Paralelní architektury se sdílenou pamětí

Tomáš Oberhuber
tomas.oberhuber@fjfi.cvut.cz

7. března 2024

Videa na Youtube:

První část

Druhá část

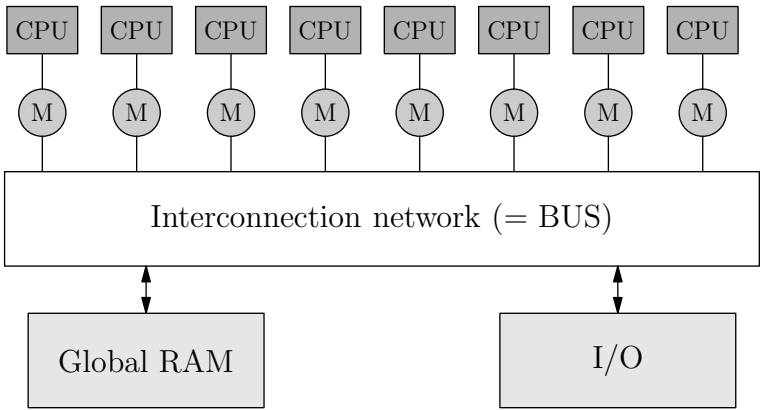
Video na Youtube

Multiprocesorové architektury I.

Multiprocesor se skládá z

- několika plnohodnotných procesorů
- sdíleného adresového prostoru
  - stejné adresy u dvou různých CPU ukazují na stejné místo v adr. prostoru

Multiprocesorové systémy se sdílenou pamětí I.



Multiprocesorové systémy se sdílenou pamětí II.

Mainframe IBM S360 Model 65 – 1965

- zřejmě první dvouprocesorový systém

Multiprocesorové systémy se sdílenou pamětí II.

- tyto systémy jsou odvozeny z jednoprocesorového systému pouhým přidáním dalších CPU propojených **sběrnicí** (BUS)
- všechny procesory jsou rovnocenné
  - odtud název - **symmetric multiprocessor - SMP**
- přístup do globální paměti je vždy stejně rychlý
  - odtud název - **uniform memory acces multiprocessor - UMA**

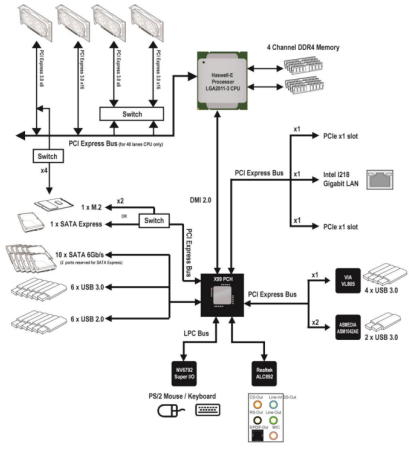
Příkladem SMP jsou dnes běžné vícejádrové PC.

SMP architektury

- ukážeme si některé současné mikroarchitektury založené na architektuře se sdílenou pamětí
  - Intel Alder Lake
  - AMD Zen
  - Apple M1/M2

Procesory Intel

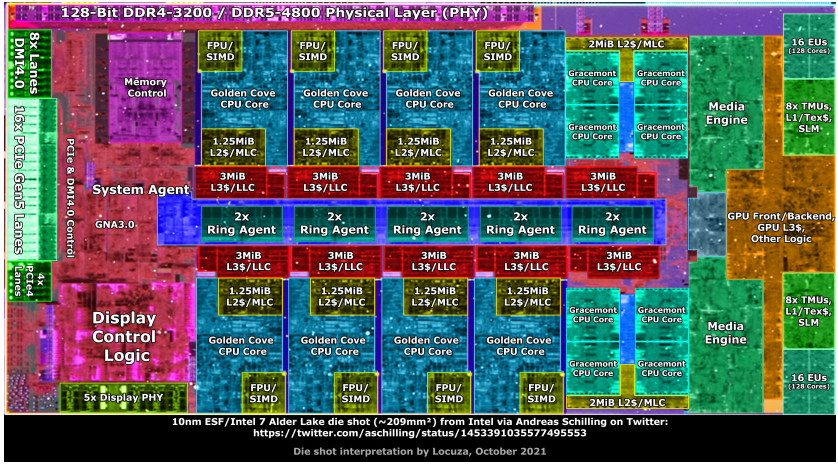
Intel Haswell a chipset X99



Zdroj: `http://www.anandtech.com/show/8557/x99-motherboard-roundup-asus-x99-deluxe-gigabyte-x99-ud7-ud5-asrock-x99-ws-msi-x99s-sli-plus-intel-haswell-e/`

Procesory Intel

Intel Alder Lake Die



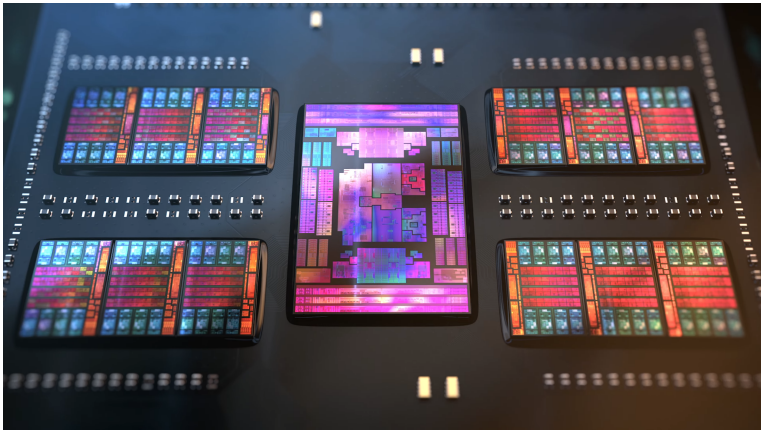
Procesory Intel

Procesory architektury Alder Lake:

- až 8 výkonných jader (16 vláken)
  - frekvence 1-5.5 GHz
  - L1 cache 32kB instrukční a 48 kB datová
  - L2 cache 1.25 MB
- a až 8 efektivních jader (8 vláken)
  - frekvence 0.7-4 GHz
  - L1 cache 64kB instrukční a 32 kB datová
  - L2 cache 2 MB
- L3 cache 30 MB

Procesory AMD

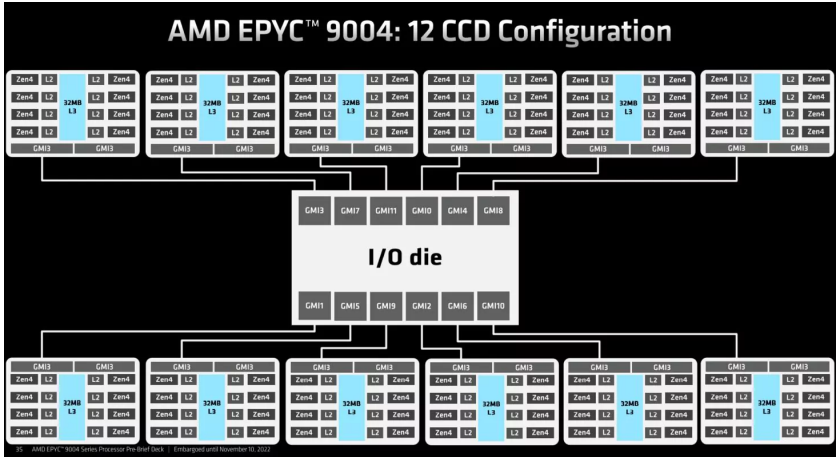
AMD Epyc 9004 Genoa Zen 4



Zdroj: WCCFTECH

Procesory AMD

AMD Epyc 9004 Genoa Zen 4

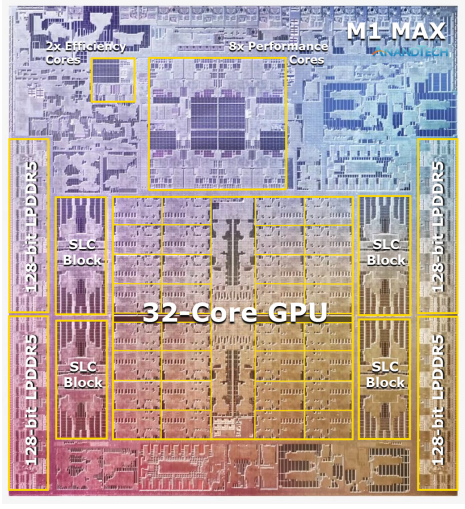


Procesory AMD

- procesor je složený z tzv. čipletů tj. několika čipů
- obsahuje
  - 96 jader pro 192 vláken až na 3.55 GHz
  - 96 x 32kB L1 cache
  - 96 x 1 MB L2 cache
  - 384 (=4x96) MB L3 cache
  - 12 kanálů pro DDR5 paměti, 460 GB/s

Procesory Apple Silicon

Apple M1



Zdroj: Anandtech

Procesory Apple Silicon

- jde o procesory postavené na architektuře ARM
- je to tzv. System-on-chip, tj. obsahuje CPU, GPU i systemovou paměť
- tím odpadají dlouhé sběrnice mezi CPU a pamětí, což napomáhá větší efektivitě
- paměť pak ale nejde rozšiřovat
- M1 Ultra obsahuje
  - 20 CPU jader, 16 výkonných na 3.2 GHz a 4 efektivní na 2.0 GHz
  - 16x(192kB + 128kB) L1 cache a 4x(128kB + 64kB) cache
  - 8 výkonných jader se dělí do dvou klastrů, každý má 12 MB L2 cache
  - 2 efektivní jádra mají 4MB L2 cache
  - 96 MB L3 cache

