Urychlování výpočtů, paralelizace a profiling

Ing. Filip Vaverka

Brno University of Technology, Faculty of Information Technology
Božetěchova 1/2. 612 66 Brno - Královo Pole

ivaverka@fit.vutbr.cz

Proč optimalizovat?

- Zkrácení doby výpočtu
  - Umožnit běh aplikací pracujících v reálném čase (řídící systémy, hry, ...)
  - Zajištění užitečnosti výsledků (simulace, predikce, ...)
  - Včasné zpracování příchozích dat
  - Zajištění pohodlí uživatele
- Snížení množství energie potřebné pro výpočet
  - Primárně mobilní a energeticky omezená zařízení
  - ale také vysoce náročné počítání

Co je to optimalizace?

- Hledání vhodné formulace problému
- Volba vhodné platformy pro řešení daného problému
- Efektivní využití dostupných zdrojů pro řešení daného problému

Návrh řešení problému

1. Hrubá formulace problému, způsobu řešení a požadavků
   - Prvotní analýza umožňující odhad výpočetní náročnosti řešení problému
2. Volba platformy vhodné pro zvolený způsob řešení
   - Bude CPU dostačující? Je úloha vhodná pro akcelerátory?
   - Kolik paměti je třeba?
3. Návrh a implementace řešení pro zvolenou platformu
   - Rozdělení úlohy mezi výpočetní jednotky
   - Volba/Návrh konkrétních algoritmů
4. Optimalizace implementovaného řešení
   - Hledání kritických míst v programu
   - Analýza využití HW prostředků
   - Modifikace implementace pro zlepšení využití HW

Postup optimalizace – volba algoritmů

- Většina problémů má více možných formulací a přístupů k řešení (algoritmů)

Postup optimalizace – volba algoritmů

- Většina problémů má více možných formulací a přístupů k řešení (algoritmů)
- Např.: Nalezení minima v poli
  - Porovnání doposud nejmenšího prvku s následujícími
  - Seřazení celého pole a výběr prvního prvku

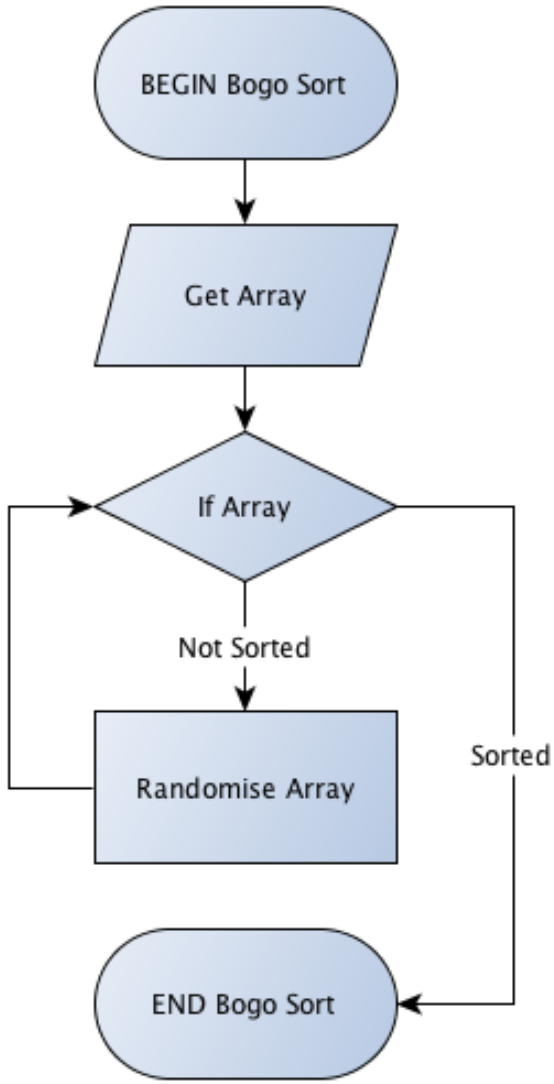
Postup optimalizace – volba algoritmů

- Většina problémů má více možných formulací a přístupů k řešení (algoritmů)
- Základním ukazatelem je **asymptotická složitost** algorimu
  - Obvykle uvažujeme **časovou**, lze ale určit i **paměťovou**
  - Značíme $\mathcal{O}(n)$, kde n je počet vstupních prvků
  - Funkce vyjadřující počet nutných operací pro vstup délky n

Postup optimalizace – volba algoritmů

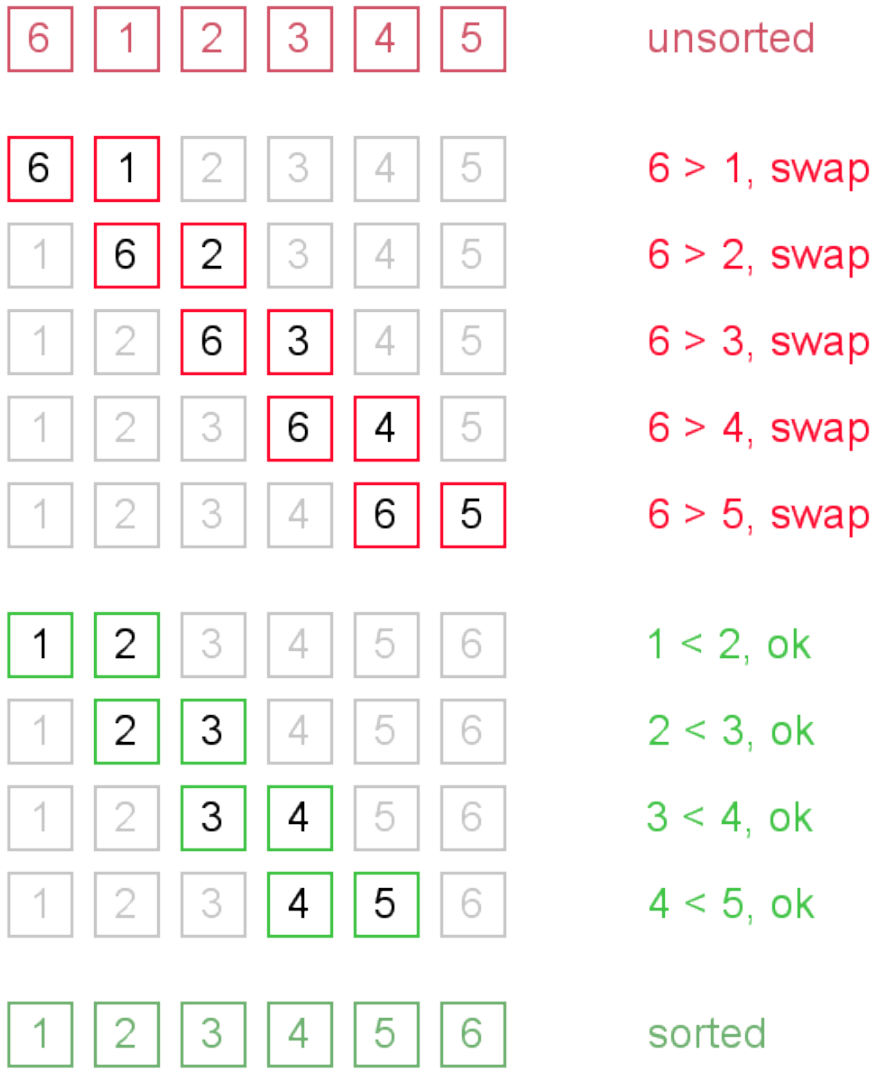
- Většina problémů má více možných formulací a přístupů k řešení (algoritmů)
- Základním ukazatelem je **asymptotická složitost** algorimu
  - Obvykle uvažujeme **časovou**, lze ale určit i **paměťovou**
  - Značíme $\mathcal{O}(n)$, kde n je počet vstupních prvků
  - Funkce vyjadřující počet nutných operací pro vstup délky n
- Je třeba pamatovat na vlastnosti zvolené platformy!
  - Obsahuje více-jádrový pocesor?
  - Obsahuje GPU?

- Bogosort
  - T_{worst}: ∞
  - T_{avg}: $\mathcal{O}((n+1)!)$
  - S_{worst}: $\mathcal{O}(n)$



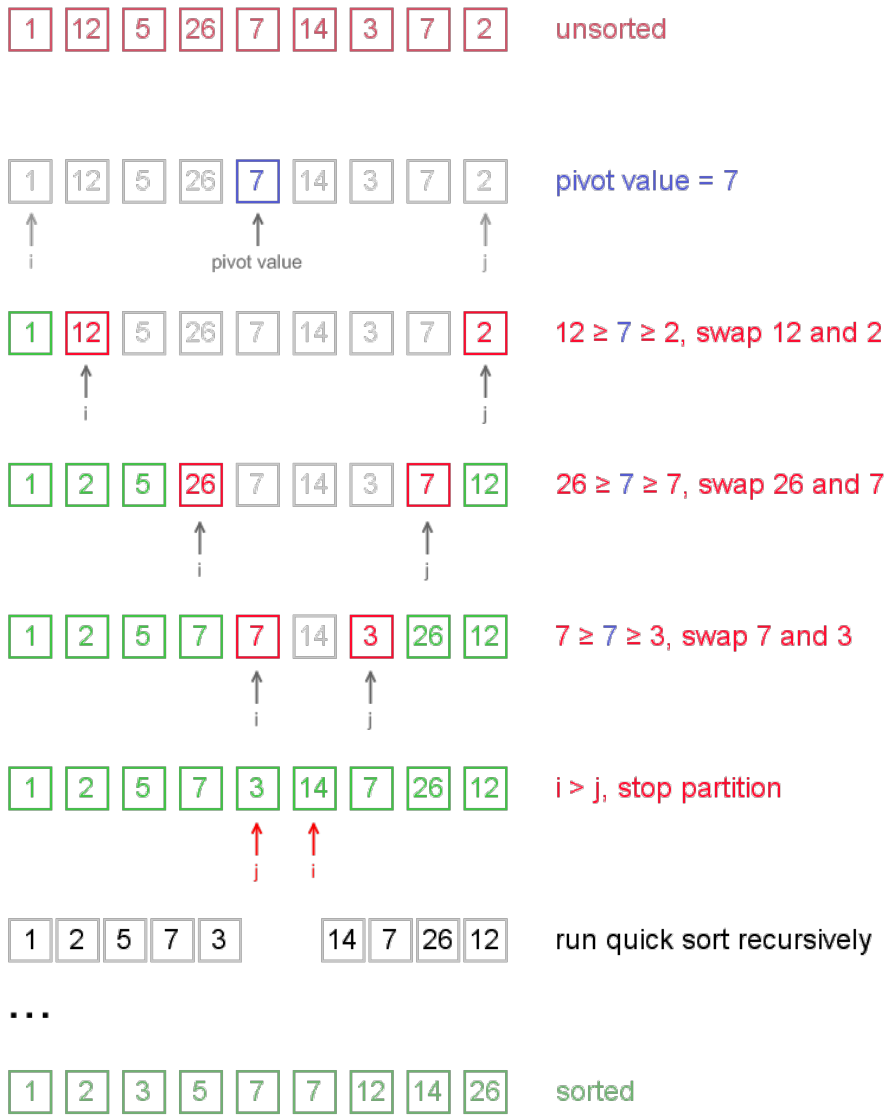
Volba algoritmů – asymptotická složitost

