

# Úvod do B4B36PDV

Organizace předmětu a seznámení se s paralelizací

---

B4B36PDV – Paralelní a distribuované výpočty

- Čím se budeme zabývat?
- Hodnocení předmětu
- Úvod do paralelního hardwaru a softwaru

## Organizace předmětu

---

**Přednášející:** Jakub Mareček      Michal Jakob

**Cvičící:**

**Paralelní část:** Petr Macejko, David Fiedler, Michal Slouka, Matěj Kafka

**Distribuovaná část:** Petr Macejko, Jan Mrkos, David Milec

Důležité odkazy:

<https://cw.fel.cvut.cz/wiki/courses/b4b36pdv>

<https://cw.felk.cvut.cz/forum/>

<https://cw.felk.cvut.cz/brute/>

Paralelní      a      Distribuované výpočty

## Paralelní a Distribuované výpočty

### Paralelní výpočty

- **Jeden** výpočet provádí současně **více** vláken
- Vlákná typicky sdílí paměť a výpočetní prostředky
- Cíl: Zrychlit výpočet úlohy
- (7 týdnů)

## Paralelní a Distribuované výpočty

### Paralelní výpočty

- **Jeden** výpočet provádí současně **více** vláken
- Vlákna typicky sdílí paměť a výpočetní prostředky
- Cíl: Zrychlit výpočet úlohy
- (7 týdnů)

### Distribuované výpočty

- Výpočet provádí současně více oddělených výpočetních uzlů (často i geograficky)
- Cíle:
  - Zrychlit výpočet
  - Robustnost výpočtu
- (6 týdnů)

## Paralelní a Distribuované výpočty

### Paralelní výpočty

- **Jeden** výpočet provádí současně **více** vláken
- Vlákna typicky sdílí paměť a výpočetní prostředky
- Cíl: Zrychlit výpočet úlohy
- (7 týdnů)

### Distribuované výpočty

- Výpočet provádí současně více oddělených výpočetních uzlů (často i geograficky)
- Cíle:
  - Zrychlit výpočet
  - **Robustnost výpočtu**
- (6 týdnů)



## Paralelní výpočty

- |                                |         |
|--------------------------------|---------|
| • 5 malých programovacích úloh | 10 bodů |
| • Semestrální práce            | 12 bodů |

## Distribuované výpočty

- |                     |         |
|---------------------|---------|
| • 2 malé úlohy      | 4 body  |
| • Semestrální práce | 14 bodů |

## Praktická zkouška

- |  |         |
|--|---------|
| • Praktická část zkoušky (min. 10b)<br>(Vyřešení zadaného problému za použití paralelizace.) | 20 bodů |
| • Teoretická část zkoušky (min. 20b)   | 40 bodů |

V semestru musíte získat **alespoň 20 bodů**

Celkem: 100 bodů

Vyžadujeme **samostatnou** práci na všech úlohách.

 **Plagiáty jsou zakázané.** Nepřidělávejte prosím starosti nám, ani sobě.

Docházka na cvičení není povinná.

To ale neznamená, že byste na cvičení neměli chodit...

- Budeme probírat látku, která se Vám bude hodit u úkolů a u zkoušky.
- Dostanete prostor pro práci na semestrálních pracích.
- Konzultace budou probíhat **primárně** na cvičeních.
- Ušetříme Vám čas a nervy (nebo v to alespoň doufáme ;-)

---

**⚠** Pokud se na cvičení rozhodnete nechodit, budeme předpokládat, že probírané látce dokonale rozumíte. Případné konzultace v žádném případě nenahrazují cvičení!