Procesory Apple Silicon

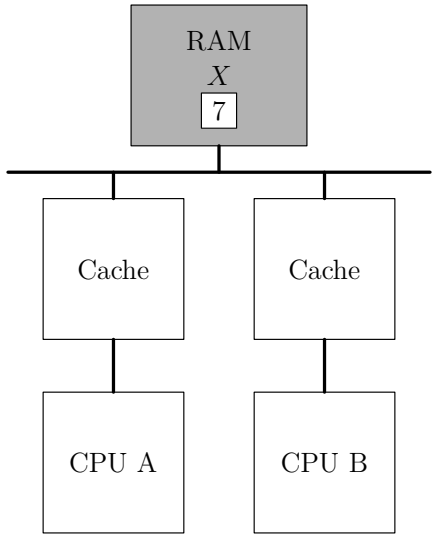
Paměti na čipech M1 Max a M1 Ultra

- až 2x64GB low-power LPDDR5
- maximální datová propustnost až 2x400 GB/s

Cache coherence problem

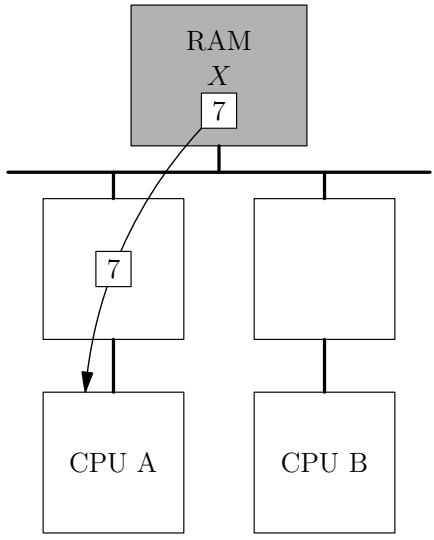
- již víme, že paměťové moduly jsou až 200x pomalejší než processor
- i jedno jádro tak dokáže plně vytížit paměťový subsystém
- pro efektivní využití více jader je (až na výjimky) nezbytné optimalizovat přístupy do paměti
- vše se nyní výrazně komplikuje tzv. *cache coherence* problémem

Cache coherence problem



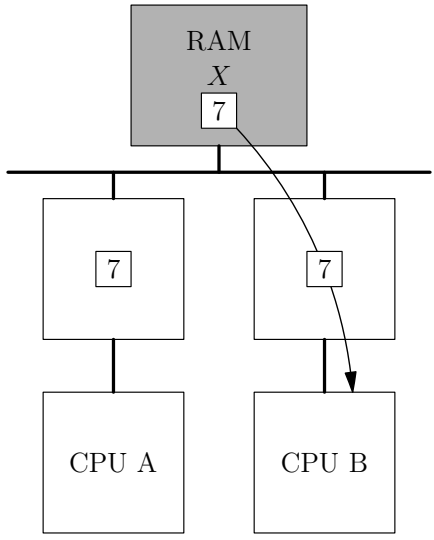
Cache coherence problem

CPU A načítá proměnnou X



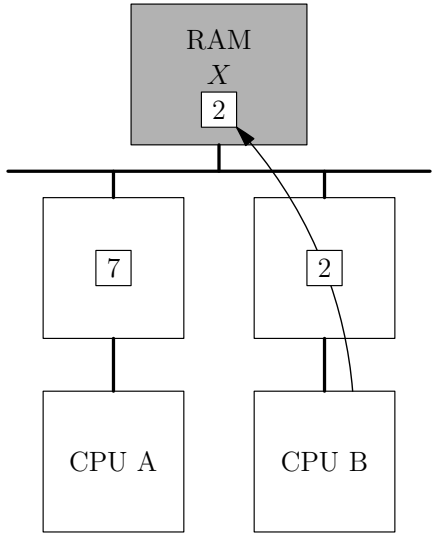
Cache coherence problem

CPU B načítá proměnnou X



Cache coherence problem

CPU B zapisuje 2 do X, což se neprojevuje v cache procesoru A



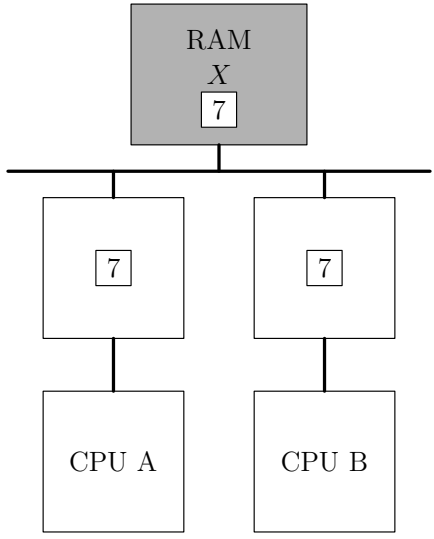
Cache coherence problem

Cache coherence problem je řešen hardwarově. Existují dva způsoby řešení:

- **update protocol**
- **invalidate protocol**

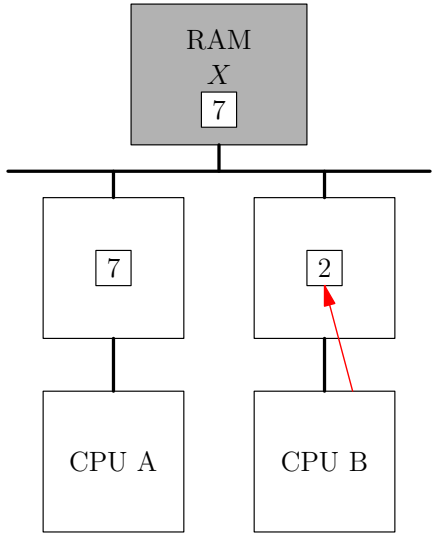
Cache coherence problem

Update protocol - *X* je sdílená proměnná



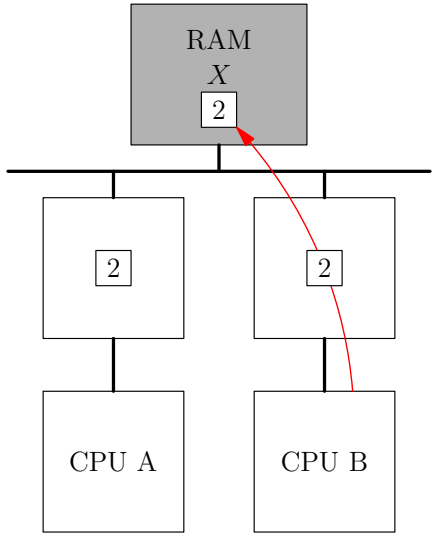
Cache coherence problem

Procesor *B* zapisuje 2 do *X* ve své cache, ...



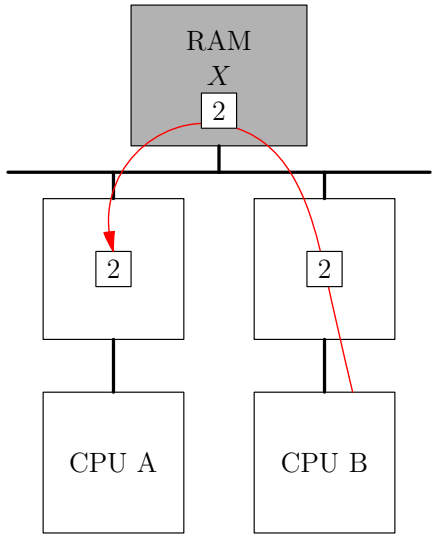
Cache coherence problem

... současně mění hodnotu X i v RAM ...



Cache coherence problem

... a v cache procesoru *A*.



Cache coherence problem

Nevýhody *update* protokolu:

- pokud procesor A načte proměnnou X jen jednou na začátku, a potom s ní pracuje pouze procesor B, zbytečně pokaždé posílá novou hodnotu

Cache coherence problem

Nevýhody *update* protokolu:

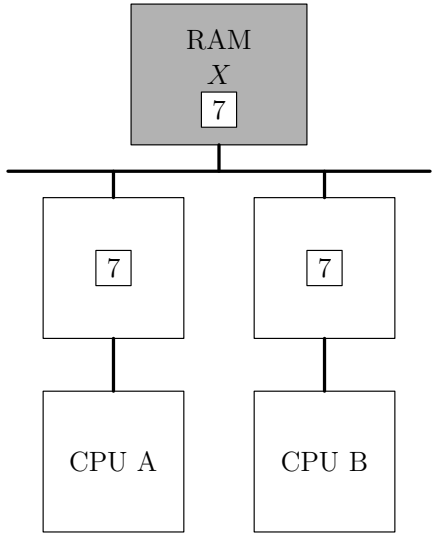
- pokud procesor *A* načte proměnnou *X* jen jednou na začátku, a potom s ní pracuje pouze procesor *B*, zbytečně pokaždé posílá novou hodnotu

V současnosti se častěji používá *invalidate* protokol. Nazývá se také *MESI* protocol podle stavů cache lines:

1. Modified
2. Exclusive – proměnná není sdílena více procesory
3. Shared
4. Invalid

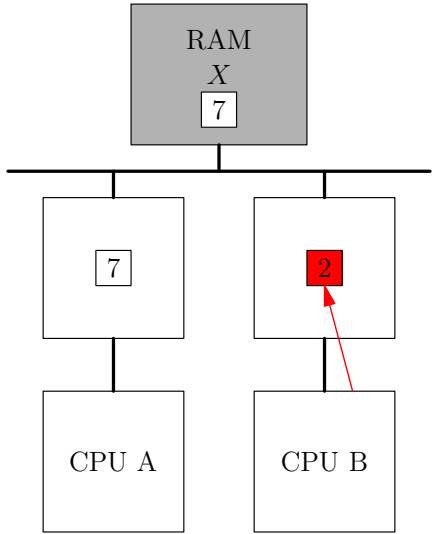
Cache coherence problem

Invalidate protocol - *X* je sdílená proměnná, tj. označená jako SHARED



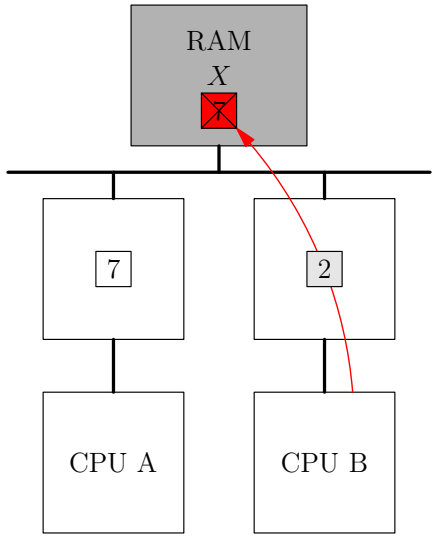
Cache coherence problem

Procesor *B* zapisuje 2 do *X* ve své cache a označuje *X* jako *MODIFIED*, ...



