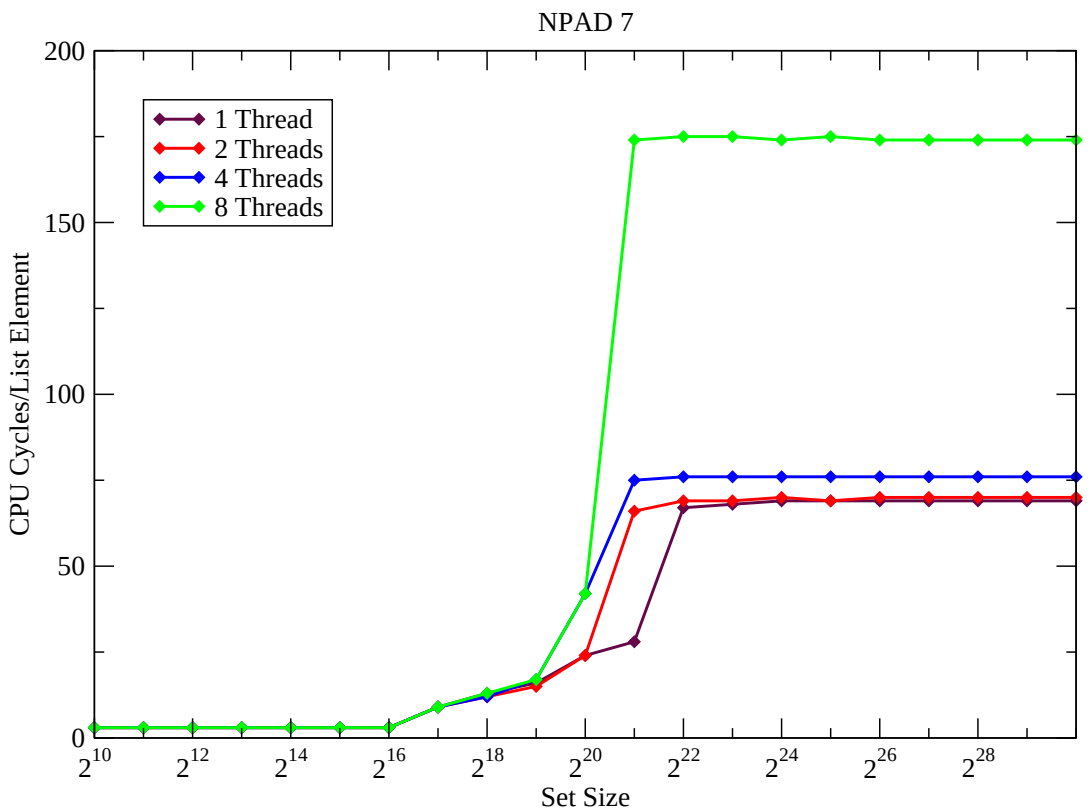
QuadCore Opteron, 64kB L1, 512kB L2, 2MB L3 Cache
NPAD 1, Inc