- Bogosort
  - T_{worst}: ∞
  - T_{avg}: $\mathcal{O}((n+1)!)$
  - S_{worst}: $\mathcal{O}(n)$
- Bubble-Sort
  - T_{worst}: $\mathcal{O}(n^2)$
  - T_{avg}: $\mathcal{O}(n^2)$
  - S_{worst}: $\mathcal{O}(1)$

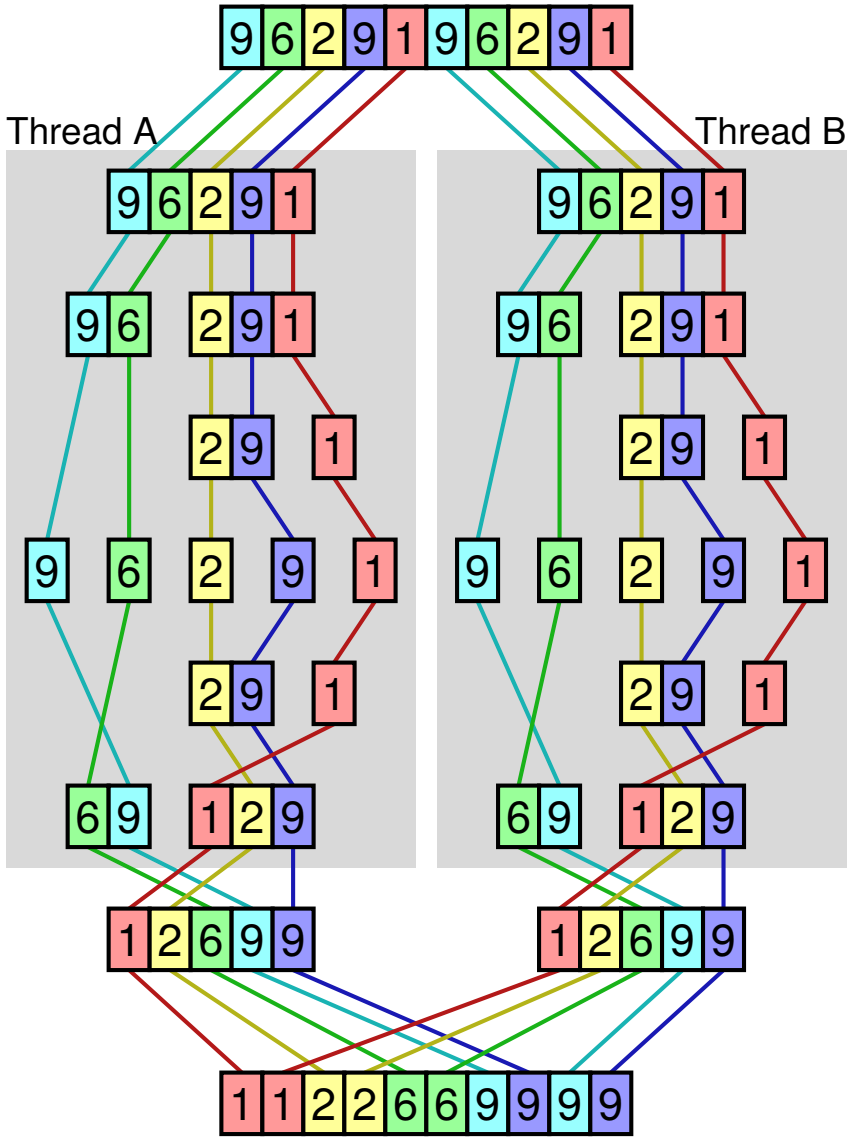


Volba algoritmů – asymptotická složitost

- Bogosort
  - T_{worst}: ∞
  - T_{avg}: $\mathcal{O}((n+1)!)$
  - S_{worst}: $\mathcal{O}(n)$
- Bubble-Sort
  - T_{worst}: $\mathcal{O}(n^2)$
  - T_{avg}: $\mathcal{O}(n^2)$
  - S_{worst}: $\mathcal{O}(1)$
- Quick-Sort
  - T_{worst}: $\mathcal{O}(n^2)$
  - T_{avg}: $\mathcal{O}(n \log n)$
  - S_{worst}: $\mathcal{O}(\log n)$

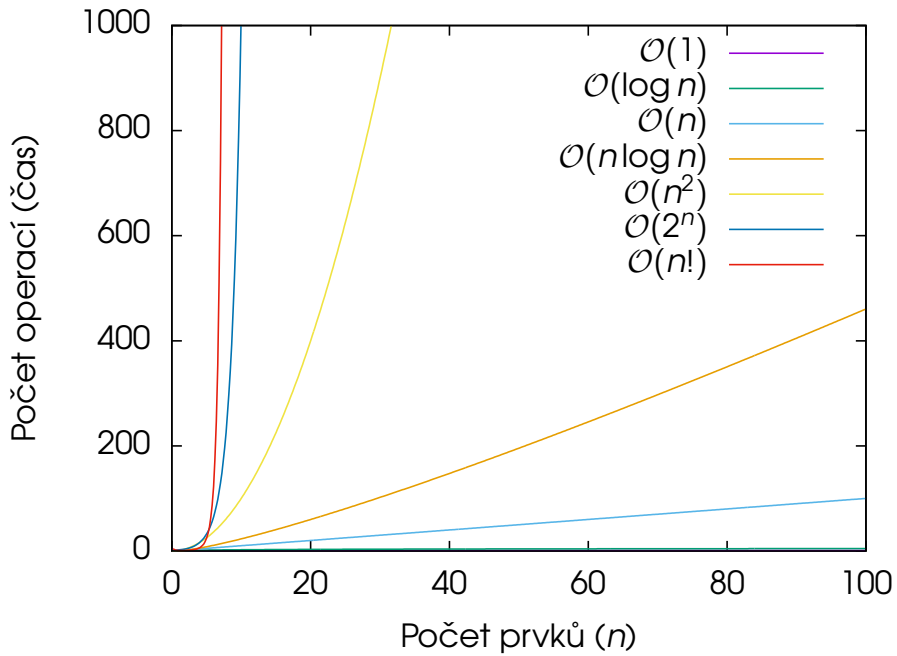


- Bogosort
  - T_{worst}: ∞
  - T_{avg}: $\mathcal{O}((n+1)!)$
  - S_{worst}: $\mathcal{O}(n)$
- Bubble-Sort
  - T_{worst}: $\mathcal{O}(n^2)$
  - T_{avg}: $\mathcal{O}(n^2)$
  - S_{worst}: $\mathcal{O}(1)$
- Quick-Sort
  - T_{worst}: $\mathcal{O}(n^2)$
  - T_{avg}: $\mathcal{O}(n \log n)$
  - S_{worst}: $\mathcal{O}(\log n)$
- Merge-Sort
  - T_{worst}: $\mathcal{O}(n \log n)$
  - T_{avg}: $\mathcal{O}(n \log n)$
  - S_{worst}: $\mathcal{O}(n)$



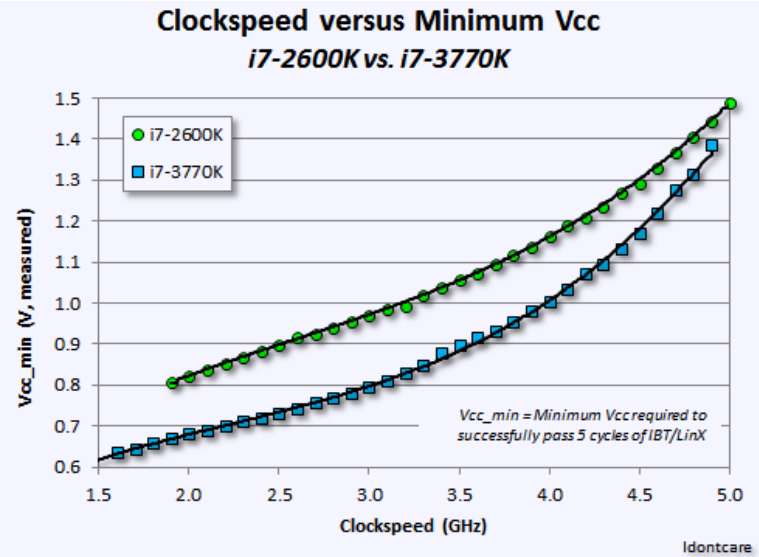
Volba algoritmů – asymptotická složitost

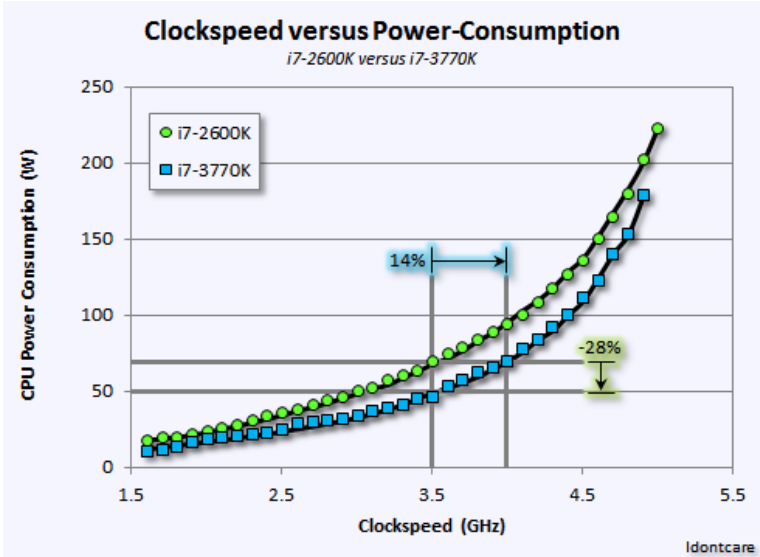
- Bogosort
  - T_{worst}: ∞
  - T_{avg}: $\mathcal{O}((n+1)!)$
  - S_{worst}: $\mathcal{O}(n)$
- Bubble-Sort
  - T_{worst}: $\mathcal{O}(n^2)$
  - T_{avg}: $\mathcal{O}(n^2)$
  - S_{worst}: $\mathcal{O}(1)$
- Quick-Sort
  - T_{worst}: $\mathcal{O}(n^2)$
  - T_{avg}: $\mathcal{O}(n \log n)$
  - S_{worst}: $\mathcal{O}(\log n)$
- Merge-Sort
  - T_{worst}: $\mathcal{O}(n \log n)$
  - T_{avg}: $\mathcal{O}(n \log n)$
  - S_{worst}: $\mathcal{O}(n)$