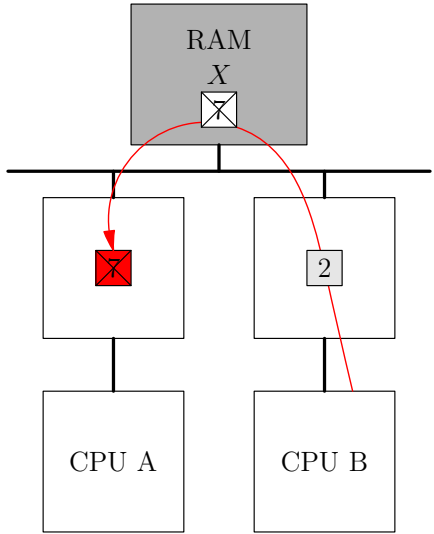
Cache coherence problem

... současně označuje hodnotu X v RAM za neplatnou - *INVALID* ...



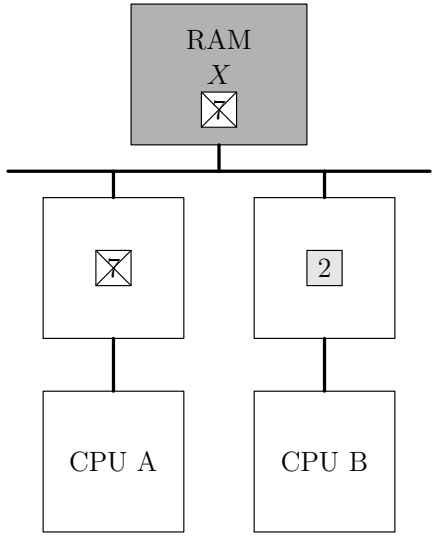
Cache coherence problem

... a stejně tak označí i hodnotu X v cache procesoru B.



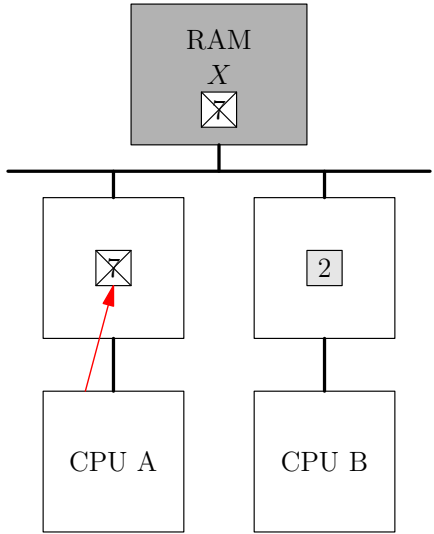
Cache coherence problem

Nakonec je *X* *MODIFIED* v cache CPU *B* a *INVALID* v RAM a cache CPU *A*.



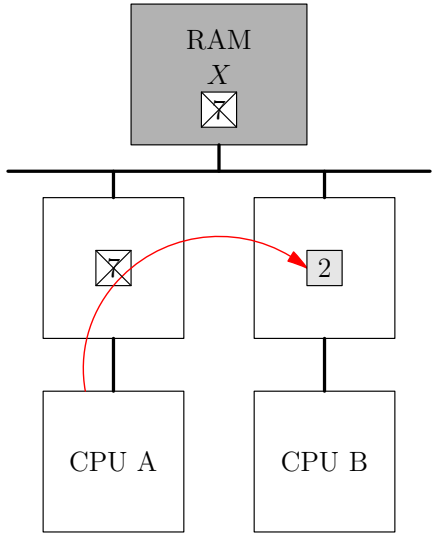
Cache coherence problem

CPU *A* načítá *X* ze své cache a vidí ji označenou jako *INVALID*.



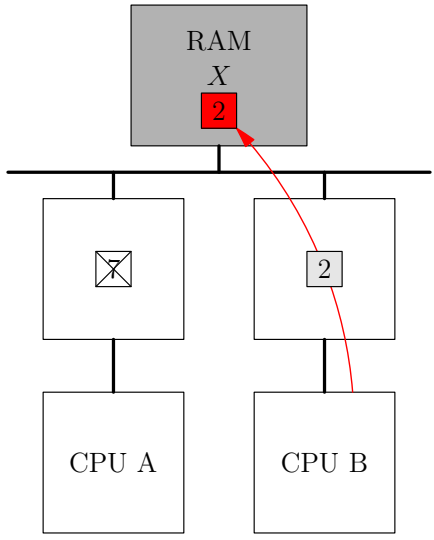
Cache coherence problem

CPU *A* se tedy dotazuje CPU *B*, které má *X* označenou jako *MODIFIED*.



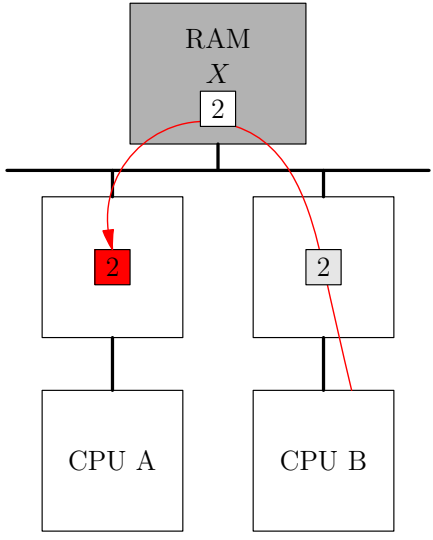
Cache coherence problem

CPU *B* kopíruje hodnotu *X* do RAM ...



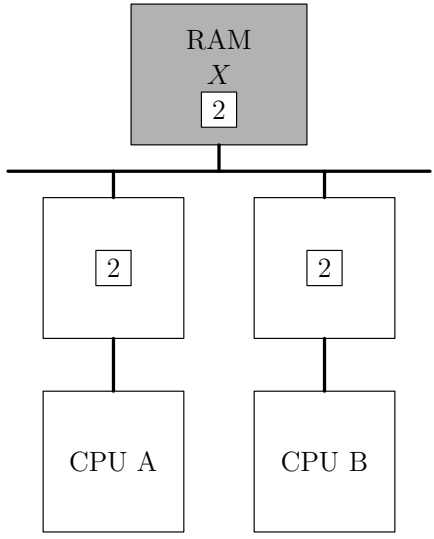
Cache coherence problem

... a do cache CPU *A*.



Cache coherence problem

Proměnná *X* je nakonec všude označena jako sdílená - *SHARED*.



Cache coherence problem

Nevýhody invalidate protokolu - tzv. **false sharing**:

- protokoly update/invalidate se ve skutečnosti vždy vztahují na celou *cache line*
- dva procesory mohou měnit dvě různé proměnné uložené ve stejné **cache line** (přitom každý jednu a tu samou),
  - např. dvě vlákna ukládají mezivýsledky do sdíleného pole
- systém to nepozná a stejně se pokaždé provádí *update*
- režie spojená s *invalidate* protokolem je tu zbytečná
- *update* protokol je v takové situaci lepší

Cache coherence problem

Snoopy cache system

- každý procesor odposlouchává všechnu komunikaci tj. i ostatních CPU
- podle toho pak nastavuje stavy *SHARE, INVALID* a *MODIFIED* u sdílených proměnných

Přístupy do paměti

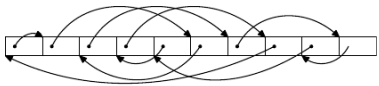
Příklad:

- provedeme stejný test, který jsme dělali pro sekvenční architektury
- nyní ale využijeme více vláken

Pro připomenutí:

```
1  template< int Size >
2  class ArrayElement
3  {
4     ArrayElement* next;
5     long int data[ Size ];
6  }
```

- všechny prvky seznamu se alokují jako velké pole
- následně se propojí buď sekvenčně nebo náhodně



Přístupy do paměti

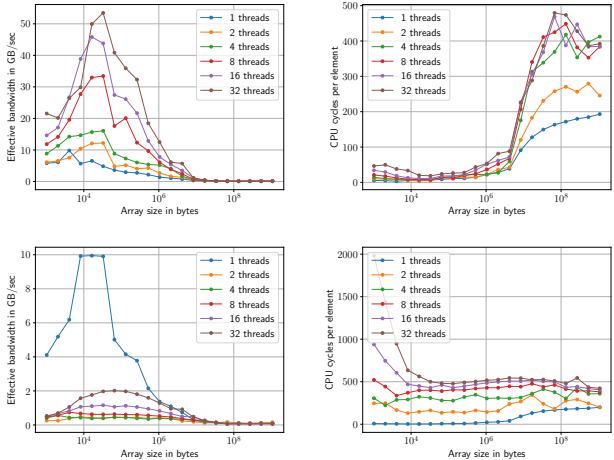
Test budeme provádět na následujících systémech:

- Intel Xeon 6134 (2017)
  - 8 jader
  - L1 cache - 32 kB
  - L2 cache - 1024 kB
  - L3 cache - 24.75 MB
  - paměti DDR4
- AMD Epyc 7281 (2017)
  - 16 jader
  - L1 cache - 32 kB
  - L2 cache - 512 kB
  - L3 cache - 4 MB
  - paměti DDR4 - 170 GB/s na socket
- Apple M1 Pro (2022)
  - 8 výkonných jader
    - L1 cache - 192 kB instrukční a 128 kB datová
    - L2 cache - 12 MB sdílená
  - 2 efektivní jádra
    - L1 cache - 128 kB instrukční a 64 kB datová
    - L2 cache - 4 MB sdílená
  - paměti Low Power DDR5

Náhodné přístupy do paměti

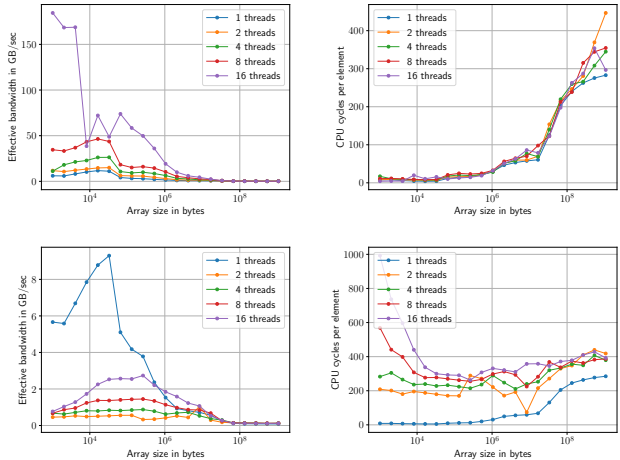
- velikost `Size` nastavíme na 1
- nejprve budeme testovat náhodný přístup do paměti
- při N vláknech vytvoříme N disjunktních stejně dlouhých spojových seznamů
- i-tý seznam začíná na i-tém prvku pole a je zakončen nulovým ukazatelem
- seznamy procházíme opakovaně

Náhodné přístupy do paměti



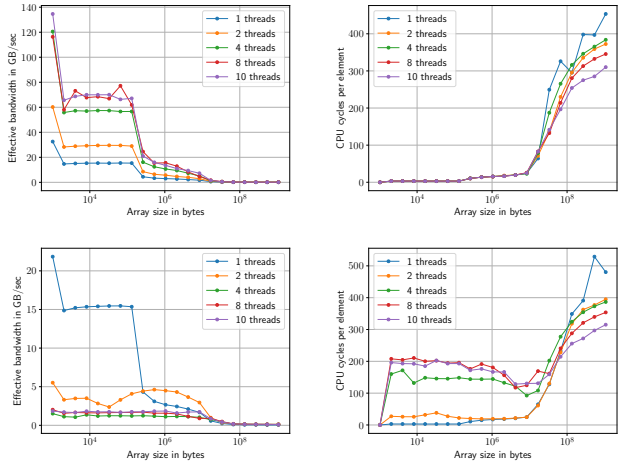
Obrázek: Vícevláknové náhodné čtení (nahoře) a zapisování (dole) na 2xAMD Epyc 7281, tj. až 32 vláken.