- Programování v jazyce C/C++ (B0B36PRP)
  - Základy programování v jazyce C/C++
  - Kompilace programů v jazyce C/C++
  - Základy objektového programování (znalost C++11 výhodou)

- Programování v jazyce C/C++ (B0B36PRP)
  - Základy programování v jazyce C/C++
  - Kompilace programů v jazyce C/C++
  - Základy objektového programování (znalost C++11 výhodou)
- Technologické předpoklady paralelizace (B4B36OSY)
  - Vlákna a jejich princip
  - Metody synchronizace a komunikace vláken


- Programování v jazyce C/C++ (B0B36PRP)
  - Základy programování v jazyce C/C++
  - Kompilace programů v jazyce C/C++
  - Základy objektového programování (znalost C++11 výhodou)
- Technologické předpoklady paralelizace (B4B36OSY)
  - Vlákna a jejich princip
  - Metody synchronizace a komunikace vláken
- Základní znalost fungování počítače a procesoru (B4B35APO)

- Programování v jazyce C/C++ (B0B36PRP)
  - Základy programování v jazyce C/C++
  - Kompilace programů v jazyce C/C++
  - Základy objektového programování (znalost C++11 výhodou)
- Technologické předpoklady paralelizace (B4B36OSY)
  - Vlákna a jejich princip
  - Metody synchronizace a komunikace vláken
- Základní znalost fungování počítače a procesoru (B4B35APO)
- Znalost základních algoritmů (B4B33ALG)

## Opakování

---



 tutorial\_01.zip

## Vygenerování build scriptů

```
cmake <src dir>
```

Zde <src dir> je složka se souborem CMakeLists.txt.

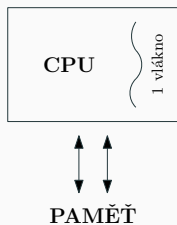
## Kompilace

```
cmake --build <build dir>
```

Zde <build dir> je složka s vygenerovanými soubory pro sestavení programu.

Nebo použijte IDE s dobrou podporou C++, například CLion (multiplatformní) nebo Visual Studio (Windows)

Pro připomenutí: Cílem paralelních výpočtů je dosáhnout zvýšení výkonu

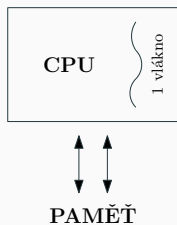


## von Neumannova architektura

- Jaké má nevýhody?
- Jak bychom je mohli opravit?
- A jak bychom dále mohli navýšit výkon procesoru?

👁 `memory.cpp`

Pro připomenutí: Cílem paralelních výpočtů je dosáhnout zvýšení výkonu



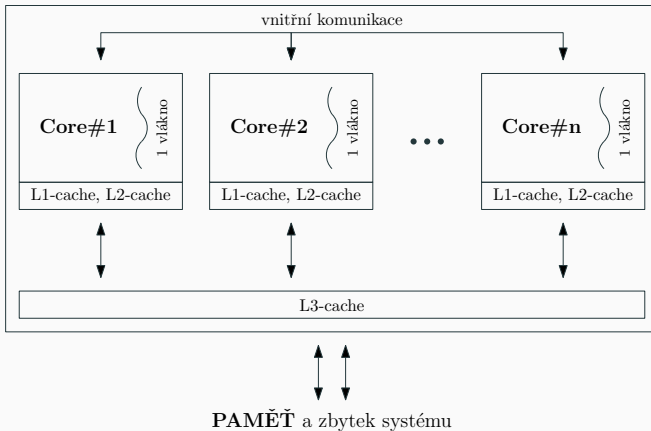
## von Neumannova architektura

- Jaké má nevýhody?
- Jak bychom je mohli opravit?
- A jak bychom dále mohli navýšit výkon procesoru?

👁 `memory.cpp`

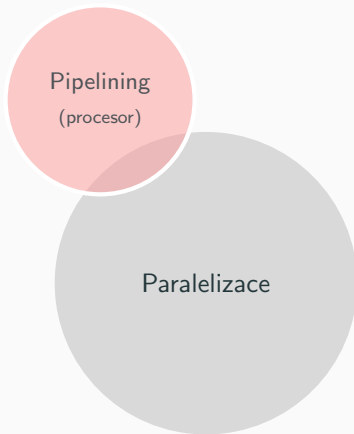
Vyzkoušejte si prosím, že Vám funguje přístup do BRUTE a odevzdejte soubor `memory.cpp`.