Strategie zrychlování procesorů (CPU)

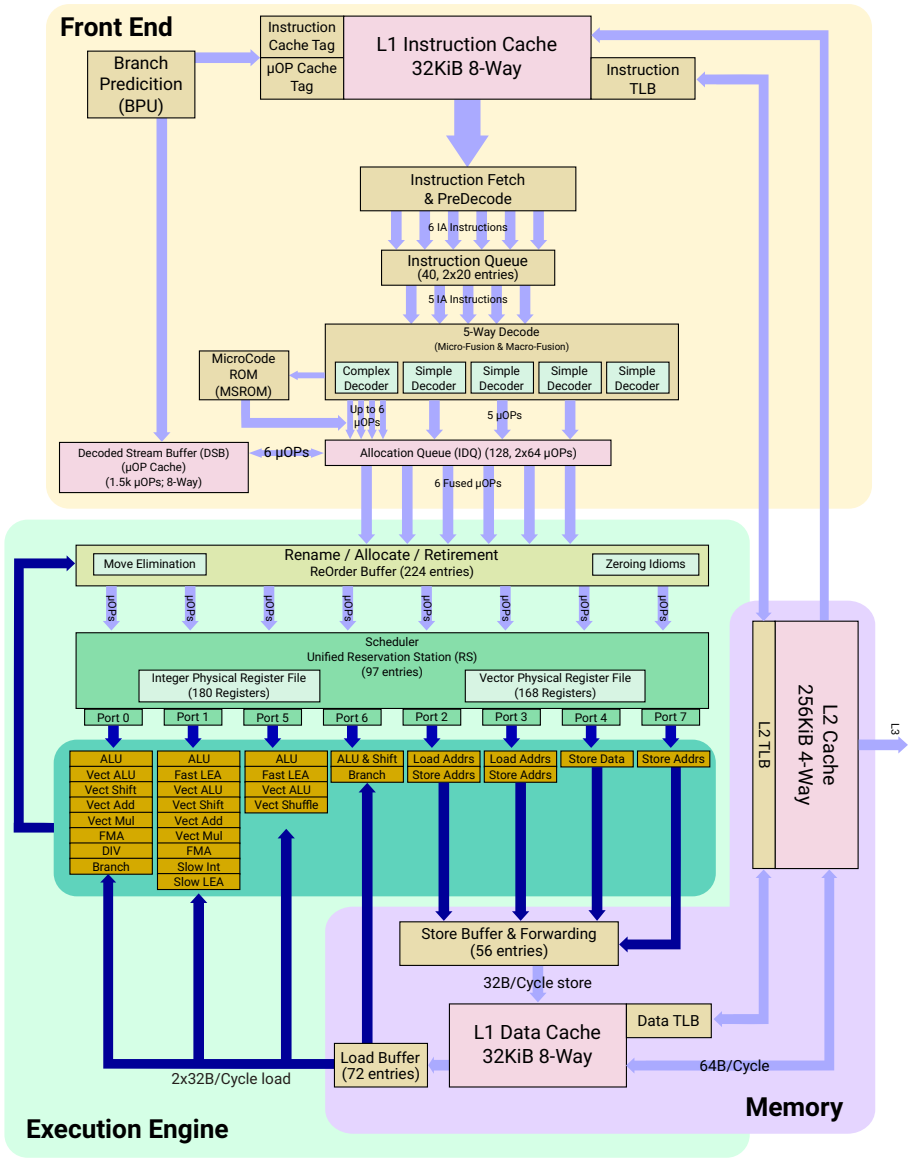
- Zvyšování pracovní frekvence
  - Limitováno příkonem: $P = C_L(U_{dd}^2/2)f_{\text{clk}}SW$
  - Pracovní napětí U_{dd} roste s frekvencí f_{clk}





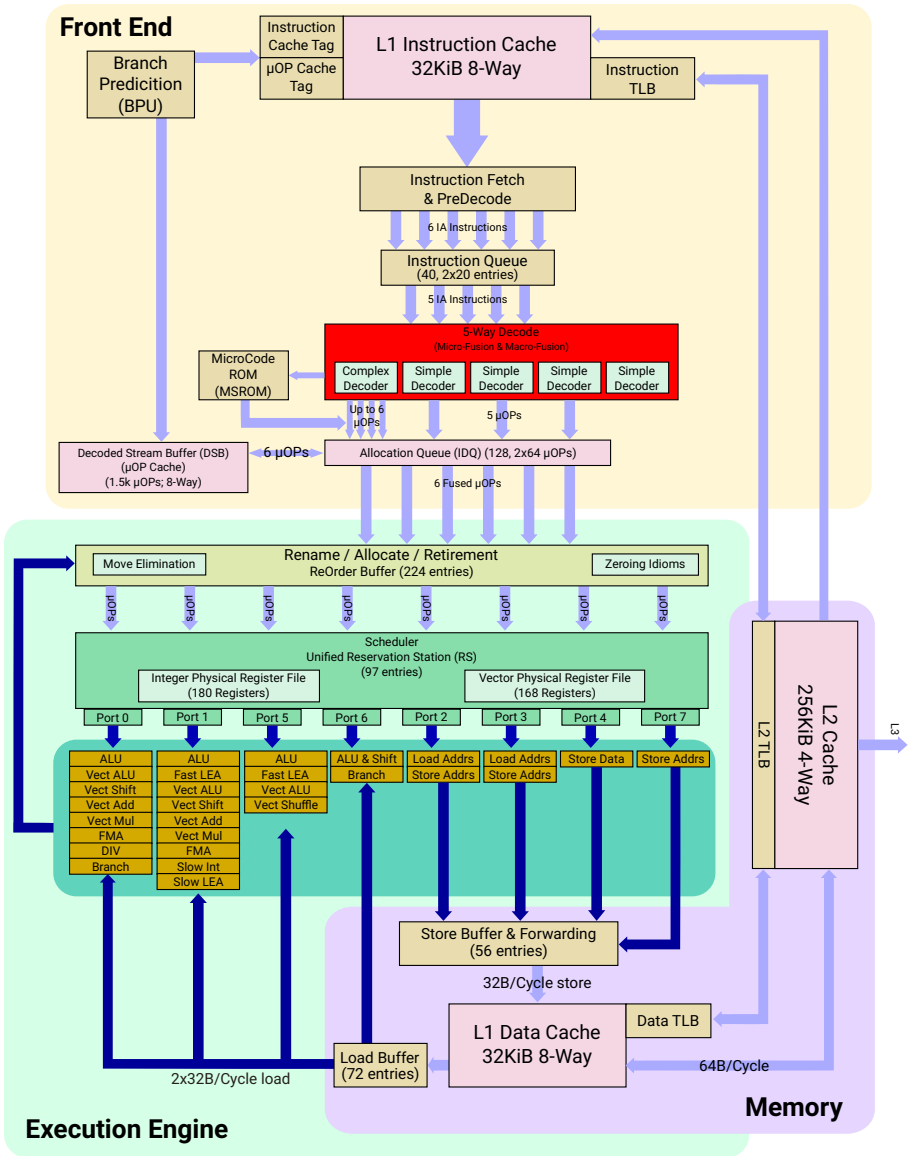
- Rozšiřování úrovně paralelismu
  - Na úrovni instrukcí (super-skalární architektury – IPC)
  - Na úrovni dat (SIMD – MMX, SSE, AVX)
  - Na úrovni vláken (SMT) a jader (MIMD)