Náhodné přístupy do paměti



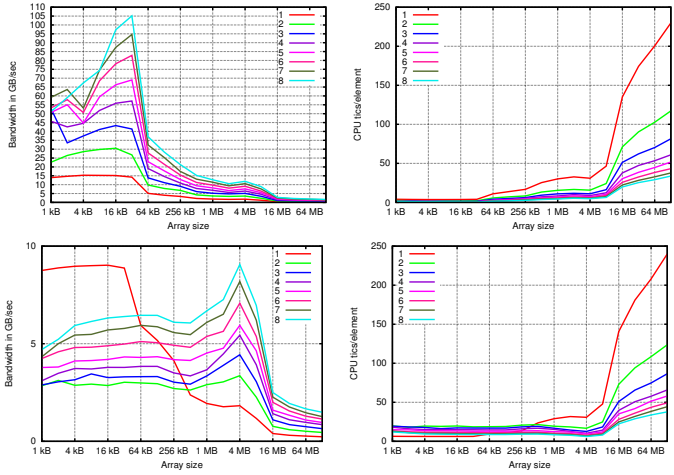
Obrázek: Vícevláknové náhodné čtení (nahoře) a zapisování (dole) na 2x Intel Xeon Gold 6134 8 jader, tj. až 16 vláken.

Náhodné přístupy do paměti



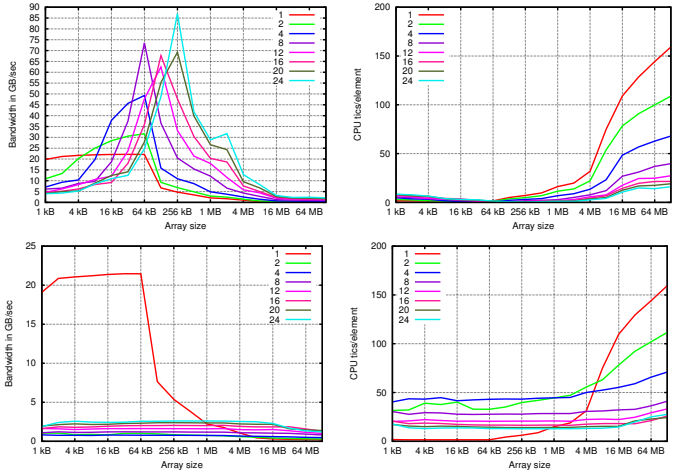
Obrázek: Vícevláknové náhodné čtení (nahoře) a zapisování (dole) na Apple M1 Pro s 10 jádry.

Náhodné přístupy do paměti



Obrázek: Vícevláknové náhodné čtení (nahoře) a zapisování (dole) na Intel i7 3770K s čtyřmi jádry a hyperthreadingem - až 8 vláken.

Náhodné přístupy do paměti

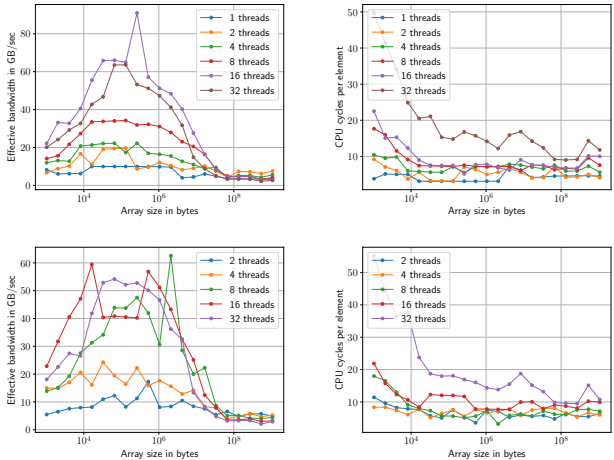


Obrázek: Vícevláknové náhodné čtení (nahoře) a zapisování (dole) na 2x AMD Opteron 6172 s dvanácti jádry - až 24 vláken.

Sekvenční přístupy do paměti

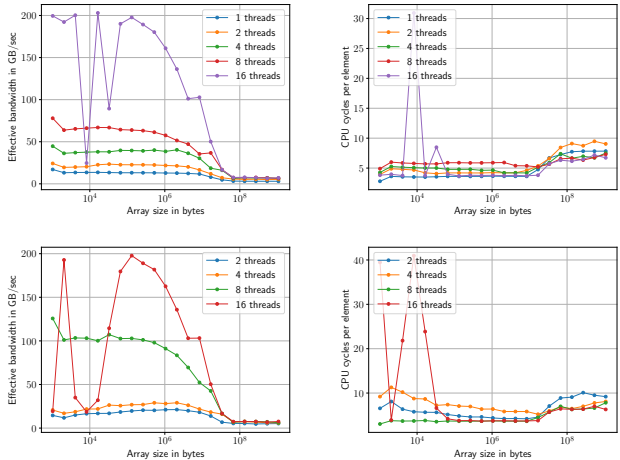
- dále provedeme test sekvenčního přístupu
- porovnáme dva způsoby:
  - vlákna prochází pole na přeskáčku
    - při N vláknech bude i-té vlákno procházet prvky $i + jN$, pro $j = 1, 2, \ldots$
  - každé vlákno prochází svůj blok
    - při N vláknech rozdělíme celé pole na N disjunktních stejně velkých souvislých bloků a každé vlákno prochází jeden blok
- pole procházíme opakovaně

Sekvenční přístupy do paměti



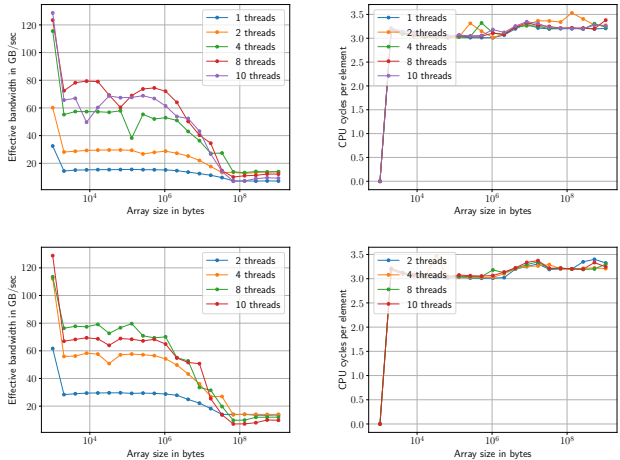
Obrázek: Vícevláknové sekvenční čtení po blocích (nahoře) a na přeskáčku (dole) na 2xAMD Epyc 7281, tj. až 32 vláken.

Sekvenční přístupy do paměti



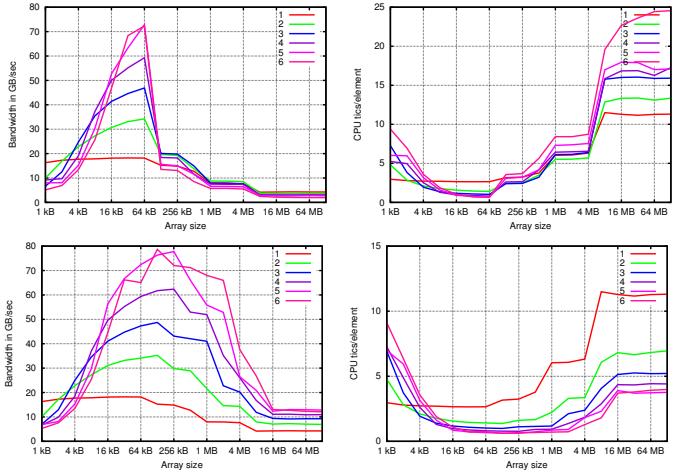
Obrázek: Vícevláknové sekvenční čtení blokově (nahoře) a na přeskáčku (dole) na 2x Intel Xeon Gold 6134, tj. až 16 vláken.

Sekvenční přístupy do paměti



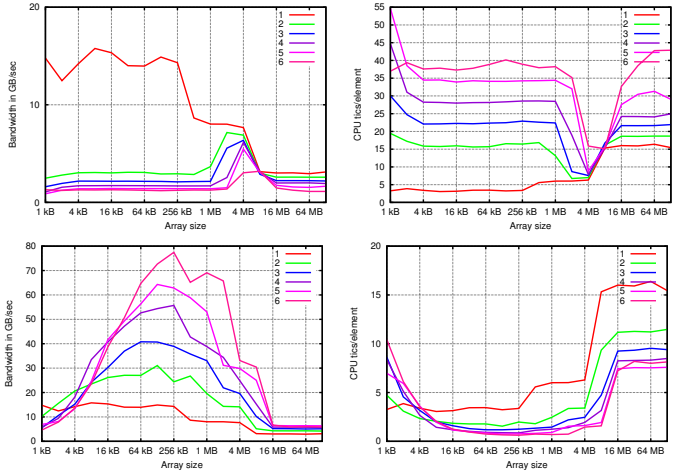
Obrázek: Vícevláknové sekvenční čtení blokově (nahoře) a na přeskáčku (dole) na 2x Intel Xeon Gold 6134, tj. až 16 vláken na Apple M1 Pro s 10 jádry.