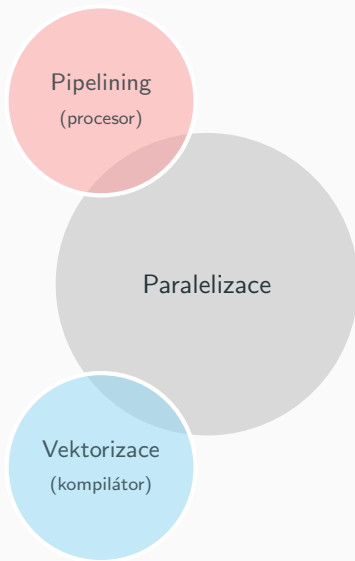
# Moderní procesor

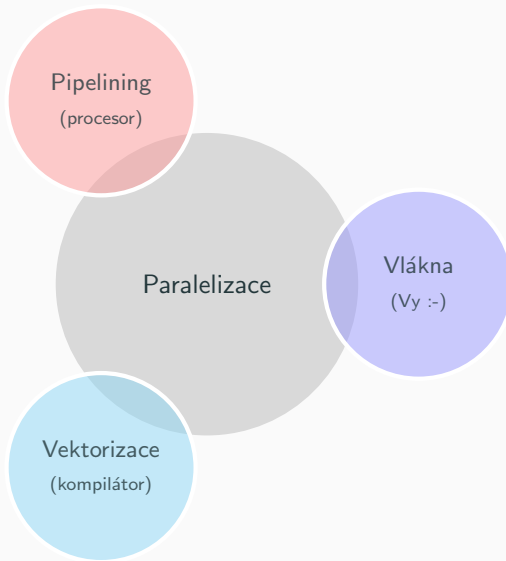




Paralelizace









Možné “nástrahy” použití moderního procesoru s více jádry a cache:

- Komunikace s pamětí je stále pomalá (problém *cache-miss*)
- Přístup ke sdíleným datům více vláken (*true sharing*)
- Udržování koherence cache může být drahé (*false sharing*)
- ... a jiné

👁 memory.cpp

👁 matrix.cpp

```
void multiply(int* number, int multiplyBy) {  
    *number = (*number) * multiplyBy;  
}
```

Předpokládejme `int number = 1;` a mějme dvě vlákna:

- Vlákno 1: `multiply(&number, 2)`
- Vlákno 2: `multiply(&number, 3)`

Co bude v proměnné `number` po skončení obou vláken?

```
void multiply(int * number, int multiplyBy) {  
    *number = (*number) * multiplyBy;  
}
```



```
imul esi, DWORD PTR [rdi]  
mov DWORD PTR [rdi], esi  
ret
```

```
void multiply(int * number, int multiplyBy) {  
    *number = (*number) * multiplyBy;  
}
```



```
imul esi, DWORD PTR [rdi]  
mov  DWORD PTR [rdi], esi  
ret
```