bez * je původní seznam

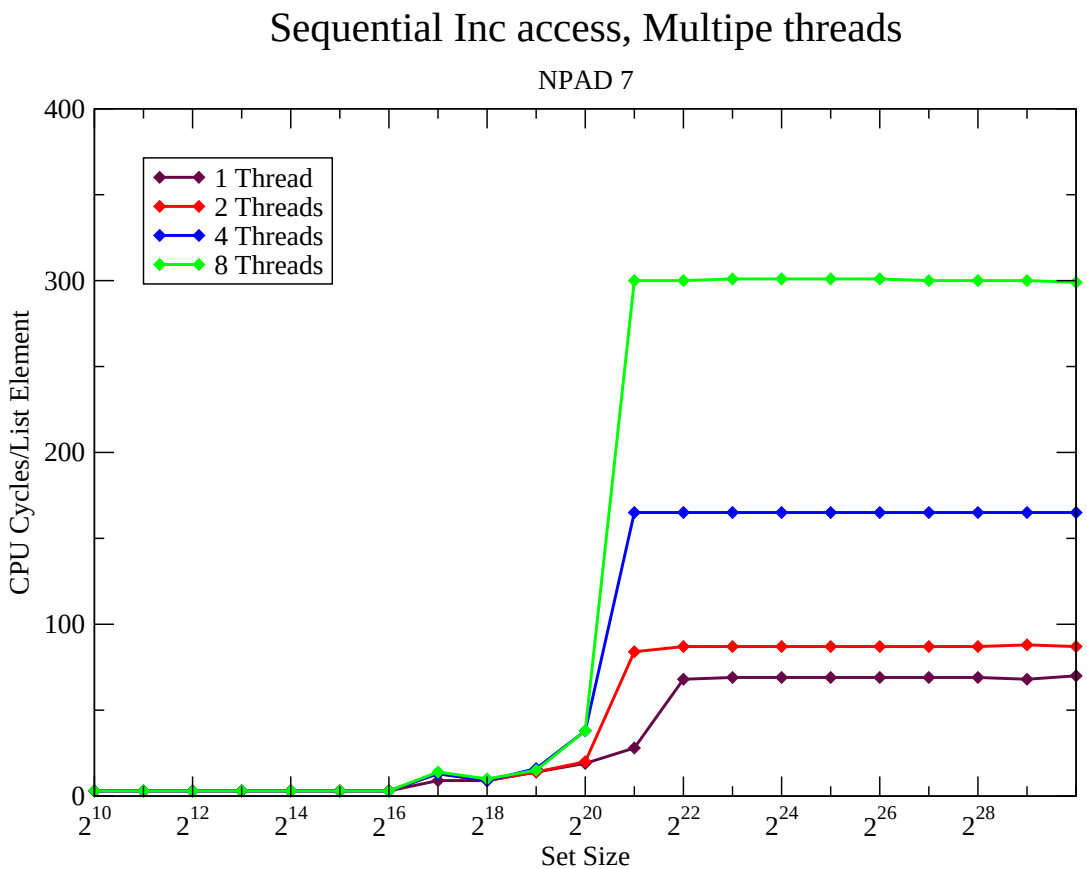
Práce ve vláknech

QuadCore Opteron, 64kB L1, 512kB L2, 2MB L3 Cache
NPAD 7



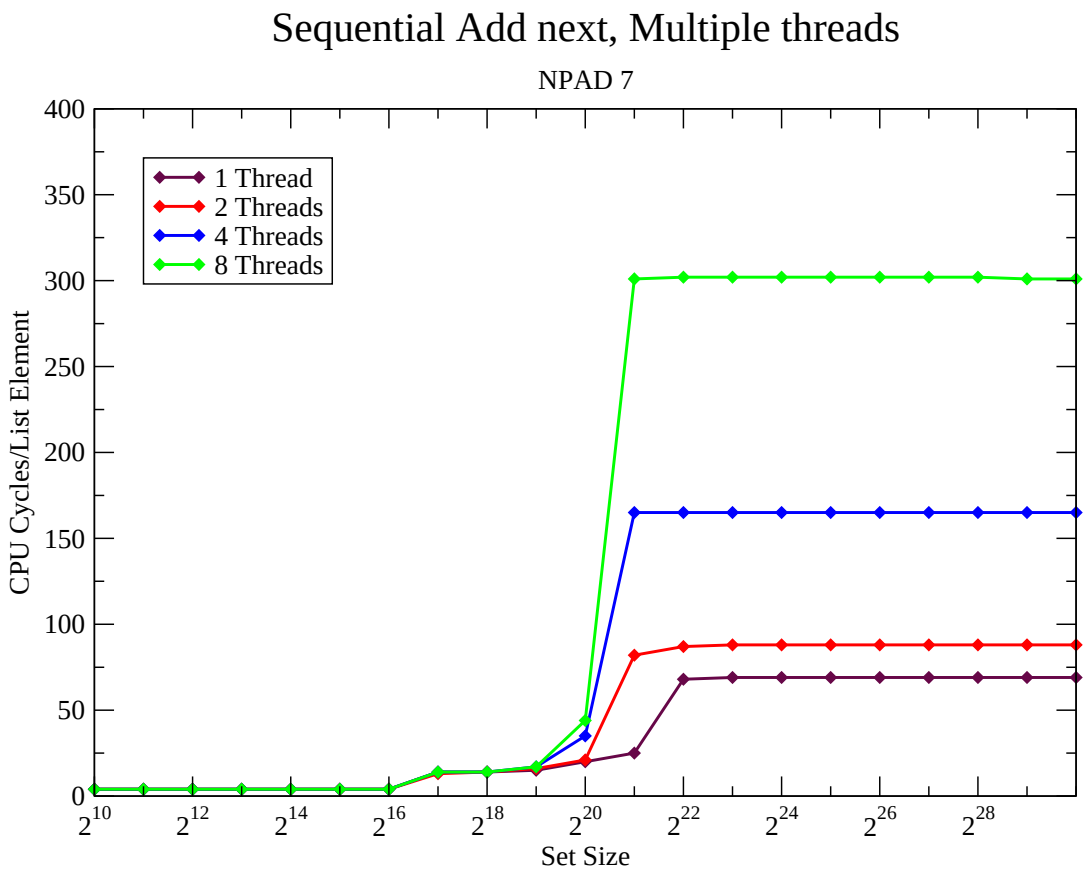
Práce ve vláknech

QuadCore Opteron, 64kB L1, 512kB L2, 2MB L3 Cache
NPAD 7, Increment



Práce ve vláknech

QuadCore Opteron, 64kB L1, 512kB L2, 2MB L3 Cache
NPAD 7, Add next



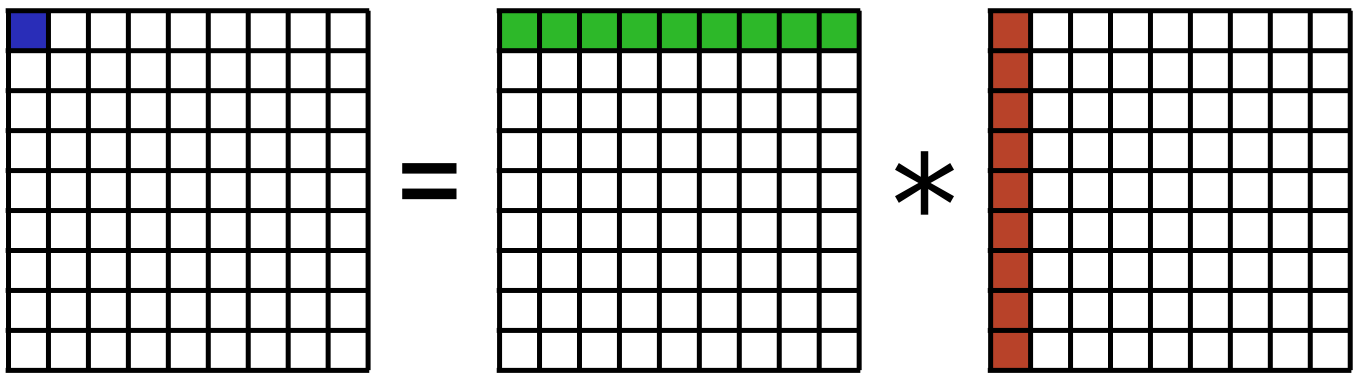
Násobení matic

- Mějme 2 velké matice 1000×1000 typu double
- Potřebujeme je vynásobit
- Naivní algoritmus

```
1 for(i=0; i < N; ++i)
2         for(j=0; j < N; ++j)
3                 for(k=0; k < N; ++k)
4                         res[i][j] += mul1[i][k] * mul2[k][j];
```

- Lze vidět, že matici **mul2** procházíme po sloupcích = nesekvenční přístup

Násobení matic



Násobení matic

- Matici **mul2** můžeme transponovat

```
1 for(i=0; i < N; ++i)
2         for(j=0; j < N; ++j)
3                 tmp[i][j] = mul2[j][i];
4
5 for(i=0; i < N; ++i)
6         for(j=0; j < N; ++j)
7                 for(k=0; k < N; ++k)
8                         res[i][j] += mul1[i][k] * tmp[j][k];
```