- Skrytý paralelismus

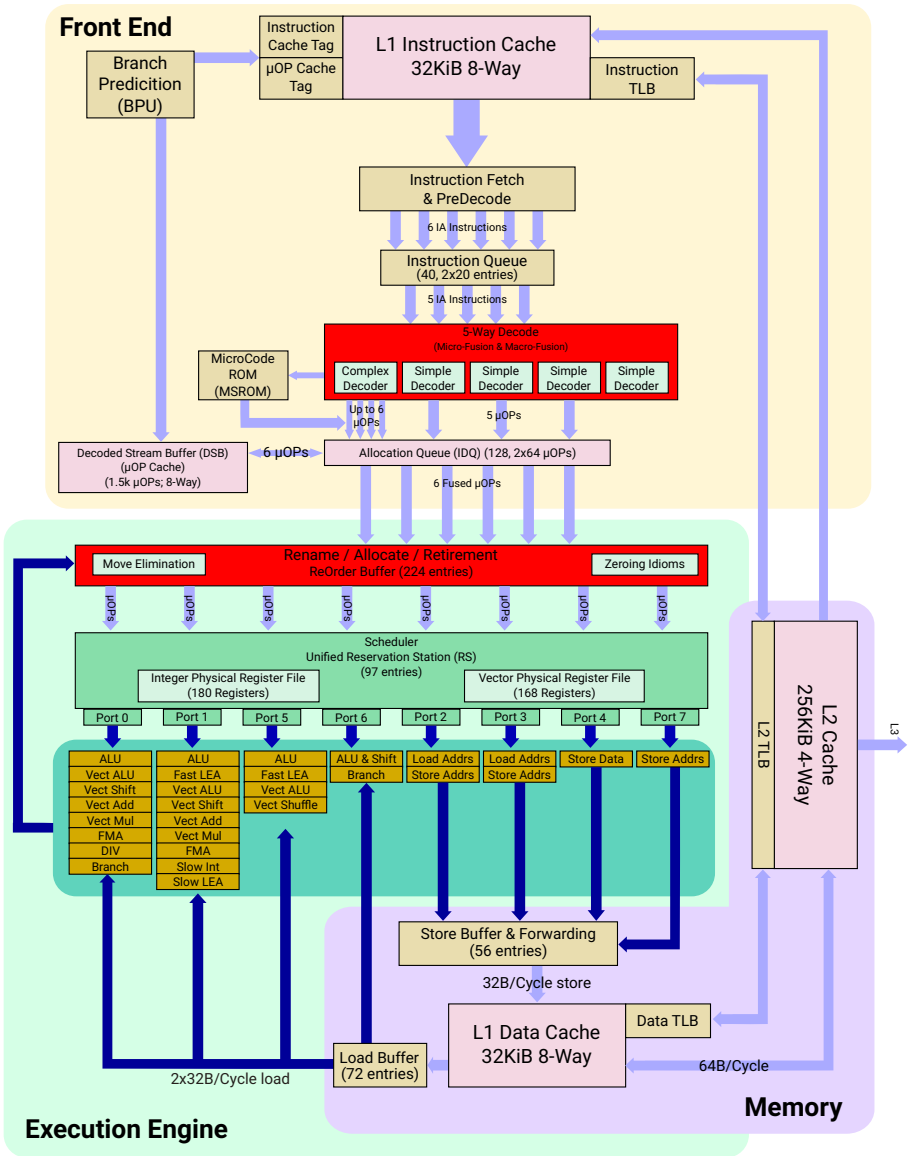


Paralelismus v moderním CPU

- Skrytý paralelismus
  - Lze rozpracovat až 5 instrukcí v cyklu

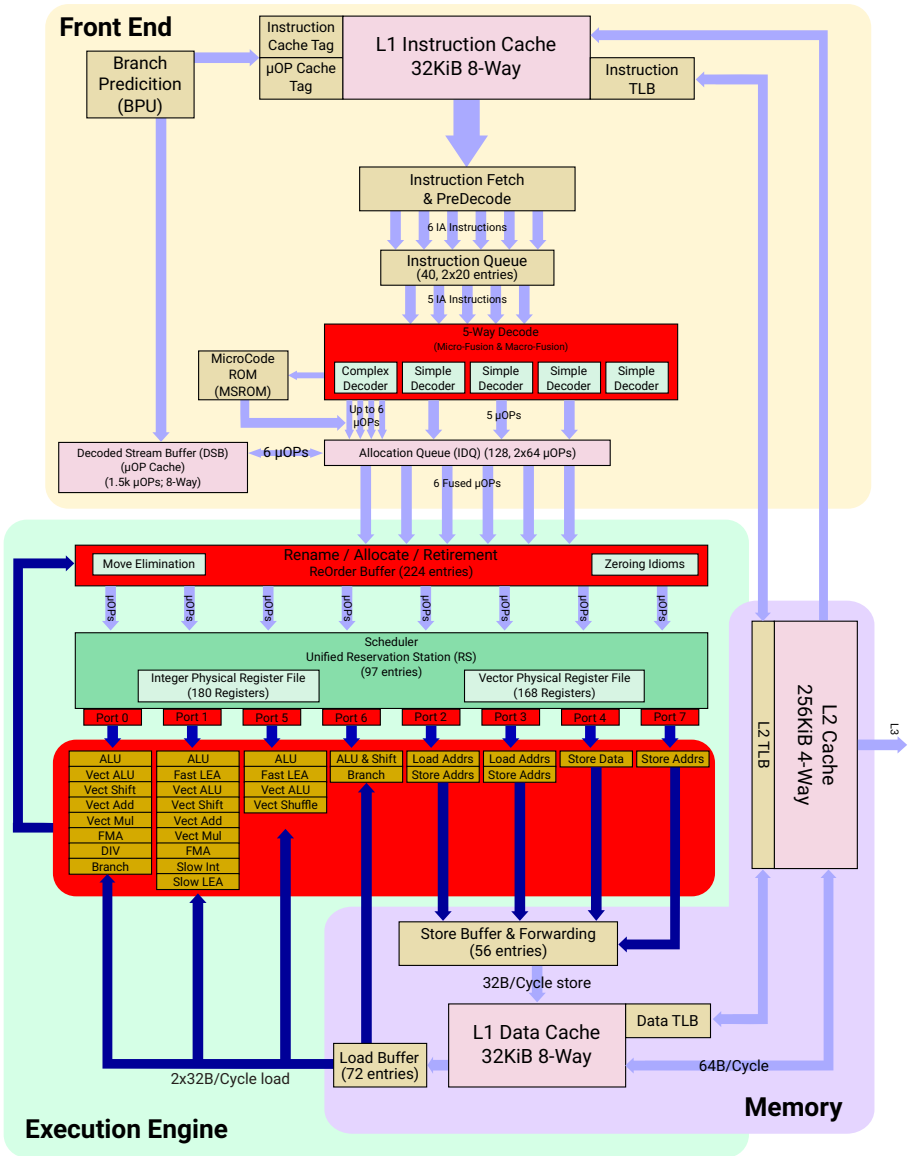


- Skrytý paralelismus
  - Lze rozpracovat až 5 instrukcí v cyklu
  - Až 224 rozpracovaných μ-operací

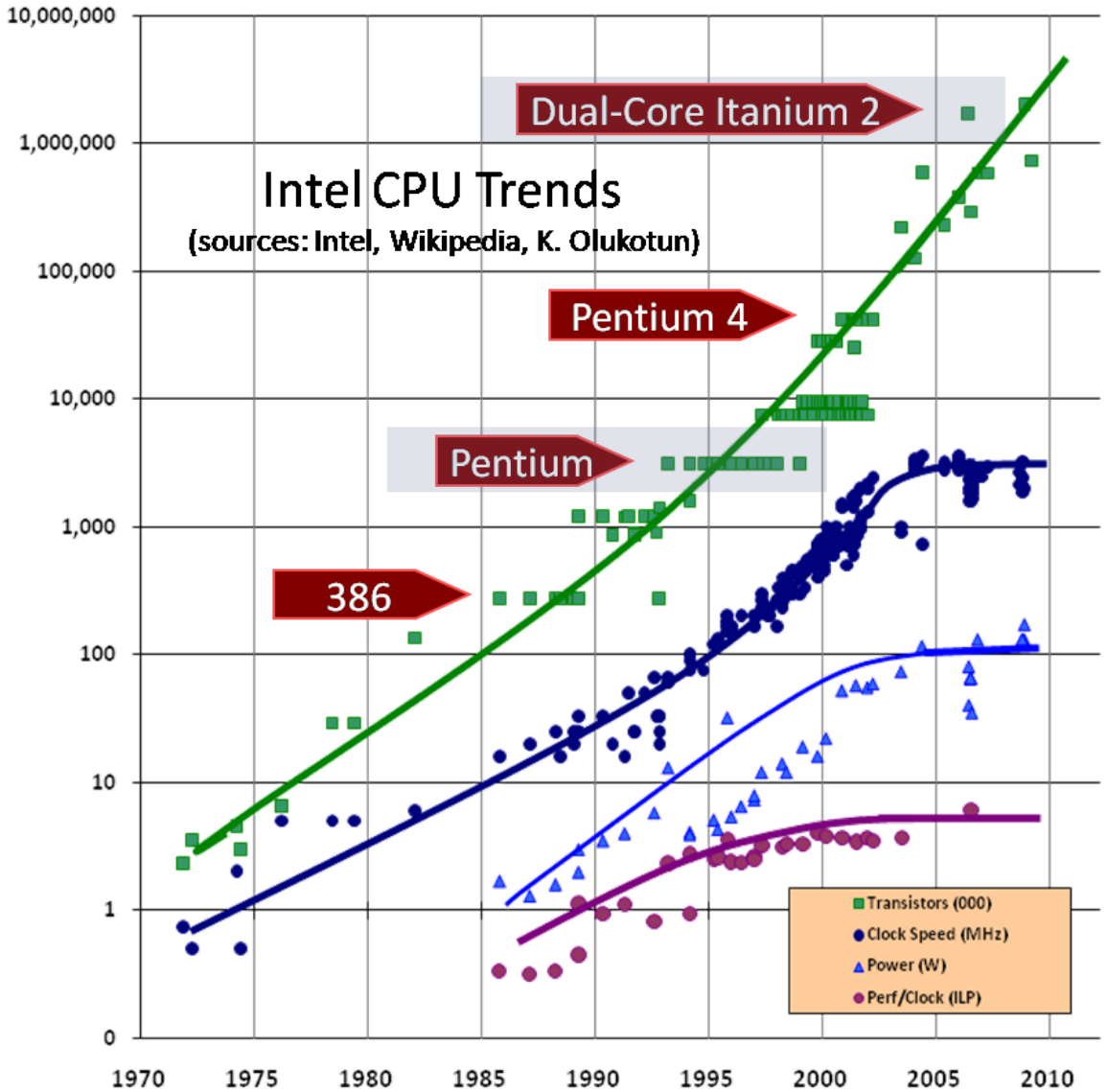


Paralelismus v moderním CPU

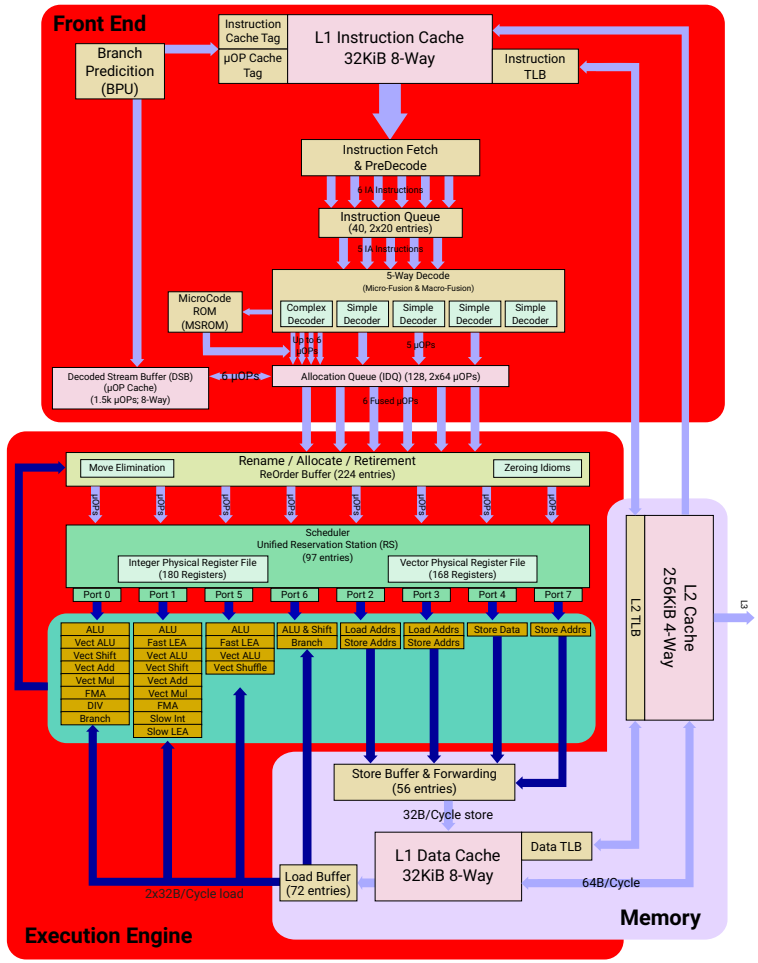
- Skrytý paralelismus
  - Lze rozpracovat až 5 instrukcí v cyklu
  - Až 224 rozpracovaných μ-operací
  - 7 paralelně pracujících jednotek
- Nelze donekonečna