👁 <http://godbolt.org>

Vláknó 1 / `mov esi, 2`

---

```
imul esi, DWORD PTR [rdi]
mov DWORD PTR [rdi], esi
ret
```

Vláknó 2 / `mov esi, 3`

---

```
imul esi, DWORD PTR [rdi]
mov DWORD PTR [rdi], esi
ret
```

Vláknno 1 / `mov esi, 2`

---

```
imul esi, DWORD PTR [rdi]
mov DWORD PTR [rdi], esi
ret
```

Vláknno 2 / `mov esi, 3`

---

```
imul esi, DWORD PTR [rdi]
mov DWORD PTR [rdi], esi
ret
```

Vláknno 1 / `mov esi, 2`

---

```
imul esi, DWORD PTR [rdi]
mov DWORD PTR [rdi], esi
ret
```

Vláknno 2 / `mov esi, 3`

---

```
imul esi, DWORD PTR [rdi]
mov DWORD PTR [rdi], esi
ret
```

Výsledek:     `number = 6`



Vláknno 1 / `mov esi, 2`

---

```
imul esi, DWORD PTR [rdi]
mov DWORD PTR [rdi], esi
ret
```

Vláknno 2 / `mov esi, 3`

---

```
imul esi, DWORD PTR [rdi]
mov DWORD PTR [rdi], esi
ret
```

Vláknno 1 / `mov esi, 2`

---

```
imul esi, DWORD PTR [rdi]
mov DWORD PTR [rdi], esi
ret
```

Vláknno 2 / `mov esi, 3`

---

```
imul esi, DWORD PTR [rdi]
mov DWORD PTR [rdi], esi
ret
```

Výsledek:      `number = 6`

Vláknó 1 / `mov esi, 2`

---

```
imul esi, DWORD PTR [rdi]
```

```
mov DWORD PTR [rdi], esi  
ret
```

Vláknó 2 / `mov esi, 3`

---

```
imul esi, DWORD PTR [rdi]
```

```
mov DWORD PTR [rdi], esi  
ret
```

Vláknno 1 / `mov esi, 2`

---

```
imul esi, DWORD PTR [rdi]
```

```
mov DWORD PTR [rdi], esi  
ret
```

Vláknno 2 / `mov esi, 3`

---

```
imul esi, DWORD PTR [rdi]
```

```
mov DWORD PTR [rdi], esi  
ret
```

Výsledek:     `number = 3`

Vláknno 1 / `mov esi, 2`

---

```
imul esi, DWORD PTR [rdi]
```

```
mov DWORD PTR [rdi], esi  
ret
```

Vláknno 2 / `mov esi, 3`

---

```
imul esi, DWORD PTR [rdi]  
mov DWORD PTR [rdi], esi  
ret
```

Vláknno 1 / `mov esi, 2`

---

```
imul esi, DWORD PTR [rdi]
```

```
mov DWORD PTR [rdi], esi  
ret
```

Vláknno 2 / `mov esi, 3`

---

```
imul esi, DWORD PTR [rdi]  
mov DWORD PTR [rdi], esi  
ret
```

Výsledek:      `number = 2`

Vláknno 1 / `mov esi, 2`

---

```
imul esi, DWORD PTR [rdi]
```

```
mov DWORD PTR [rdi], esi  
ret
```

Vláknno 2 / `mov esi, 3`

---

```
imul esi, DWORD PTR [rdi]  
mov DWORD PTR [rdi], esi  
ret
```

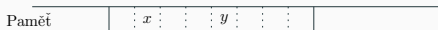
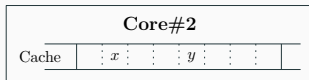
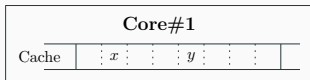
---

Jaké máme možnosti, abychom dosáhli deterministického výsledku (který pravděpodobně chceme)?

# False-sharing

Moderní procesor pracuje s pamětí *po blocích*, které se mapují do cache.

- I když vlákna nepracují se stejnými proměnnými, mohou chtít pracovat se stejným *blokem*.
- Jeden blok se pak nutně musí nacházet v cachích různých jader – a ve více kopiích

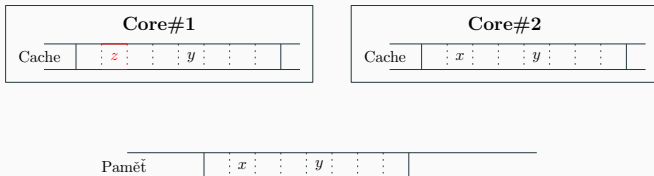




# False-sharing

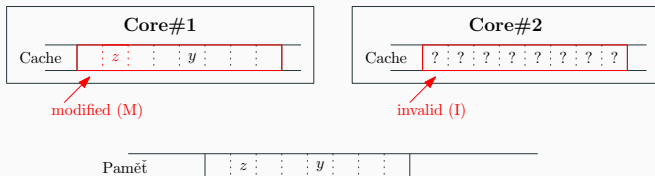
Moderní procesor pracuje s pamětí *po blocích*, které se mapují do cache.

- I když vlákna nepracují se stejnými proměnnými, mohou chtít pracovat se stejným *blokem*.
- Jeden blok se pak nutně musí nacházet v cachích různých jader – a ve více kopiích



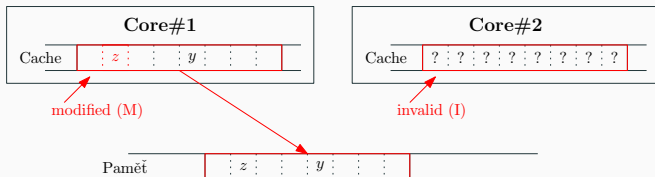
Moderní procesor pracuje s pamětí *po blocích*, které se mapují do cache.