- Čas nutný na transpozici započítáme do běhu programu

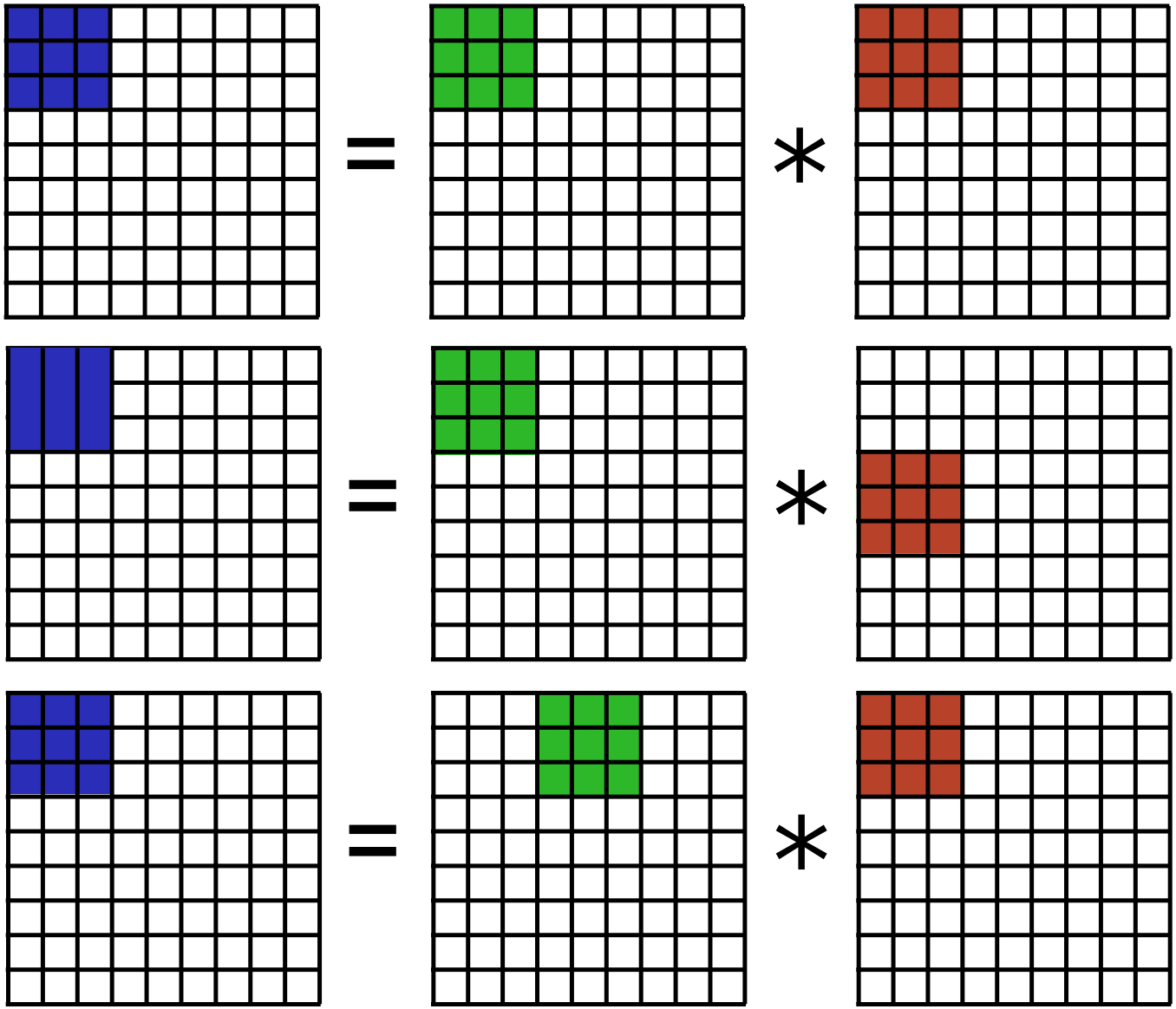
	Original	Transposed
Opteron 2356		
Cycles	54,415,069,438	11,303,294,945
Relative	100%	20%
C2D E6850		
Cycles	10,543,642,668	3,822,534,279
Relative	100%	36%