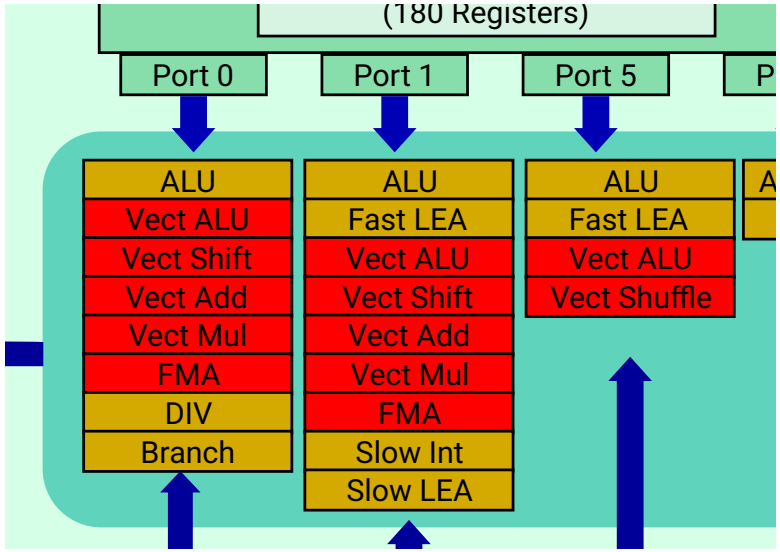
- Jak nejlépe využít stále rostoucí počet tranzistorů?

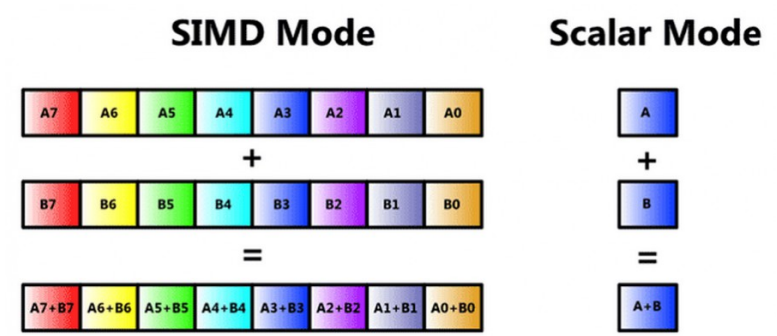


- Explicitní paralelismus
  - SMT (Hyper-threading)



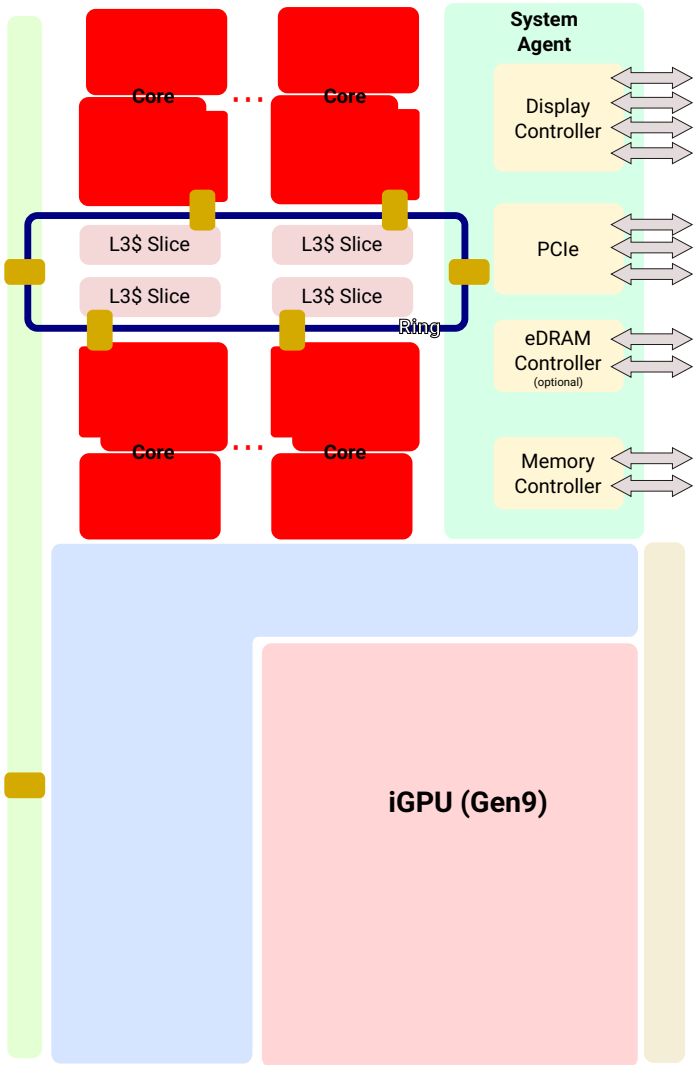
- Explicitní paralelismus
  - SMT (Hyper-threading)
  - Vektorové operace (SIMD)





Paralelismus v moderním CPU

- Explicitní paralelismus
  - SMT (Hyper-threading)
  - Vektorové operace (SIMD)
  - Více jader se sdílenou pamětí (MIMD)



Moderní hardware (CPU)

- Mobilní CPU
  - Obvykle 64-bit ARM architektura
  - až 8 jader/vláken (často 4+4)
  - až 2.5 GHz
  - Cache: 2 MB L2 cache (bez L3)
- Desktop CPU
  - Nejčastěji 64-bit x86 architektura
  - až 10 jader (20 vláken)
  - až 4.0 GHz
  - Cache: 256 KB/jádro L2 + 25 MB L3
- Server/Cluster CPU
  - Nejčastěji 64-bit x86 architektura
  - až 32 jader (64 vláken)
  - až 2.8 GHz
  - Cache: 512 KB/jádro L2 + 64 MB L3

Postup optimalizace – nízká úroveň

- Optimalizace sekvenčního programu (1 vlákno)
- Primárně využívat schopností překladače
  - Přepínače jako: `-O3`, `-march=native`, `-msseX`, `-mavxX`
- Pozor na "premature optimization"
  - Nejdříve profilovat, pak optimalizovat (kde je třeba)
- Asembler a "intrisic funkce"
  - Jen v krajních případech – znemožňují přenostitelnost kódu

Profiling (CPU)

- Na základě běhu, nebo simulace běhu aplikace umožňuje vyhodnotit vlastnosti jednotlivých částí kódu
- Mezi takto získaná data patří
  - Počet provedených instrukcí (a rozdělení do kategorií)
  - Počet instrukcí vykonaných během jednoho cyklu (IPC)
  - Počet vykonaných skoků a úspěšnost jejich předpovídání
  - Počet přístupů do pamětí cache L1, L2, L3
  - Úspěšnost hledání dat v pamětech cache (hit/miss ratio)
- Pomocí těchto informací lze určit jak dobře je daný procesor využíván
- Cílem je nalézt části aplikace, které vykazují nízkou efektivitu (a odhalit příčinu)

Profiling – Valgrind/Cachegrind

- Simulací provádění programu umí získat:
  - Informace o počtu provedených instrukcí

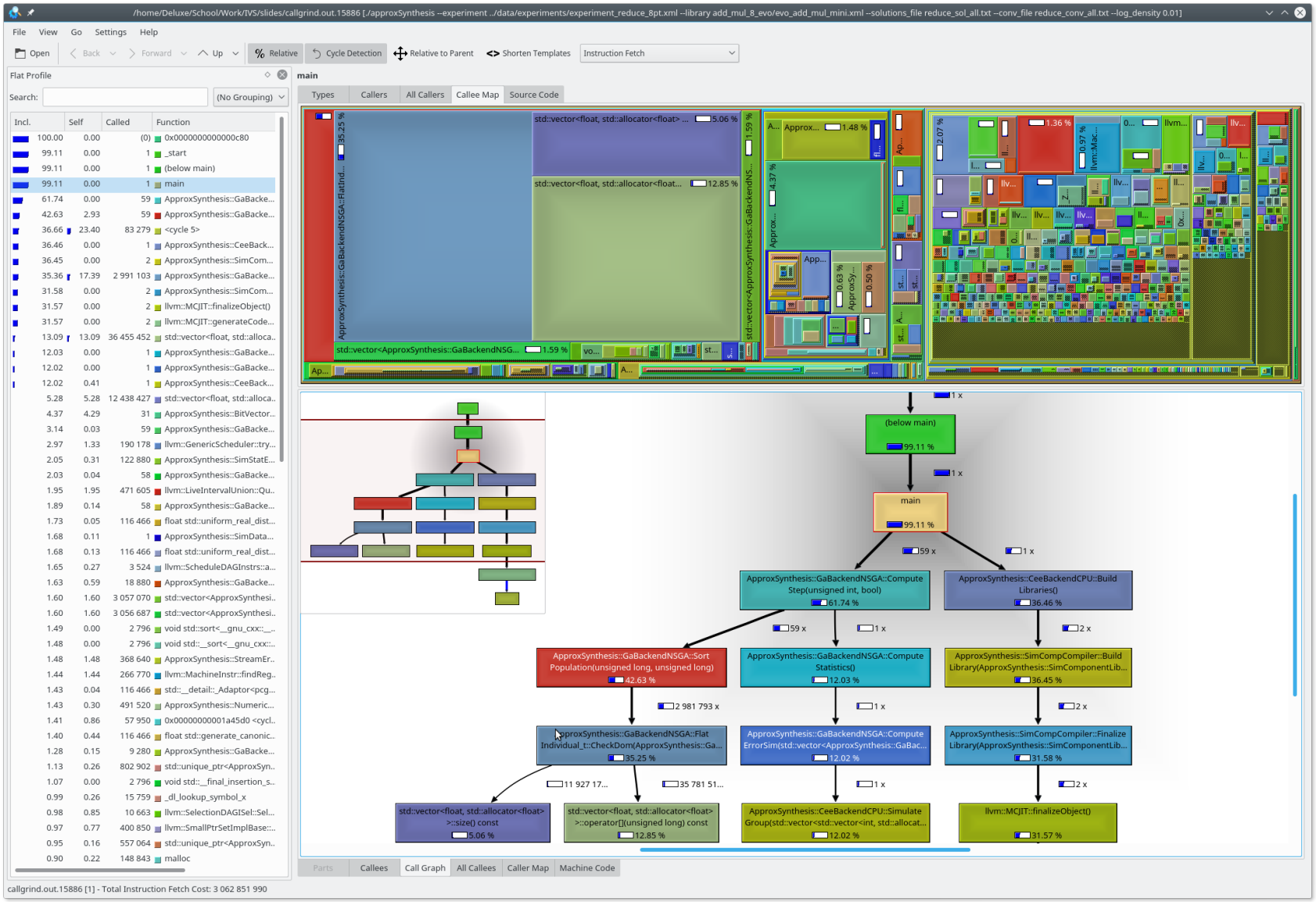
    ```
    > valgrind --tool=callgrind <cesta-k-bin>
    ```