- I když vlákna nepracují se stejnými proměnnými, mohou chtít pracovat se stejným *blokem*.
- Jeden blok se pak nutně musí nacházet v cachích různých jader – a ve více kopiích



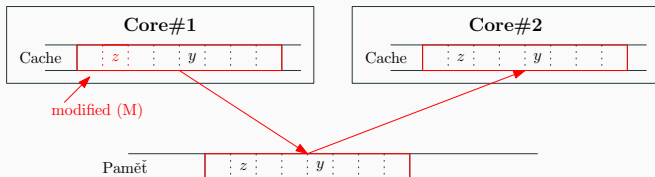
Moderní procesor pracuje s pamětí *po blocích*, které se mapují do cache.

- I když vlákna nepracují se stejnými proměnnými, mohou chtít pracovat se stejným *blokem*.
- Jeden blok se pak nutně musí nacházet v cachích různých jader – a ve více kopiích



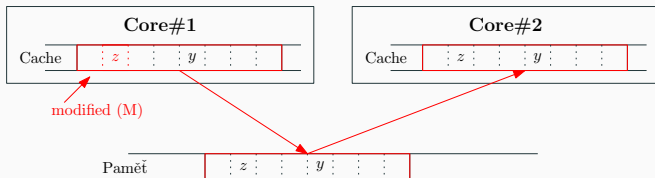
Moderní procesor pracuje s pamětí *po blocích*, které se mapují do cache.

- I když vlákna nepracují se stejnými proměnnými, mohou chtít pracovat se stejným *blokem*.
- Jeden blok se pak nutně musí nacházet v cachích různých jader – a ve více kopiích



Moderní procesor pracuje s pamětí *po blocích*, které se mapují do cache.

- I když vlákna nepracují se stejnými proměnnými, mohou chtít pracovat se stejným *blokem*.
- Jeden blok se pak nutně musí nacházet v cachích různých jader – a ve více kopiích



- Ale právě té komunikaci s pamětí jsme se chtěli použitím cache vyhnout!

👁 `false_sharing.cpp`

## Paralelizace v praxi

---

### Je následující tvrzení pravdivé?

Mějme procesor s  $p$  jádry a úlohu, která při využití jednoho jádra trvá  $T$  milisekund. Využijeme-li všech  $p$  jader pro vyřešení úlohy, vyřešení úlohy zvládneme za  $T/p$  milisekund.



### Je následující tvrzení pravdivé?

Mějme procesor s  $p$  jádry a úlohu, která při využití jednoho jádra trvá  $T$  milisekund. Využijeme-li všech  $p$  jader pro vyřešení úlohy, vyřešení úlohy zvládneme za  $T/p$  milisekund.

**Tvrzení není pravdivé.** Proč? Zkuste vymyslet co možná nejvíce důvodů, proč tomu tak není.

### Je následující tvrzení pravdivé?

Mějme procesor s  $p$  jádry a úlohu, která při využití jednoho jádra trvá  $T$  milisekund. Využijeme-li všech  $p$  jader pro vyřešení úlohy, vyřešení úlohy zvládneme za  $T/p$  milisekund.

**Tvrzení není pravdivé.** Proč? Zkuste vymyslet co možná nejvíce důvodů, proč tomu tak není.

O úlohách, kde toto tvrzení platí říkáme, že jsou tzv. *lineární* nebo také *embarrassingly parallel*. Takových úloh ale v praxi potkáme velmi málo.

## Je následující tvrzení pravdivé?

Mějme pole o 1,000,000 prvků. S každým prvkem pole máme za úkol  $100\times$  provést “magickou operaci”  $x \leftarrow e^{\ln x}$ . Tuto úlohu lze dobře paralelizovat.

```
void magic_operation(double * array) {  
    for(unsigned int i = 0 ; i < 1000000 ; i++) {  
        for(unsigned int k = 0 ; k < 500 ; k++) {  
            array[i] = exp(log(array[i]));  
        }  
    }  
}
```

## Je následující tvrzení pravdivé?

Mějme pole o 1,000,000 prvků. S každým prvkem pole máme za úkol  $100\times$  provést “magickou operaci”  $x \leftarrow e^{\ln x}$ . Tuto úlohu lze dobře paralelizovat.

```
void magic_operation(double * array) {  
    for(unsigned int i = 0 ; i < 1000000 ; i++) {  
        for(unsigned int k = 0 ; k < 500 ; k++) {  
            array[i] = exp(log(array[i]));  
        }  
    }  
}
```

**Tvrzení je pravdivé.** Jednotlivé výpočty hodnot `array[i]` na sobě nezávisí a můžeme je rozložit mezi různá vlákna a dosáhnout téměř lineárního nárůstu výkonu.