Násobení matic

- Ne vždy je možné udělat kopii matice, není to ani optimální řešení
- Lze rozdělit matici do bloků SM×SM a násobit bloky

```
1 for(i=0; i < N; i+= SM)
2     for(j=0; j < N; j+=SM)
3         for(k=0; k < N; k+= SM)
4             for(i2 = 0, rres = &res[i][j], rmul1 = &mul1[i][k];
5                                 i2 < SM; ++i2, rres += N, rmul1 += N)
6                 for(k2 = 0, rmul2 = &mul2[k][j]; k2 < SM; ++k2,
7                                         rmul2 += N)
8                     for(j2 = 0; j2 < SM; ++j2)
9                         rres[j2] += rmul1[k2] * rmul2[j2];
```

Násobení matic



Násobení matic

- Dále je možno využít vektorových instrukcí, které zpracují několik operací najednou

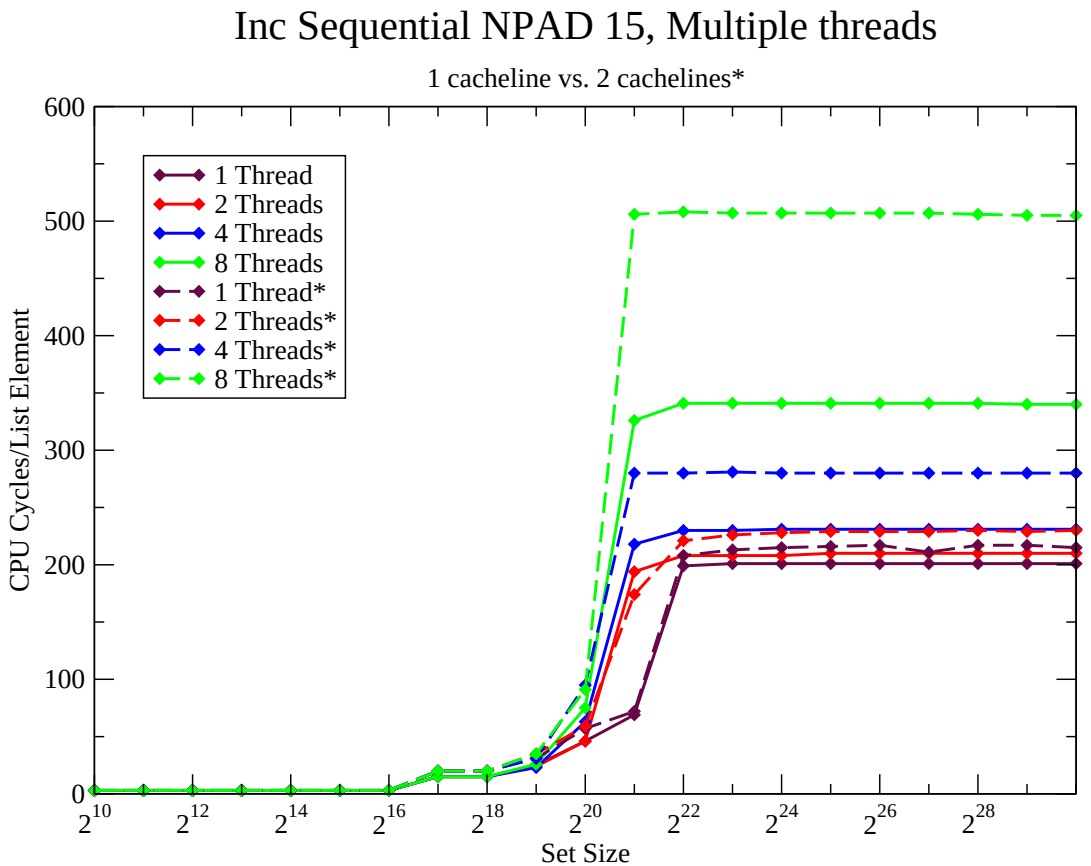
```
1  for(i=0; i < N; i+= SM)
2      for(j=0; j < N; j+=SM)
3          for(k=0; k < N; k+= SM)
4              for(i2 = 0, rres = &res[i][j], rmul1 = &mul1[i][k];
5                                  i2 < SM; ++i2, rres += N, rmul1 += N) {
6                  _mm_prefetch(&rmul1[8], _MM_HINT_NTA);
7                  for(k2 = 0, rmul2 = &mul2[k][j]; k2 < SM; ++k2,
8                                       rmul2 += N) {
9                      __m128d m1d = _mm_load_sd(&rmul1[k2]);
10                     m1d = _mm_unpacklo_pd(m1d, m1d);
11                     for(j2 = 0; j2 < SM; j2+=2) {
12                         __m128d m2 = _mm_load_pd(&rmul2[j2]);
13                         __m128d r2 = _mm_load_pd(&rres[j2]);
14                         _mm_store_pd(&rres[j2],
15                                 _mm_add_pd(_mm_mul_pd(m2, m1d), r2));
16                     }
17                 }
18             }
```