  - Informace o práci s pamětí cache

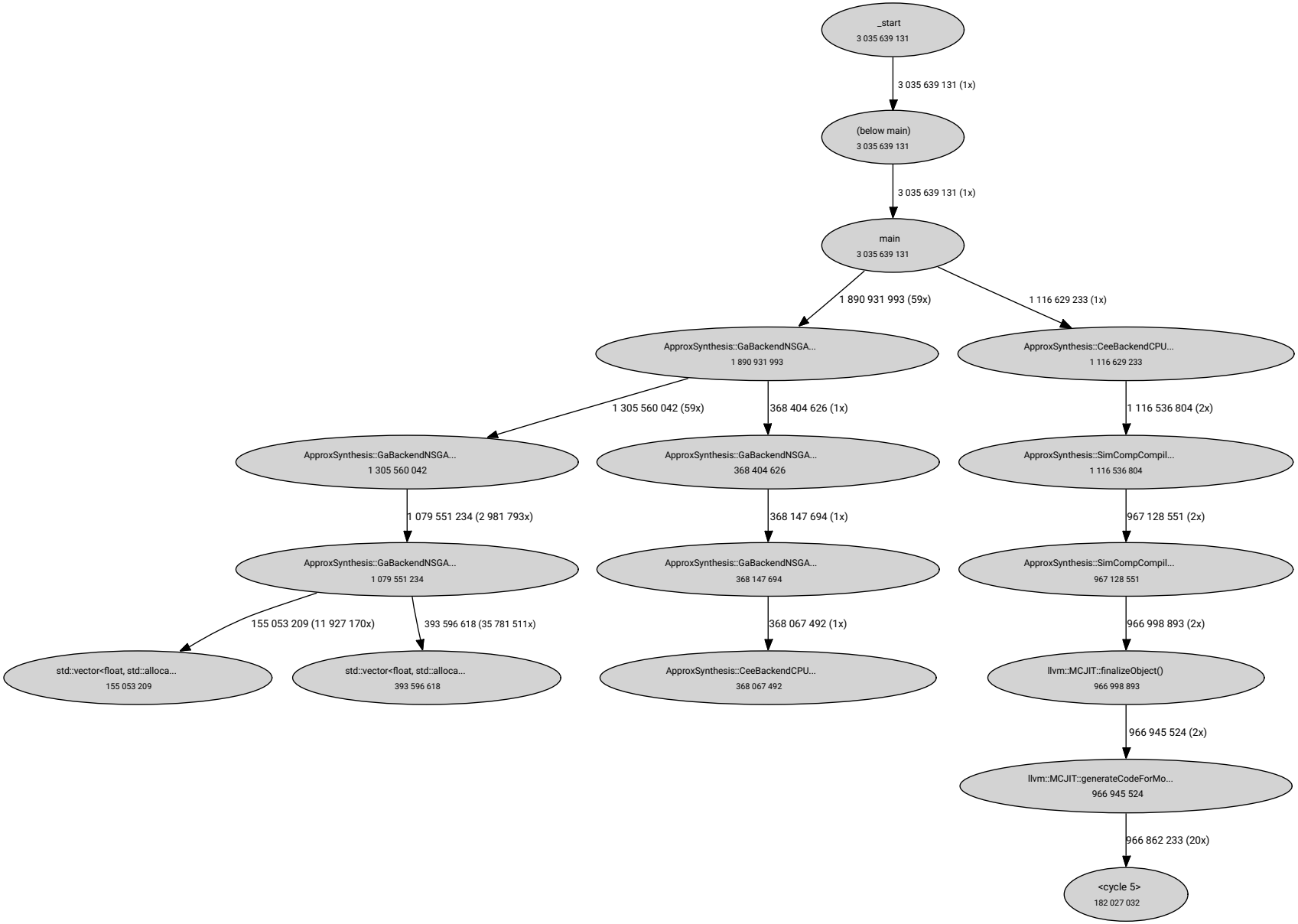
    ```
    > valgrind --tool=cachegrind <cesta-k-bin>
    ```

- Výstup (soubor `callgrind.out.<PID>`) lze zobrazit nástrojem KCachegrind
- Velkou nevýhodou je nízká rychlost simulace