Sekvenční přístupy do paměti



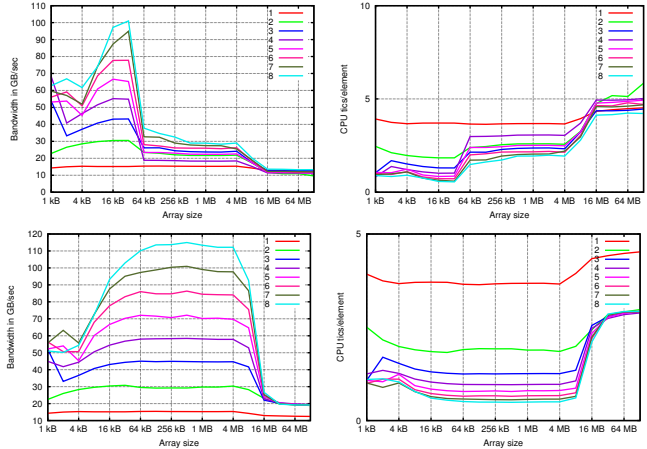
Obrázek: Vícevláknové sekvenční čtení na AMD Phenom 2 X6 1075T - až 6 vláken.
Nahoře je procházení vlákny na přeskáčku dole prochází každé vlákno svůj blok.

Sekvenční přístupy do paměti



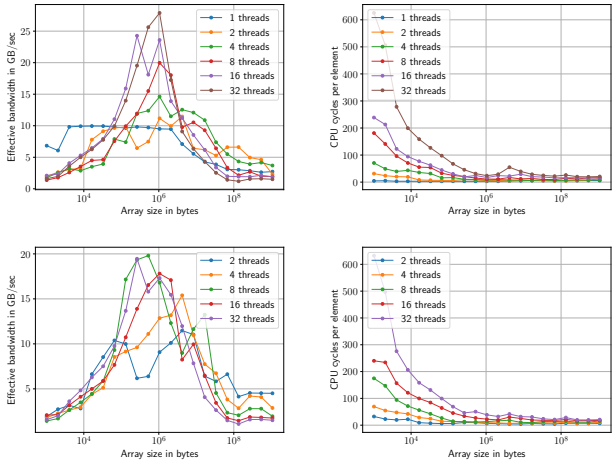
Obrázek: Vícevláknové sekvenční zapisování na AMD Phenom 2 X6 1075T - až 6 vláken.
Nahoře je procházení vlákny na přeskáčku dole prochází každé vlákno svůj blok.

Sekvenční přístupy do paměti



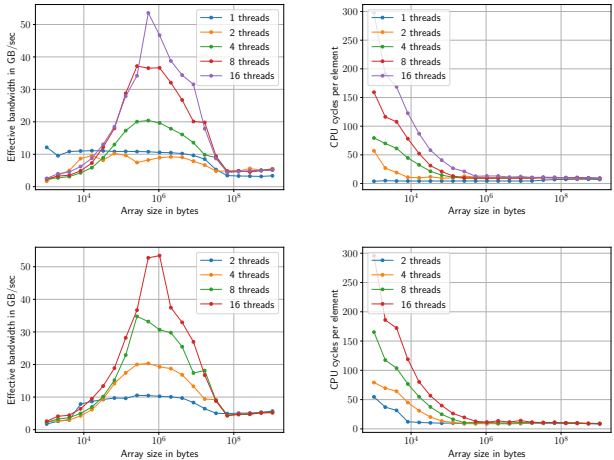
Obrázek: Vícevláknové sekvenční čtení na Intel i7 3770K s čtyřmi jádry a hyperthreadingem - až 8 vláken. Nahoře je procházení vlákny na přeskáčku dole prochází každé vlákno svůj blok.

Sekvenční přístupy do paměti



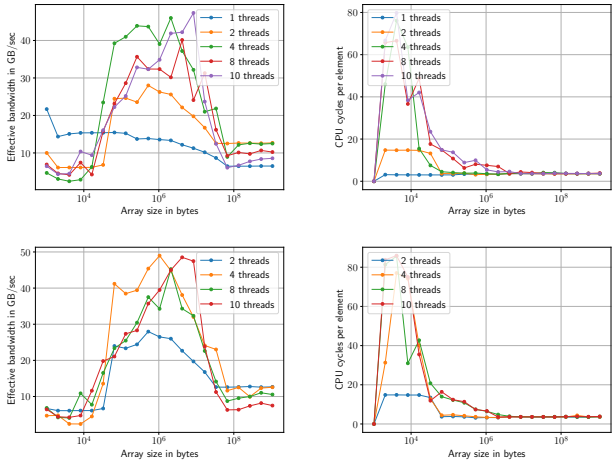
Obrázek: Vícevláknový sekvenční zápis po blocích (nahoře) a na přeskáčku (dole) na 2xAMD Epyc 7281, tj. až 32 vláken.

Sekvenční přístupy do paměti



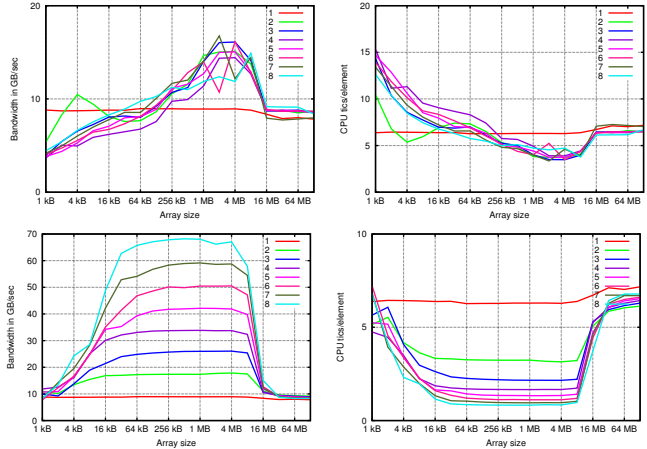
Obrázek: Vícevláknový sekvenční zápis blokově (nahoře) a na přeskáčku (dole) na 2x Intel Xeon Gold 6134, tj. až 16 vláken.

Sekvenční přístupy do paměti



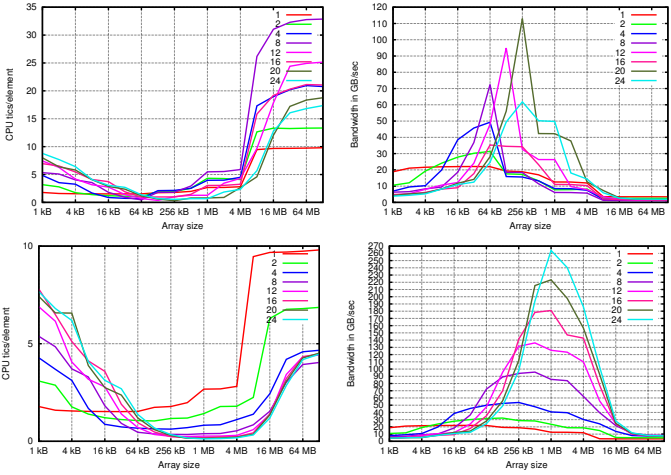
Obrázek: Vícevláknový sekvenční zápis blokově (nahoře) a na přeskáčku (dole) na Apple M1 Pro s 10 jádry.

SMP architektury – sekvenční přístup do paměti



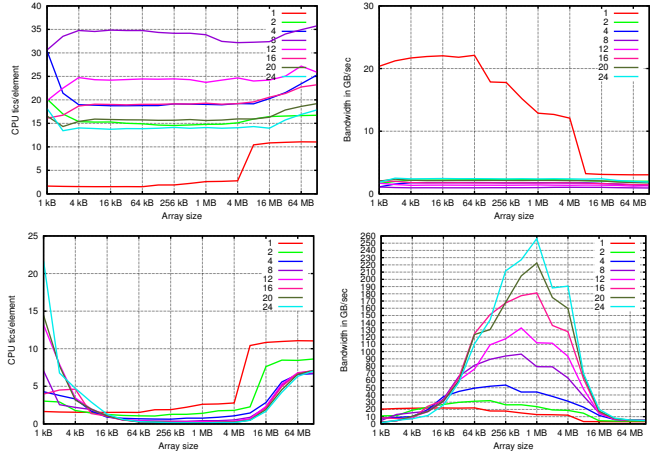
Obrázek: Vícevláknové sekvenční zapisování na Intel i7 3770K s čtyřmi jádry a hyperthreadingem - až 8 vláken. Nahoře je procházení vlákny na přeskáčku dole prochází každé vlákno svůj blok.

SMP architektury – sekvenční přístup do paměti



Obrázek: Vícevláknové sekvenční čtení na 2x AMD Opteron 6172 s dvanácti jádry - až 24 vláken. Nahoře je procházení vlákny na přeskáčku dole prochází každé vlákno svůj blok.

SMP architektury – sekvenční přístup do paměti



Obrázek: Vícevláknové sekvenční zapisování na 2x AMD Opteron 6172 s dvanácti jádry - až 24 vláken. Nahoře je procházení vlákny na přeskáčku dole prochází každé vlákno svůj blok.