Násobení matic

	Original	Transposed	Sub-matrix	Vectorized
Opteron 2356				
Cycles	54,415,069,438	11,303,294,945	5,237,161,135	5,178,595,240
Relative	100%	20%	9%	9%
C2D E6850				
Cycles	10,543,642,668	3,822,534,279	3,056,154,579	1,442,368,584
Relative	100%	36%	28%	13%

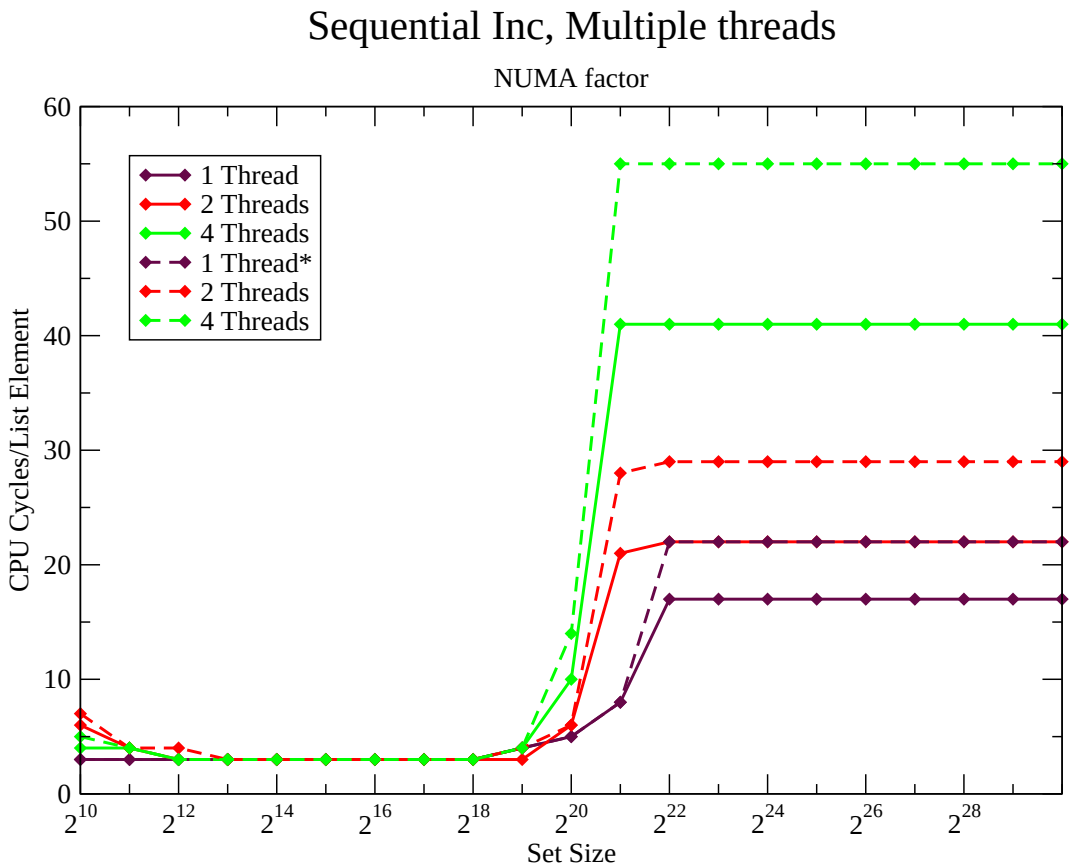
Blízkost dat

- Rozdíly mezi malou strukturou zaujímající 1 cacheline a mezi větší přes více cachelines je znát