## Je následující tvrzení pravdivé?

Mějme pole o 1,000,000 prvků. S každým prvkem pole máme za úkol  $100\times$  provést “magickou operaci”  $x \leftarrow e^{\ln x}$ . Tuto úlohu lze dobře paralelizovat.

```
void magic_operation(double * array) {  
    for(unsigned int i = 0 ; i < 1000000 ; i++) {  
        for(unsigned int k = 0 ; k < 500 ; k++) {  
            array[i] = exp(log(array[i]));  
        }  
    }  
}
```

**Tvrzení je pravdivé.** Jednotlivé výpočty hodnot `array[i]` na sobě nezávisí a můžeme je rozložit mezi různá vlákna a dosáhnout téměř lineárního nárůstu výkonu.

---

A nebo bychom si mohli vzpomenout, že  $\ln x$  a  $e^x$  jsou inverzní funkce. Ale to bychom neměli co paralelizovat ;-)

## Je následující tvrzení pravdivé?

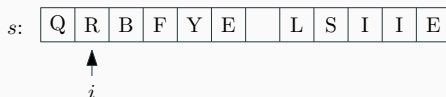
Mějme pole o 1,000,000 prvků. S každým prvkem pole máme za úkol  $100\times$  provést “magickou operaci”  $x \leftarrow e^{\ln x}$ . Tuto úlohu lze dobře paralelizovat.

```
void magic_operation(double * array) {  
    for(unsigned int i = 0 ; i < 1000000 ; i++) {  
        for(unsigned int k = 0 ; k < 500 ; k++) {  
            array[i] = exp(log(array[i]));  
        }  
    }  
}
```

Proč jsme ale nedosáhli  $s$ -násobného zrychlení (kde  $s$  je počet jader procesoru?). Vzpomeňte si na Amdahlův zákon.

$$S = \frac{1}{(1 - p) + \frac{p}{s}}$$

Dokážete říct, co tvoří neparalelizovatelnou část programu? (vyžadující  $(1 - p)\%$  času)



Jeden krok dešifrování:

- $s_i \leftarrow \left[ s_i + p_1 \times \text{secret}(\overbrace{s_{[i-2..i+2]}}^{\text{EQRBF}}) \right] \bmod |\Sigma|$
- $i \leftarrow \left[ i + p_2 \times \text{secret}(s_{[i-2..i+2]}) \right] \bmod |s|$

... opakován  $N$ -krát

**Úkol:** Doimplementujte dešifrovací pravidlo do metody `decrypt` v souboru `PDVCrypt.cpp`.

## Je následující tvrzení pravdivé?

Proces dešifrování řetězce zašifrovaného pomocí PDVCrypt lze snadno paralelizovat.



## Je následující tvrzení pravdivé?

Proces dešifrování řetězce zašifrovaného pomocí PDVCrypt lze snadno paralelizovat.

**Tvrzení není pravdivé.** Proč paralelní verze dešifrovacího algoritmu vůbec nefunguje?

## Je následující tvrzení pravdivé?

Proces dešifrování řetězce zašifrovaného pomocí PDVCrypt lze snadno paralelizovat.

**Tvrzení není pravdivé.** Proč paralelní verze dešifrovacího algoritmu vůbec nefunguje?

---

Uvažujte množinu zašifrovaných řetězců, které máte za úkol dešifrovat. Mohli bychom využít více jader v tomto případě?

**Díky za pozornost!**

Budeme rádi za Vaši  
zpětnou vazbu! →



[https://forms.gle/  
q6dTqGkjFfMdKMTr5](https://forms.gle/q6dTqGkjFfMdKMTr5)