Programování systémů se sdílenou pamětí

- opět vidíme, že sekvenční přístup je mnohem efektivnější než náhodný
- je výhodnější, když má každé vlákno svůj vlastní blok paměti, odpadá tak náročné řešení *cache coherence* problému
- zapisování do paměti může být výrazně pomalejší než čtení, obzvlášť, když vlákna přistupují do stejné oblasti v paměti

Programování systémů se sdílenou pamětí

Programy pro architektury se sdílenou pamětí spouštějí několik souběžných vláken.
Standardy:

- POSIX - standard pro manipulaci s vlákny
- OpenMP - standard pro podporu vláken na úrovni překladače
- u `gcc` překladače je pro zapnutí podpory OpenMP potřeba použít přepínač `-fopenmp`
- linkeru je potřeba předat `-lgomp`

OpenMP

OpenMP využívá direktivy preprocesoru.

```
#pragma omp directive [ clause list]
```

Direktiva `parallel` způsobí, že následující blok instrukcí bude zpracován více vlákny.

```
#pragma omp parallel [ clause list]
{
   ...
}
```

Funkce pro identifikaci vláken:

```
omp_get_num_threads() // vrací počet vláken

omp_get_thread_num()} // vrací celočíselný identifikátor vlákna
```

OpenMP - Hello 1

```cpp
#include <omp.h>
#include <stdlib.h>
#include <iostream>

int main( int argc, char** argv )
{
#pragma omp parallel
   {
      std::cout << "Hello from thread number "
                << omp_get_thread_num() << " of "
                << omp_get_num_threads() << " threads."
                << std::endl;
   }
   return EXIT_SUCCESS;
}
```

Výstup:

```
Hello from thread number Hello from thread number 04 of 12 threads.Hello from thread number  of
Hello from thread number 3 of 12 threads.
Hello from thread number 5 of 12 threads.
Hello from thread number 11 of 12 threads.
Hello from thread number 1 of 12 threads.
10 of 12 threads.
Hello from thread number 2 of 12 threads.
Hello from thread number 6 of 12 threads.
12 threads.
Hello from thread number 9 of 12 threads.
Hello from thread number 8 of 12 threads.
Hello from thread number 7 of 12 threads.
```

Kritické bloky

- výstupy na konzoli mohou být chaotické
- je potřeba zařídit, aby současně neprovádělo výpis více vláken
- to lze pomocí kritické sekce

Kritické bloky

Kritické bloky obsahují kód, který může současně provádět jen jedno vlákno.

```
#pragma omp critical [(name)]
```

Bloky pro jedno vlákno

```
#pragma omp single { ... }
```

Tento blok bude zpracován jen jedním (prvním) vláknem. Pokud není uvedeno `nowait`, ostatní vlákna čekají na konci bloku.

```
#pragma omp master { ... }
```

Tento blok bude zpracován jen vláknem s ID = 0, ostatní vlákna nečekají.

OpenMP - Hello s kritickou sekcí

```
1  std::cout << std::endl << "Now with critical section:"
2            << std::endl;
3  #pragma omp parallel
4  {
5  #pragma omp critical
6     std::cout << "Hello from thread number "
7               << omp_get_thread_num() << " of "
8               << omp_get_num_threads() << " threads."
9               << std::endl;
10 }
```

Výstup:

```
1  Now with critical section:
2  Hello from thread number 0 of 12 threads.
3  Hello from thread number 9 of 12 threads.
4  Hello from thread number 10 of 12 threads.
5  Hello from thread number 5 of 12 threads.
6  Hello from thread number 8 of 12 threads.
7  Hello from thread number 2 of 12 threads.
8  Hello from thread number 1 of 12 threads.
9  Hello from thread number 3 of 12 threads.
10 Hello from thread number 4 of 12 threads.
11 Hello from thread number 7 of 12 threads.
12 Hello from thread number 6 of 12 threads.
13 Hello from thread number 11 of 12 threads.
```

OpenMP - Hello se single sekcí

```
1  std::cout << std::endl << "Now with single section:"
2            << std::endl;
3  #pragma omp parallel
4  {
5  #pragma omp single
6     std::cout << "Hello from thread number "
7               << omp_get_thread_num() << " of "
8               << omp_get_num_threads() << " threads."
9               << std::endl;
10 }
```

Výstup:

```
1  Now with single section:
2  Hello from thread number 10 of 12 threads.
```

OpenMP - Hello se master sekcí

```
1  std::cout << std::endl << "Now with master section:"
2                 << std::endl;
3  #pragma omp parallel
4  {
5  #pragma omp master
6     std::cout << "Hello from thread number "
7               << omp_get_thread_num() << " of "
8               << omp_get_num_threads() << " threads."
9               << std::endl;
10 }
```

Výstup:

```
1  Now with master section:
2  Hello from thread number 0 of 12 threads.
```

OpenMP - 2x Hello s kritickou sekcí

```
std::cout << std::endl << "Now two greetings:"
          << std::endl;
#pragma omp parallel
{
#pragma omp critical( print )
  std::cout << "Hello number ONE from thread number "
            << omp_get_thread_num() << " of "
            << omp_get_num_threads() << " threads."
            << std::endl;

#pragma omp critical( print )
  std::cout << "Hello number TWO from thread number "
            << omp_get_thread_num() << " of "
            << omp_get_num_threads() << " threads."
            << std::endl;


}
```

Výstup:

```
Now two greetings:
Hello number ONE from thread number 0 of 12 threads.
Hello number TWO from thread number 0 of 12 threads.
Hello number ONE from thread number 3 of 12 threads.
Hello number TWO from thread number 3 of 12 threads.
Hello number ONE from thread number 9 of 12 threads.
Hello number ONE from thread number 11 of 12 threads.
Hello number TWO from thread number 11 of 12 threads.
Hello number TWO from thread number 9 of 12 threads.
Hello number ONE from thread number 4 of 12 threads.
Hello number TWO from thread number 4 of 12 threads.
Hello number ONE from thread number 7 of 12 threads.
Hello number TWO from thread number 7 of 12 threads.
Hello number ONE from thread number 5 of 12 threads.
...
```

OpenMP - 2x Hello s kritickou sekcí a bariérou

```
std::cout << std::endl << "Now two greetings:"
          << std::endl;
#pragma omp parallel
{
#pragma omp critical( print )
  std::cout << "Hello number ONE from thread number "
            << omp_get_thread_num() << " of "
            << omp_get_num_threads() << " threads."
            << std::endl;
#pragma omp barrier
#pragma omp critical( print )
  std::cout << "Hello number TWO from thread number "
            << omp_get_thread_num() << " of "
            << omp_get_num_threads() << " threads."
            << std::endl;

}
```

Výstup:

```
... and now two greetings with barrier:
Hello number ONE from thread number 11 of 12 threads.
Hello number ONE from thread number 3 of 12 threads.
Hello number ONE from thread number 8 of 12 threads.
Hello number ONE from thread number 6 of 12 threads.
Hello number ONE from thread number 4 of 12 threads.
Hello number ONE from thread number 7 of 12 threads.
Hello number ONE from thread number 0 of 12 threads.
Hello number ONE from thread number 5 of 12 threads.
Hello number ONE from thread number 9 of 12 threads.
Hello number ONE from thread number 10 of 12 threads.
Hello number ONE from thread number 1 of 12 threads.
Hello number ONE from thread number 2 of 12 threads.
Hello number TWO from thread number 8 of 12 threads.
Hello number TWO from thread number 3 of 12 threads.
Hello number TWO from thread number 9 of 12 threads.
Hello number TWO from thread number 0 of 12 threads.
...
```

OpenMP - bariéra

- do kódu je potřeba vložit bariéru
- to je bod v programu, kde na sebe všechna vlánka počkají
- až když každé z vláken dosáhne bariery, mohou vlákna pokračovat dále

OpenMP - proměnné

Přístup více CPU do paměti

- rozlišujeme dva typy proměnných
  - **soukromé** (private) - jsou přístupné jen jednomu procesoru
  - **sdílené** (shared) - může k nim přistupovat více procesorů

Ošetření sdílených proměnných

- multiprocesor se sdílenou pamětí neumožňuje, aby současně přistupovalo více procesorů na stejné místo v paměti
  - pokud se tak stane výsledek je nepředvídatelný
  - většinou je nutné se této situaci vyhnout dobře napsaným kódem

OpenMP - proměnné

Pomocí `[` *clause list* `]` lze udat:

- podmínku (pouze jednu) paralelizace: `if( ... )`
- počet vláken: `num_threads(` *integer expression* `)`
- zacházení s daty
  - `private(` *variable list* `)`
    Určuje lokální proměnné = každé vlákno má svou vlastní kopii.
  - `firstprivate(` *variable list* `)`
    Stejně jako `private`, ale u všech kopii se nastaví hodnota, kterou měla proměnná před rozvětvením běhu programu na vlákna.
  - `shared(` *variable list* `)`
    Tyto proměnné budou sdílené mezi vlákny.
  - `reduction(` *operator*: *variable list* `)`
    Dané proměnné budou mít lokální kopie a nakonec se provede redukce pomocí asociativní operace: +,*,&, |,&&, ||.

Příklady

```
#pragma omp parallel if( is_parallel == true )
         num_threads(8) private(a)
            firstprivate(b)
{

...

}
```

```
#pragma omp parallel if( size > 1000 )
        num_threads( MIN( size/1000+1,8) )
           reduction(+:sum)
{

...

```

OpenMP - proměnné

```
1  int main( int argc, char** argv )
2  {
3     int shared_int( 1 ), private_int( 1 ), firstprivate_int( 1 );
4  #pragma omp parallel shared( shared_int ), private( private_int ),
5                       firstprivate( firstprivate_int )
6     {
7  #pragma omp critical
8        std::cout << "Variables of thread "
9                  << omp_get_thread_num() << " are"
10                 << " shared_int = " << shared_int
11                 << " private_int = " << private_int
12                 << " firstprivate_int = " << firstprivate_int
13                 << std::endl;
14 #pragma omp barrier
15       shared_int ++;
16       private_int ++;
17       firstprivate_int ++;
18 #pragma omp barrier
19 #pragma omp single
20       std::cout << "======================================" << std::endl;
21 #pragma omp critical
22       std::cout << "Variables of thread "
23                 << omp_get_thread_num() << " are"
24                 << " shared_int = " << shared_int
25                 << " private_int = " << private_int
26                 << " firstprivate_int = " << firstprivate_int
27                 << std::endl;
28    }
29    return EXIT_SUCCESS;
30 }
```

OpenMP - proměnné

Výstup:

```
Variables of thread 0 are shared_int = 1 private_int = 0 firstprivate_int = 1
Variables of thread 4 are shared_int = 1 private_int = 0 firstprivate_int = 1
Variables of thread 6 are shared_int = 1 private_int = 0 firstprivate_int = 1
Variables of thread 2 are shared_int = 1 private_int = 0 firstprivate_int = 1
Variables of thread 7 are shared_int = 1 private_int = 0 firstprivate_int = 1
Variables of thread 3 are shared_int = 1 private_int = 0 firstprivate_int = 1
Variables of thread 8 are shared_int = 1 private_int = 0 firstprivate_int = 1
Variables of thread 10 are shared_int = 1 private_int = 0 firstprivate_int = 1
Variables of thread 9 are shared_int = 1 private_int = 0 firstprivate_int = 1
Variables of thread 1 are shared_int = 1 private_int = 0 firstprivate_int = 1
Variables of thread 11 are shared_int = 1 private_int = 0 firstprivate_int = 1
Variables of thread 5 are shared_int = 1 private_int = 0 firstprivate_int = 1
=====================================
Variables of thread 0 are shared_int = 6 private_int = 1 firstprivate_int = 2
Variables of thread 8 are shared_int = 6 private_int = 1 firstprivate_int = 2
Variables of thread 9 are shared_int = 6 private_int = 1 firstprivate_int = 2
Variables of thread 2 are shared_int = 6 private_int = 1 firstprivate_int = 2
Variables of thread 1 are shared_int = 6 private_int = 1 firstprivate_int = 2
Variables of thread 10 are shared_int = 6 private_int = 1 firstprivate_int = 2
Variables of thread 11 are shared_int = 6 private_int = 1 firstprivate_int = 2
Variables of thread 3 are shared_int = 6 private_int = 1 firstprivate_int = 2
Variables of thread 5 are shared_int = 6 private_int = 1 firstprivate_int = 2
Variables of thread 7 are shared_int = 6 private_int = 1 firstprivate_int = 2
Variables of thread 4 are shared_int = 6 private_int = 1 firstprivate_int = 2
Variables of thread 6 are shared_int = 6 private_int = 1 firstprivate_int = 2
```

Kde je chyba?