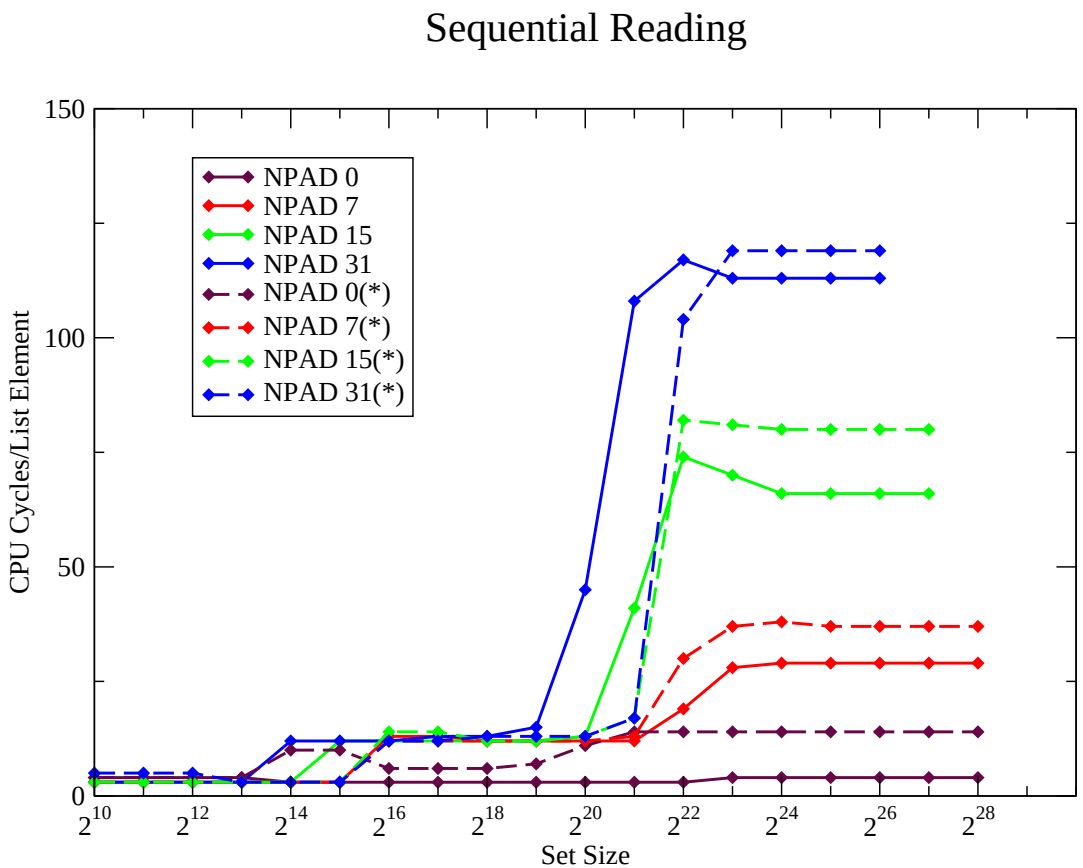
Blízkost dat

- Blízkost dat se vztahuje i na NUMA systémy, přístup do nelokální paměti je dražší



Blízkost dat

- Blízkost malých dynamických kusů paměti narušuje **`malloc()`**
- Každý alokovaný kus paměti má 16 bytů hlavičku a případně patičku pro zarovnání na 16 bytů