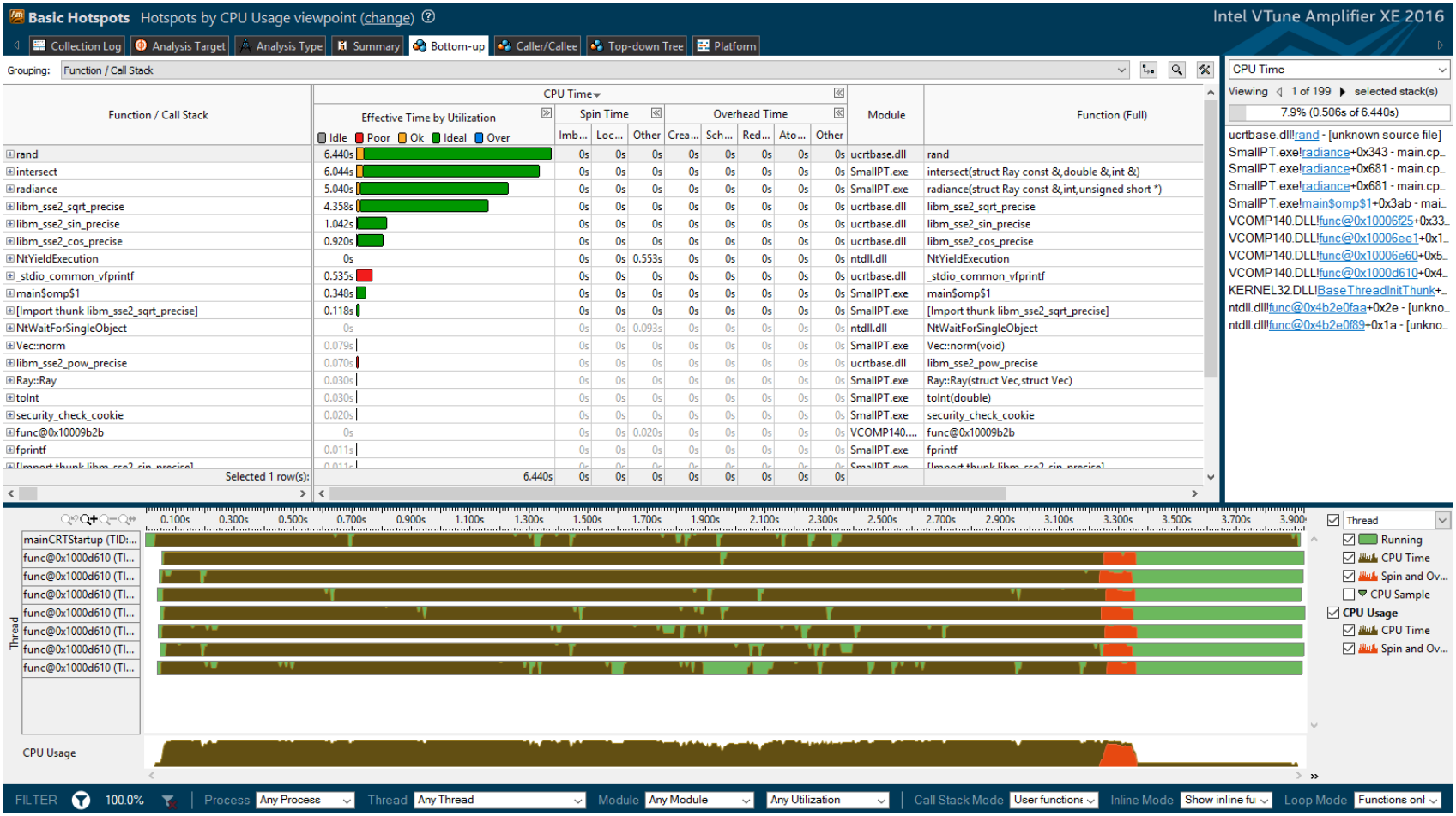
Profiling – KCachegrind



Profiling – KCachegrind

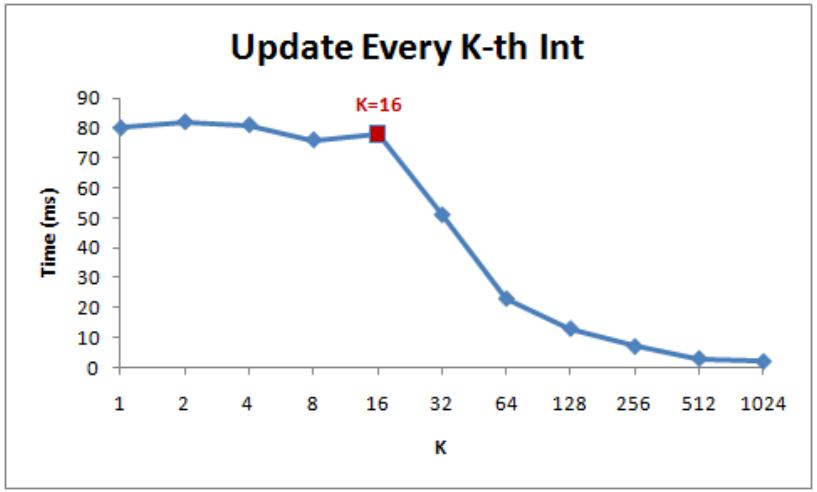


Profiling – Intel VTune



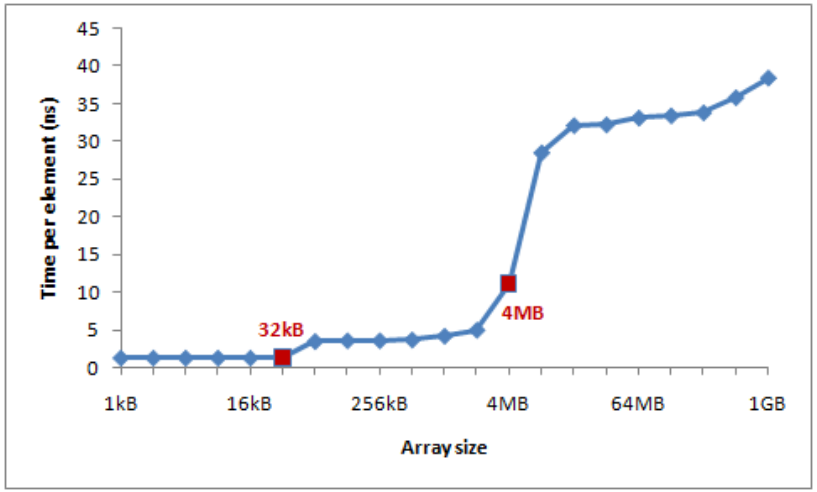
- Přístup do paměti (a tedy efektivní využití cache) významě ovlivňuje rychlost programu
- V profilovacích nástrojích obvykle vyjádřeno jako "Cache-Hit" a "Cache-Miss"
- Vliv modifikace každé K-té hodnoty:

```
for(int i = 0; i < LENGTH; i += K)
    arr[i] *= 3
```



- Přístup do paměti (a tedy efektivní využití cache) významě ovlivňuje rychlost programu
- V profilovacích nástrojích obvykle vyjádřeno jako "Cache-Hit" a "Cache-Miss"
- Vliv velikosti paměti cache:

```
for(int i = 0; i < STEPS; i++)
    arr[i % (SIZE - 1)]++;
```



Optimalizace – vektorizace

- Umožňuje využít vektorové instrukce CPU (SIMD)
  - MMX, SSE, SSE2, AVX, AVX2, AVX-512, ...
- Často ji dokáže provést překladač (ICC, GCC, ...)
- V ostatních případech je nutné "napovědět" překladači pomocí direktiv
  - Nejlépe pomocí OpenMP 4.0+ (`#pragma omp simd ...`)
  - Nebo specializovaných pro daný překladač
  - Je třeba dávat pozor na závislosti v datech
- Příklad direktiv OpenMP
  - Sečtení polí "a" a "b"

```
for(int i = 0; i < N; ++i)
    c[i] = a[i] + b[i];
```

  - Suma pole "arr"

```
for(int i = 0; i < N; ++i)
    s += arr[i];
```

Optimalizace – vektorizace

- Umožňuje využít vektorové instrukce CPU (SIMD)
  - MMX, SSE, SSE2, AVX, AVX2, AVX-512, ...
- Často ji dokáže provést překladač (ICC, GCC, ...)
- V ostatních případech je nutné "napovědět" překladači pomocí direktiv
  - Nejlépe pomocí OpenMP 4.0+ (`#pragma omp simd ...`)
  - Nebo specializovaných pro daný překladač
  - Je třeba dávat pozor na závislosti v datech
- Příklad direktiv OpenMP
  - Sečtení polí "a" a "b"

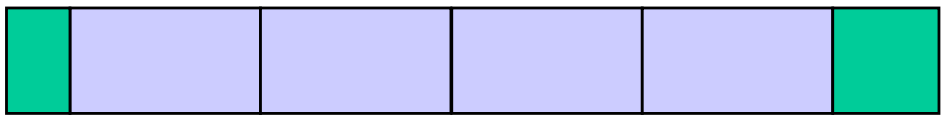
```
#pragma omp simd aligned(a,b,c:16)
for(int i = 0; i < N; ++i)
    c[i] = a[i] + b[i];
```

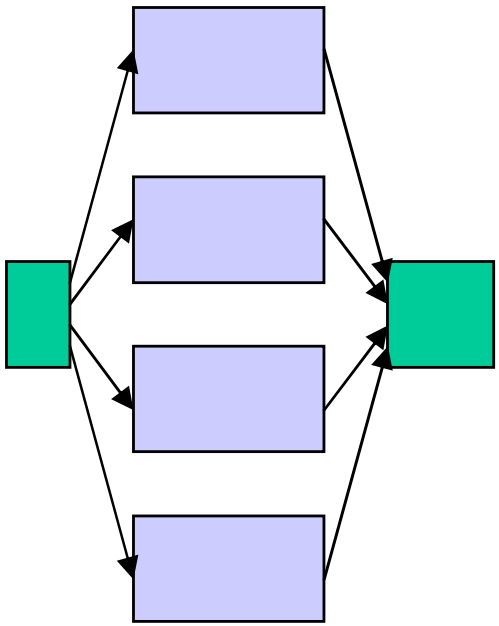
  - Suma pole "arr"

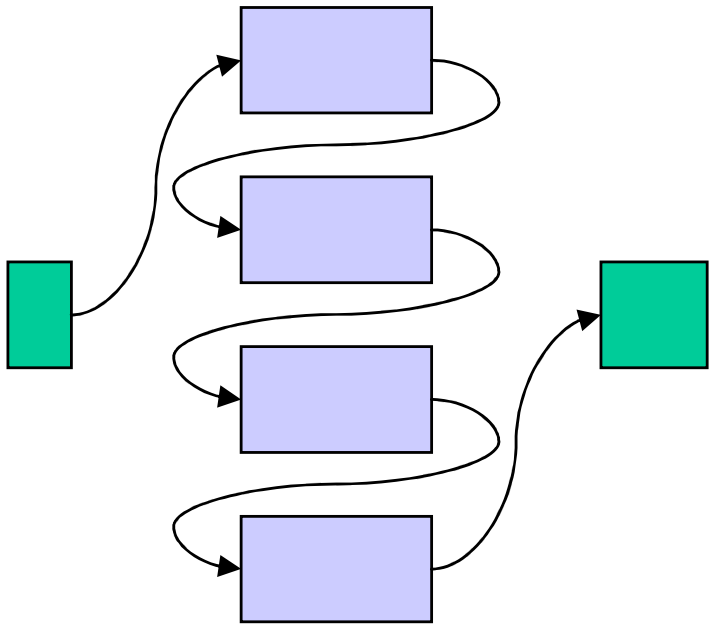
```
#pragma omp simd reduction(+:s)
for(int i = 0; i < N; ++i)
    s += arr[i];
```

Paralelizace

- Rozdělení výpočtu na více částí
- Minimalizace závislostí a komunikace mezi částmi
- Výpočet nad každou částí zvlášť

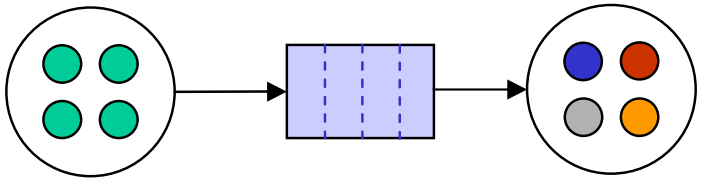


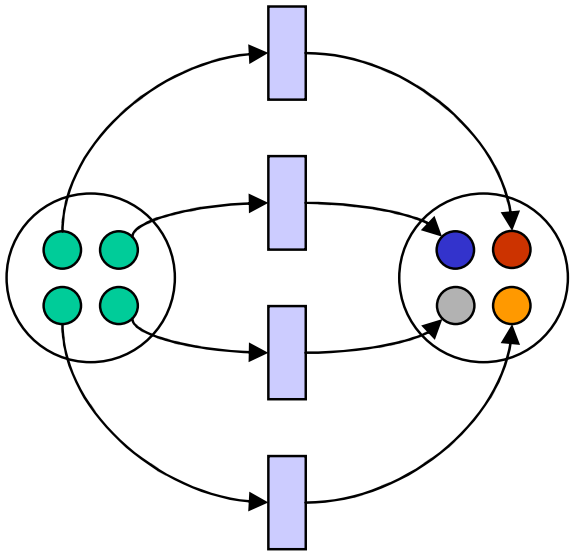




Paralelizace – datový paralelismus

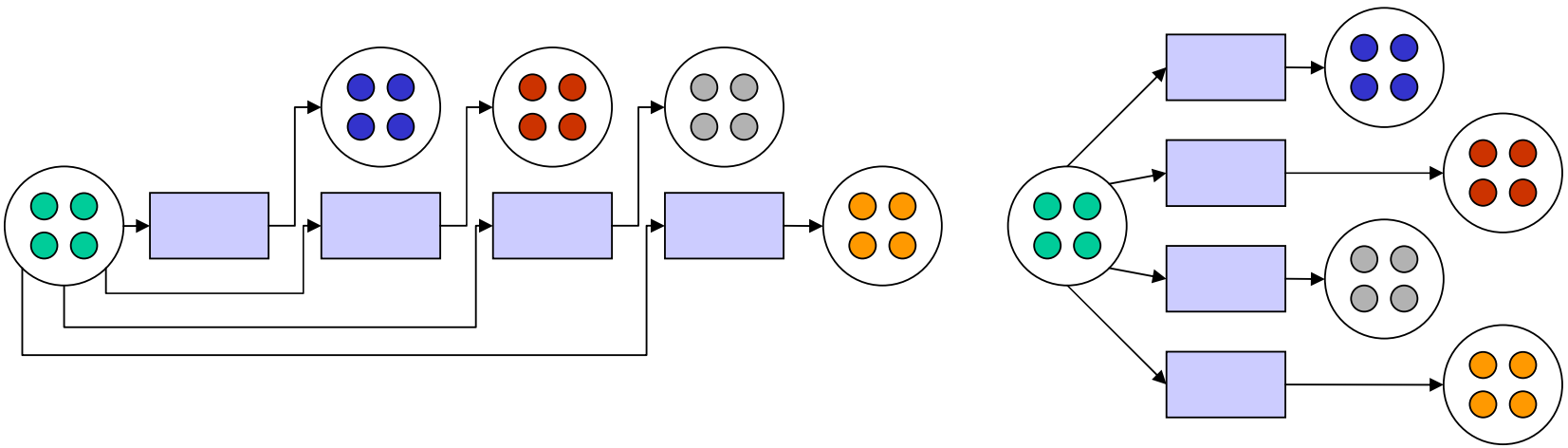
- Vykonání stejné operace nad velkým množstvím dat
- Data lze snadno rozdělit mezi vlákna a zpracovávat paralelně
- Paralelizace omezená množstvím dat