OpenMP - proměnné

```
int main( int argc, char** argv )
{
   int shared_int( 1 ), private_int( 1 ), firstprivate_int( 1 );
#pragma omp parallel shared( shared_int ), private( private_int ),
                     firstprivate( firstprivate_int )
   {
#pragma omp critical
      std::cout << "Variables of thread "
                << omp_get_thread_num() << " are"
                << " shared_int = " << shared_int
                << " private_int = " << private_int
                << " firstprivate_int = " << firstprivate_int
                << std::endl;
#pragma omp barrier
#pragma omp critical
      shared_int ++;
      private_int ++;
      firstprivate_int ++;
#pragma omp barrier
#pragma omp single
      std::cout << "====================================" << std::endl;
#pragma omp critical
      std::cout << "Variables of thread "
                << omp_get_thread_num() << " are"
                << " shared_int = " << shared_int
                << " private_int = " << private_int
                << " firstprivate_int = " << firstprivate_int
                << std::endl;
   }
   return EXIT_SUCCESS;
}
```

OpenMP - proměnné

```
int main( int argc, char** argv )
{
   int shared_int( 1 ), private_int( 1 ), firstprivate_int( 1 );
#pragma omp parallel shared( shared_int ), private( private_int ),
                     firstprivate( firstprivate_int )
   {
#pragma omp critical
      std::cout << "Variables of thread "
                << omp_get_thread_num() << " are"
                << " shared_int = " << shared_int
                << " private_int = " << private_int
                << " firstprivate_int = " << firstprivate_int
                << std::endl;
#pragma omp barrier
#pragma omp atomic
      shared_int ++;
      private_int ++;
      firstprivate_int ++;
#pragma omp barrier
#pragma omp single
      std::cout << "====================================" << std::endl;
#pragma omp critical
      std::cout << "Variables of thread "
                << omp_get_thread_num() << " are"
                << " shared_int = " << shared_int
                << " private_int = " << private_int
                << " firstprivate_int = " << firstprivate_int
                << std::endl;
   }
   return EXIT_SUCCESS;
}
```

Atomické instrukce

Atomické instrukce

- používají se pro řízení přístupu ke sdíleným proměnným
- atomická instrukce je provedena vždy kompletně a bez přerušení
- mezi základní atomické instrukce patří
  - compare-and-swap - CAS
  - test-and-set - TS
  - read-modify-write

Atomické instrukce - CAS

Instrukce CAS by mohla být implementována takto:

```
1  bool CAS( int* pointer, int old, int new )
2  {
3      if( *pointer != old )
4          return false;
5      *pointer = new;
6      return true;
7  }
```

Její zpracování ale nikdo nemůže přerušit.

Zámek pomocí CAS

Příklad implementace **zámku** pomocí CAS:

```
bool locked( false )
while( ! CAS( &locked, false, true ) )
{
    // Kod prouze pro jedno vlakno
    ....
    locked = false;
}
```

Příklad implementace **atomického přičítání** pomocí CAS:

```
void atomicAdd( int& value, int a )
{
    bool done( false );
    while( ! done )
    {
        int aux = value;
        done = CAS( value, aux, aux + a );
    }
}
```

OpenMP a atomické operace

```
#pragma omp atomic [update/read/write/compare/capture]}
```

- `update` je defaultní hodnota

direktiva	výraz	blok

`update`	`x++;`
	`x--;`
	`++x;`
	`--x;`
	`x binop = expr;`
	`x = x binop expr;`
	`x = expr binop x;`	

`read`	`v = x;`	
`write`	`x = expr;`	

`capture`	`v = x++;`
	`v = x --;`
	`v = ++x;`
	`v = --x;`
	`v = x binop = expr;`
	`v = x = x binop expr;`
	`v = x = expr binop x;`
		`{v = x; x binop expr;}`
		`{v = x; x OP;}`
		`{v = x; OP x;}`
		`{x binop = expr; v = x;}`
		`{xOP; v = x;}`
		`{OPx; v = x;}`
		`{v = x; x= x binop expr;}`
		`{x = x binop expr; v = x;}`
		`{v = x;x = expr binop v;}`
		`{x = expr binop x; v = x;}`
		`{v = x; x = expr;}`

Kde `x`, `v` jsou *lvalue* skalární hodnoty, `expr` je skalární výraz nezávisející na `x`, `binop` je `+`, `*`, `-`, `/`, `&`, `«`, `»`, `^`, `|` a `OP` je `++`, `--`.

OpenMP - určení souběžných úloh

Po spuštění více vláken je nutné říci, co mají jednotlivá vlákna provádět.

- všechna vlákna provádějí stejnou úlohu = dělí se o `for` cyklus
- každé vlákno provádí jinou úlohu = zpracovávají sekce (`sections`) různého kódu

OpenMP - paralelizace `for` cyklů

```
1 #pragma omp for[clause list]
```

Klauzule pro ošetření proměnných:

- `private`
- `firstprivate`
- `reduction`
- `lastprivate` = hodnota proměnné je nastavena v posledním průběhu `for` cyklu

OpenMP - paralelizace `for` cyklů

Klausule pro rozdělení iterací mezi vlákny - `schedule`

`schedule(` *schedulling_class[,parameter]*`)`

Třídy:

- `static`
- `dynamic`
- `guided`
- `runtime`

OpenMP - statické přidělování iterací

```
schedule( static[, chunk-size] )
```

Každé vlákno postupně dostává stejný počet iterací daný pomocí `chunk-size`.

Není-li `chunk-size` uvedeno, jsou všechny iterace rozděleny na `n` stejných částí, kde `n` je počet vláken.

Příklad: 128 iterací, 4 vlákna
`schedule( static )` = 4 $\times$ 32 iterací
`schedule( static, 16 )` = 8 $\times$ 16 iterací

OpenMP - dynamické přidělování iterací

```
schedule( dynamic[, chunk-size] )
```

Funguje podobně jako dynamické přidělovaní iterací. Nové iterace jsou ale přidány vláknu, které skončí svou práci jako první. Některá vlákna tak mohou provést více iterací, než ostatní.

OpenMP - řízené přidělování iterací

`schedule( guided[, chunk-size] )`

Příklad: 100 iterací rozdělených po 5 $\Rightarrow$ 20 kousků pro 16 vláken.

`guided` s každým přidělením nových iterací exponenciálně zmenšuje *chunk size*.
`chunk-size` udává dolní mez pro počet přidělených iterací.

OpenMP - přidělování iterací určené za chodu programu

`schedule(runtime)`

Podle systémové proměnné `OMP_SCHEDULE` se určí, zda se má použít `static`, `dynamic` nebo `guided`.

Vhodné při vývoji programu pro zjištění nejvhodnější volby.

OpenMP - synchronizace mezi jednotlivými `for` cykly

Standardně se nezačíná nový cyklus, dokud všechna vlákna neskončila práci na předchozím cyklu - bariéra mezi cykly. Pokud to není nutné, lze použít klauzuli `nowait`.

```
#pragma omp parallel
{
   #pragma omp for nowait
      for( i = 0; i < nmax; i ++ ) { ... }

   #pragma omp for
      for( i = 0; i < mmax; i ++ ) { ... }
}
```

Lze také psát:

```
#pragma omp parallel for shared(n)
```

OpenMP - zpracování různých úloh každým vláknem

Provádí se pomocí direktivy `omp sections`:

```
#pragma omp parallel
{
   #pragma omp sections
   {
      #pragma omp section
      { TaskA(); }

      #pragma omp section
      { TaskB(); }
   }
}
```

Lze také psát:

```
#pragma omp parallel sections
```

OpenMP - vložení direktivy `parallel`

Musí být nastavena systémová proměnná
`OMP_NESTED = TRUE`.

```
1  #pragma omp parallel for ...
2  for( i = 0; i < N; i ++ )
3          #pragma omp parallel for ...
4          for( j = 0; j < N; j ++ )
5                  #pargma omp parallel for ...
6                  for( k = 0; k < N; k ++ )
7                          #pragma omp parallel for ...
8  }
```

OpenMP - kritické bloky

Příklad: Částečné úlohy pro jednotlivá vlákna lze distribuovat pomocí centrální struktury (fronty). Přístup k ní pak může mít v daný okamžik jen jedno vlákno.