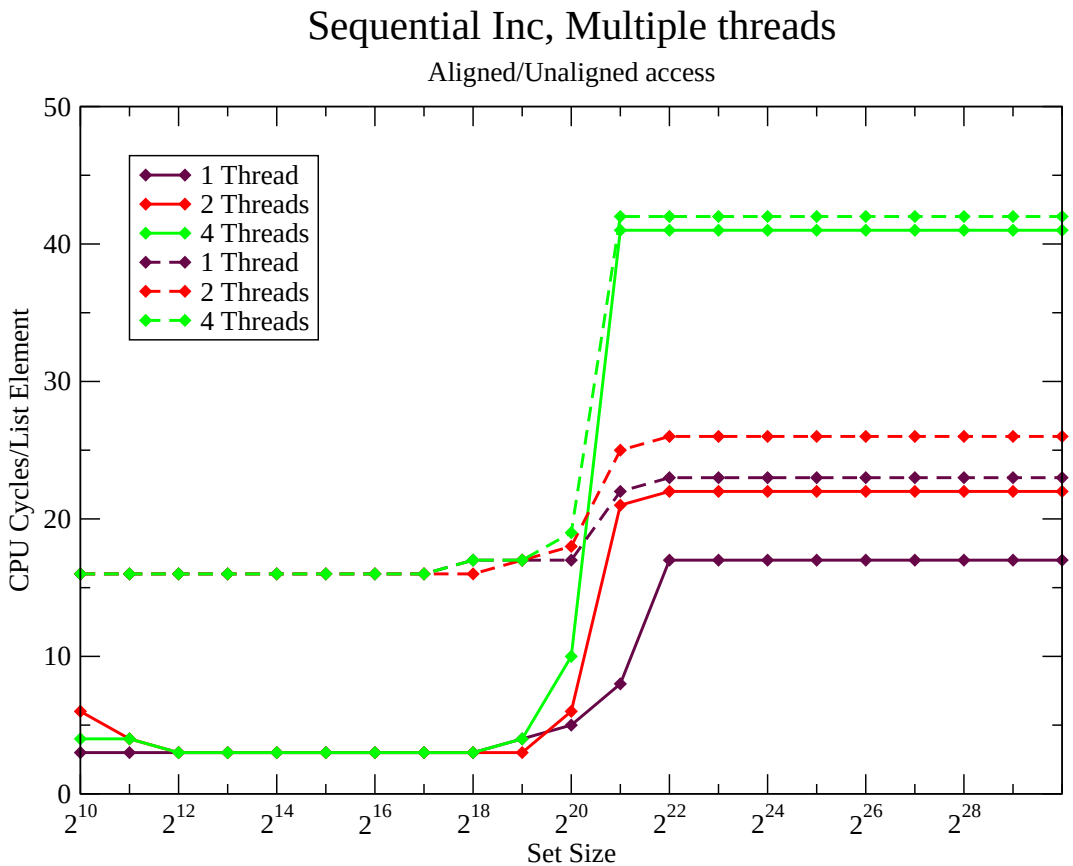


bez * je ruční alokace v **`malloc()`** bloku

Blízkost dat

- U blízkosti dat je nutné respektovat jejich vhodné zarovnání, zejména zarovnání na cacheline a slova na hranici slov
  - Tj. 8bytový long na hranici 8 bytů
  - Struktury na 64 bytů
  - Kódy funkcí na 64 bytů
- Pro zarovnání dat existuje volání **`int posix_memalign(void **memptr, size_t align, size_t size)`**
- Pro zarovnávání golbálních dat gcc nabízí **`__attribute__ ((aligned(64)))`**

Blízkost dat



Blízkost instrukcí

- Inlining funkcí nevede vždy k nárůstu rychlosti
- Penalta za cache miss je proti volání funkce obrovská
- Optimalizace větvení
  - Překladač může optimalizovat podmínky podle toho, zda se většinou provede či nikoliv
  - Konstrukty v jádře `if(likely(a>1)) { }`, `if(unlikely(a>1)) { }`
  - `#define likely(expr) __builtin_expect(!!(expr),1)`
  - `#define unlikely(expr) __builtin_expect(!!(expr),0)`

Nástroje pro analýzu

- **valgrind** – **cachegrind** – analýza využití cache, cache miss apod.
- **valgrind** – **massif** – analýza využití paměti v čase
- podobně **memusage**
- gcc je schopno analyzovat úspěšnost branchprediction **-fprofile-generate**, **-fprofile-use**
- Informace o cache hits/miss, TLB hit/miss lze získat přímo z CPU **http://user.it.uu.se/~mikpe/linux/perfctr/**

Výhled do budoucnosti

- TLB
  - Tagovaná TLB cache, není nutné vylití TLB při přepnutí kontextu
- Transakční paměť
  - Podpora LL/SC instrukcí (Nemají ABA problém)
  - Transakční L1t cache
  - Rozšíření MESI protokolu – není technologicky náročné