OpenMP - kritické bloky - příklad

```
#pragma omp parallel sections
{
        #pragma omp section
        { /* producer thread */
                task = producer_task();
                #pragma omp critical (task_queu)
                { insert_into_queu( task ); }

        }
        #pragma omp parallel section
        { /* consumer thread */
                #pragma omp critical (task_queu)
                { task = extract_from_queu(); }

        }
}
```

Funkce knihovny OpenMP

Je nutné použít hlavičkový soubor

```
1  #include <omp.h>
2
3  void omp_set_num_threads( int num_threads);
4
5  int omp_get_num_threads();
6
7  int omp_get_thread_num();
8
9  int omp_get_num_procs();
10
11 int omp_in_paralle();
```

Systémové proměnné

- `OMP_NUM_THREADS`
  - `setenv OMP_NUM_THREADS=8`
- `OMP_DYNAMIC`
  - umožňuje použití funkci `omp_set_num_threads` nebo klauzuli `num_threads`
  - `setenv OMP_DYNAMIC="TRUE"`
- `OMP_NESTED`
- `OMP_SCHEDULE`
  - `setenv OMP_SCHEDULE="static,4"`
  - `setenv OMP_SCHEDULE="dynamic"`
  - `setenv OMP_SCHEDULE="guided"`

Ošetření dat v OpenMP

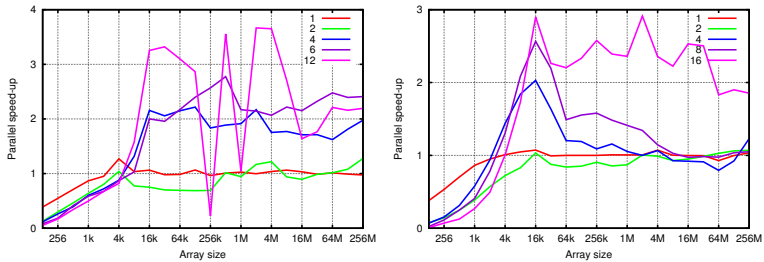
- všechny proměnné, které používá jen jedno vlákno, by měly být označené jako `private`
- při častém čtení proměnné nastavené již dříve v programu je vhodné označit ji jako `firstprivate`
- přístup všech vláken ke stejným datům je vhodné provádět pomocí lokálních proměnných (vzhledem k vláknu) a nakonec použít redukci
- přistupují-li vlákna k různým částem velkého bloku dat, je dobré je předem explicitně rozdělit
- zbývající typ proměnných již musí být zřejmě sdílen

OpenMP - skalární součin

Ukázka paralelizace skalárního součinu:

```
1  double scalarProductSequential( const double* v1,
2                                  const double* v2,
3                                  const int size )
4  {
5     double res( 0.0 );
6     for( int i = 0; i < size; i++ )
7        res += v1[ i ] * v2[ i ];
8     return res;
9  }
10
11 double scalarProductParallel( const double* v1,
12                                const double* v2,
13                                const int size )
14 {
15    double res( 0.0 );
16    #pragma omp parallel for reduction(+:res),     \
17                             schedule( static ),   \
18                             firstprivate( v1, v2 )
19    for( int i = 0; i < size; i++ )
20       res += v1[ i ] * v2[ i ];
21    return res;
22 }
```

OpenMP - skalární součin



Obrázek: Paralelní urychlení získané při výpočtu skalárního součinu na Intel i7 5820K (2x6 vlaken) a 2x Intel Xeon E5 2630 (2x2x8 vláken).

OpenMP - max

Ukážeme si několik způsobů, jak vypočítat maximum s pomocí OpenMP

1. Sekvenčně

```
1  int maxSequential(const int *v, const int size)
2  {
3     int res(std::numeric_limits<int>::lowest());
4     for (int i = 0; i < size; i++)
5        res = std::max(res, v[i]);
6     return res;
7  }
```

Link pro kompilaci.

OpenMP - max

2. OpenMP se špatným přístupem do sdílené proměnné

```
1  int maxOmpWrong(const int *v, const int size)
2  {
3     int res(std::numeric_limits<int>::lowest());
4  #pragma omp parallel for shared(res), schedule(static), firstprivate(v)
5     for (int i = 0; i < size; i++)
6     {
7        res = std::max(res, v[i]);
8     }
9     return res;
10 }
```

Link pro kompilaci.

Přístup do proměnné `res` by měl být ošetřen atomicky.

OpenMP - max

3. OpenMP s atomickým přístupem do sdílené proměnné

```
1  int maxOmpAtomic(const int *v, const int size)
2  {
3     int res(std::numeric_limits<int>::lowest());
4  #pragma omp parallel for shared(res), schedule(static), firstprivate(v)
5     for (int i = 0; i < size; i++)
6     {
7        const int aux = v[i];
8  #pragma omp atomic compare
9        if (res < aux) { res = aux; }
10    }
11    return res;
12 }
```

Link pro kompilaci.

OpenMP - max

4. OpenMP s optimalizovaným atomickým přístupem do sdílené proměnné

```
1  int maxOmpAtomicOpt(const int *v, const int size)
2  {
3     int res(std::numeric_limits<int>::lowest());
4  #pragma omp parallel for shared(res), schedule(static), firstprivate(v)
5     for (int i = 0; i < size; i++)
6     {
7        const int aux = v[i];
8        if (res < aux)
9        {
10 #pragma omp atomic compare
11          if (res < aux)
12          {
13             res = aux;
14          }
15       }
16    }
17    return res;
18 }
```

OpenMP - max

5. OpenMP **s** false sharing

```
1  int maxOmpFalseSharing(const int *v, const int size)
2  {
3     int res(std::numeric_limits<int>::lowest());
4     const int threads_count = omp_get_max_threads();
5     int *aux = new int[threads_count];
6     for (int i = 0; i < threads_count; i++)
7        aux[i] = res;
8  #pragma omp parallel shared(res), firstprivate(v)
9     {
10       const int thread_id = omp_get_thread_num();
11 #pragma omp for schedule(static)
12       for (int i = 0; i < size; i++)
13          aux[thread_id] = std::max(aux[thread_id], v[i]);
14    }
15    for (int i = 0; i < threads_count; i++)
16       res = std::max(res, aux[i]);
17    return res;
18 }
```

Link pro kompilaci.

OpenMP - max

6. OpenMP **bez** false sharing

```
1  int maxOmpNoFalseSharing(const int *v, const int size)
2  {
3     int res(std::numeric_limits<int>::lowest());
4     const int cache_line = 64 / sizeof(int);
5     const int threads_count = omp_get_max_threads();
6     int *aux = new int[threads_count * cache_line];
7     for (int i = 0; i < threads_count; i++)
8        aux[i * cache_line] = res;
9  #pragma omp parallel shared(res), firstprivate(v)
10    {
11       const int idx = omp_get_thread_num() * cache_line;
12 #pragma omp for schedule(static)
13       for (int i = 0; i < size; i++)
14          aux[idx] = std::max(aux[idx], v[i]);
15    }
16    for (int i = 0; i < threads_count; i++)
17       res = std::max(res, aux[i * cache_line]);
18    return res;
19 }
```

Link pro kompilaci.

OpenMP - max

7. OpenMP redukce

```
1  int maxOmpReduction(const int *v, const int size)
2  {
3  #pragma omp declare reduction(maxVal:int :                          \
4          omp_out = omp_in < omp_out ? omp_out : omp_in),             \
5          initializer(omp_priv = std::numeric_limits<int>::lowest())
6     int res;
7  #pragma omp parallel for reduction(maxVal : res), schedule(static), \
8                           firstprivate(v)
9     for (int i = 0; i < size; i++)
10    {
11       res = std::max(res, v[i]);
12    }
13    return res;
14 }
```

Link pro kompilaci.

OpenMP - výsledky

Výsledky benchmarku (**počet taktů** na element pole) na
AMD Ryzen 9 5950X 16 jader

N	Seq.	Wrong	Atomic	Atomic Opt.	False Sharing	No False Sharing	Reduction
				2 vlákna			
128	0.81	27.58	43.33	37.56	30.68	32.23	30.85
256	0.81	13.94	33.52	20.80	15.69	16.52	15.40
512	0.80	7.31	27.48	13.02	7.74	8.38	7.74
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
262144	0.73	0.50	23.89	5.00	0.48	0.38	0.38
				4 vlákna			
128	0.89	32.62	68.61	33.14	36.64	38.94	40.86
256	0.81	16.82	75.19	17.56	19.23	19.96	21.33
512	0.78	8.80	81.45	10.21	10.08	9.93	10.60
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
262144	0.73	0.32	78.33	3.44	0.29	0.20	0.20
				16 vláken			
128	0.89	66.46	144.84	69.29	61.98	73.96	160.07
256	0.86	29.20	109.96	33.87	30.94	36.58	80.21
512	0.79	14.80	95.70	17.57	15.70	18.20	39.53
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
262144	0.78	0.15	83.06	0.96	0.12	0.11	0.13

OpenMP - výsledky

Výsledky benchmarku (**počet taktů** na element pole) na
Intel Core i7-10700 8 jader

N	Seq.	Wrong	Atomic	Atomic Opt.	False Sharing	No False Sharing	Reduction
				2 vlákna			
128	0.65	9.69	96.38	25.48	9.92	10.58	10.68
256	0.67	5.25	92.47	19.18	5.20	5.48	5.57
512	0.64	2.87	90.21	16.61	2.80	2.94	3.04
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
262144	0.65	0.41	88.02	14.81	0.41	0.41	0.31
				4 vlákna			
128	0.66	18.87	108.02	29.16	15.10	16.32	20.12
256	0.67	7.42	99.38	18.74	7.59	8.20	10.04
512	0.66	3.82	95.80	14.96	3.93	4.19	5.08
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
262144	0.64	0.21	90.99	11.94	0.21	0.21	0.16
				8 vláken			
128	0.73	23.08	117.20	32.71	21.45	24.11	37.46
256	0.61	10.54	107.44	18.32	10.57	11.72	18.63
512	0.62	5.22	102.47	10.99	5.19	5.84	9.27
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
262144	0.69	0.12	99.73	3.43	0.11	0.11	0.10

OpenMP - výsledky

Pozor!!! OpenMP zpomaluje i při běhu pouze s jedním vláknem:

N	Seq.	Wrong	Atomic	Atomic Opt.	False Sharing	No False Sharing	Reduction
1 vlákno - AMD Ryzen 9 5950X							
128	0.50	6.74	13.34	13.88	8.45	8.72	6.07
256	0.64	3.51	10.23	10.72	4.44	4.73	3.75
512	0.68	2.12	8.71	9.34	2.60	2.72	2.25
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
262144	0.73	0.74	7.58	8.02	0.74	0.75	0.74

N	Seq.	Wrong	Atomic	Atomic Opt.	False Sharing	No False Sharing	Reduction
1 vlákno - Intel Core i7-10700							
128	0.73	23.08	117.20	32.71	21.45	24.11	37.46
128	0.54	4.63	23.46	23.50	5.86	6.10	4.83
256	0.60	2.83	21.94	21.77	3.23	3.45	2.69
512	0.62	1.79	20.82	20.82	2.06	2.13	1.66
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
262144	0.64	0.82	19.99	19.84	0.81	0.80	0.63

OpenMP - shrnutí

- Zdá se, že false sharing již není tak výrazný problém.
- Měření naopak ukazuje, že je potřeba být opatrný s atomickými instrukcemi.