Paralelizace – úlohový paralelismus

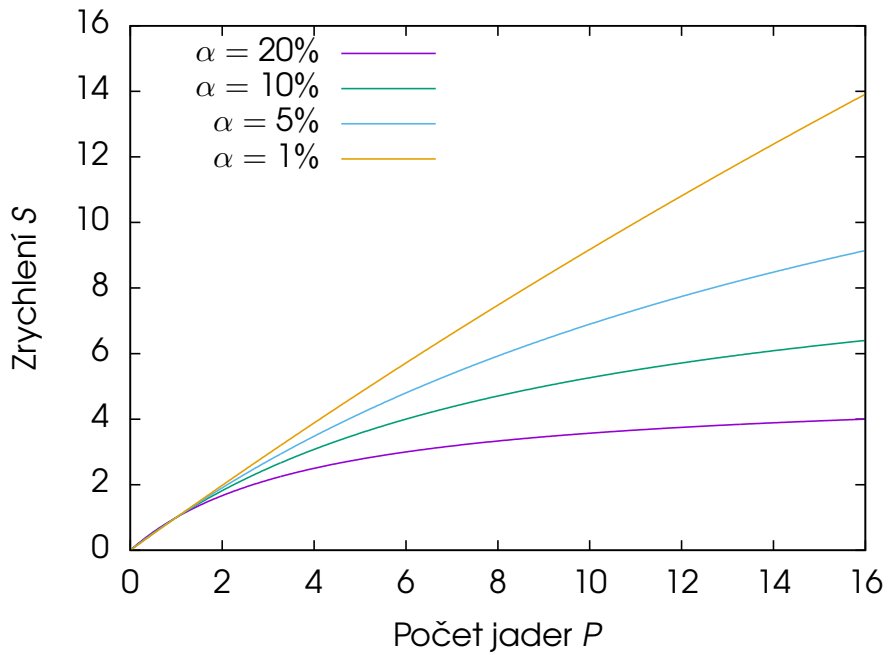
- Vykonání několika různých operací nad nějakou množinou dat (mohou ale nemusí být sdílené)
- Každé vlákno může vykonávat jinou úlohu
- Paralelizace omezená počtem nezávislých úloh



- Amdahlův zákon umožňuje určit maximální dosažitelné zrychlení S úlohy, pro kterou platí:
  - Její část αW lze provádět jedině sekvenčně (1 CPU)
  - Část $(1-\alpha)W$ lze provádět paralelně (P CPU)

Amdahlův zákon

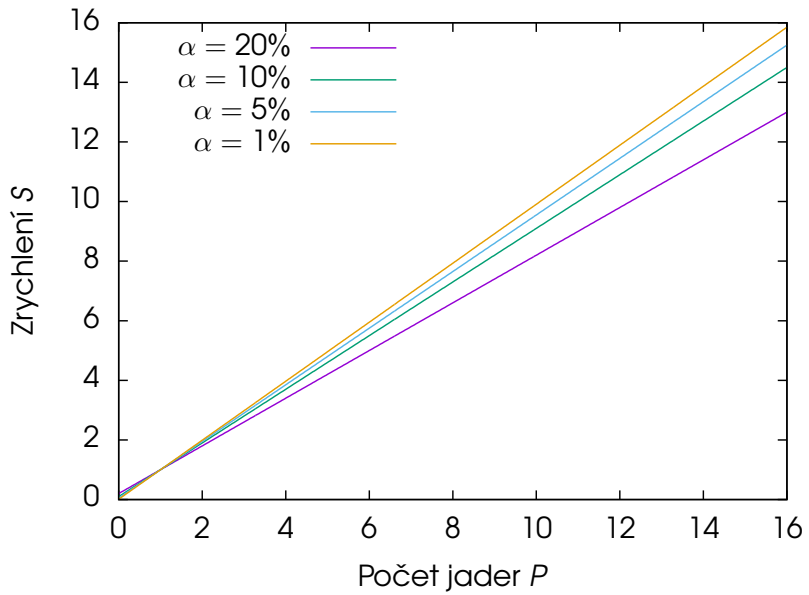
$$S(P) = \frac{T_s}{T(P)} = \frac{W/R}{W(\frac{\alpha}{R} + \frac{1-\alpha}{PR})} = \frac{P}{1+\alpha(P-1)}$$



- Amdahlův zákon předpokládá pevnou velikost řešené úlohy, maximální zrychlení je tedy limitováno sekvenční částí
- Gustafsonův zákon předpokládá, že velikost úlohy roste (lineárně) s počtem CPU
  - α_G udává část doby, kdy pracoval pouze 1 CPU

Gustafsonův zákon

$$S(P) = P - \alpha_G(P - 1)$$



Úrovně paralelizace

- Vlákna
  - Primitiva operačního systému reprezentující 1 tok instrukcí
  - Uvnitř 1 procesu, sdílejí paměťový prostor
  - OpenMP
- Procesy
  - Primitiva operačního systému
  - Odělený paměťový prostor s možností využít sdílenou paměť
  - Rychlost vytvoření a přepínání procesů je nižší než v případě vláken
  - OpenMPI (komunikace přes sdílenou paměť)
- Počítače
  - Procesy běžící na fyzicky odělených strojích
  - Fyzicky odělené paměti (nelze efektivně sdílet)
  - OpenMPI (komunikace přes síť)

- Standardizované rozšíření jazyků C/C++ o direktivy pro práci s vláknovým paralelismem
  - Podpora pro datový i úlohový paralelismus
  - Definuje chování proměnných ve vztahu k vláknům
  - Definuje synchronizační a atomická primitiva
  - Standard tvořen direktivami tvaru `#pragma omp ...` a funkcemi v hlavičkovém souboru "omp.h"

- Standardizované rozšíření jazyků C/C++ o direktivy pro práci s vláknovým paralelismem
  - Podpora pro datový i úlohový paralelismus
  - Definuje chování proměnných ve vztahu k vláknům
  - Definuje synchronizační a atomická primitiva
  - Standard tvořen direktivami tvaru `#pragma omp ...` a funkcemi v hlavičkovém souboru "omp.h"
- Paralelní region

```
#pragma omp parallel
{
    // Tento kód vykoná N vláken
}
```

- Standardizované rozšíření jazyků C/C++ o direktivy pro práci s vláknovým paralelismem
  - Podpora pro datový i úlohový paralelismus
  - Definuje chování proměnných ve vztahu k vláknům
  - Definuje synchronizační a atomická primitiva
  - Standard tvořen direktivami tvaru `#pragma omp ...` a funkcemi v hlavičkovém souboru "omp.h"
- Jednoduchá paralelizace smyčky

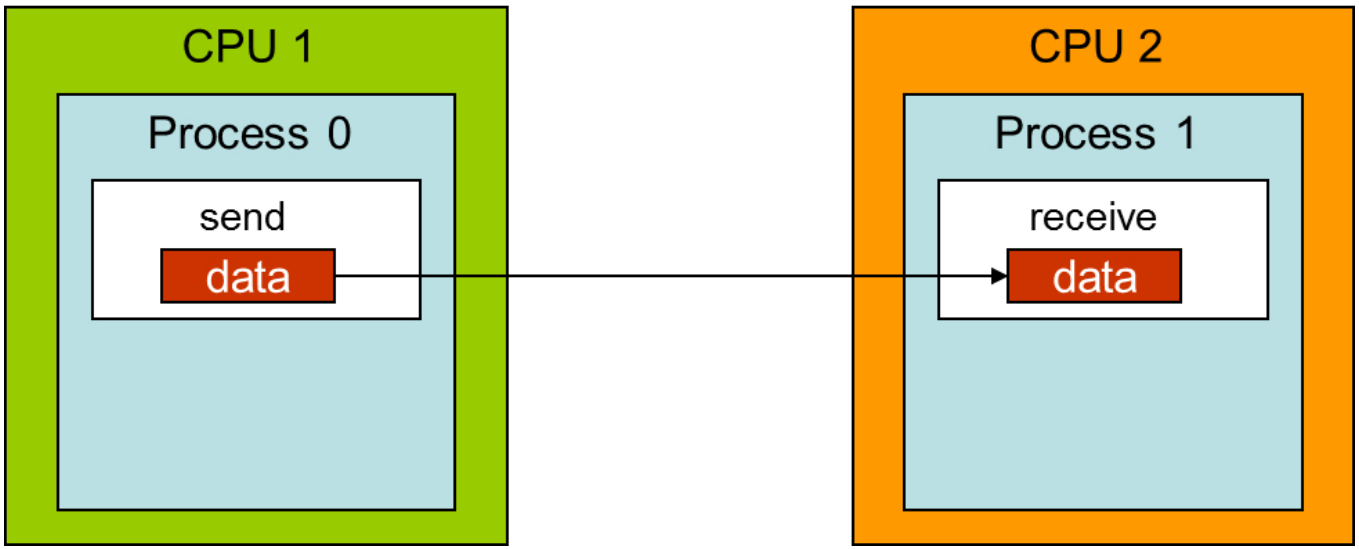
```
#pragma omp parallel for
for(int i = 0; i < N; ++i)
    c[i] = a[i] + b[i];
```

- Standardizované rozšíření jazyků C/C++ o direktivy pro práci s vláknovým paralelismem
  - Podpora pro datový i úlohový paralelismus
  - Definuje chování proměnných ve vztahu k vláknům
  - Definuje synchronizační a atomická primitiva
  - Standard tvořen direktivami tvaru `#pragma omp ...` a funkcemi v hlavičkovém souboru "omp.h"
- Úlohový paralelismus

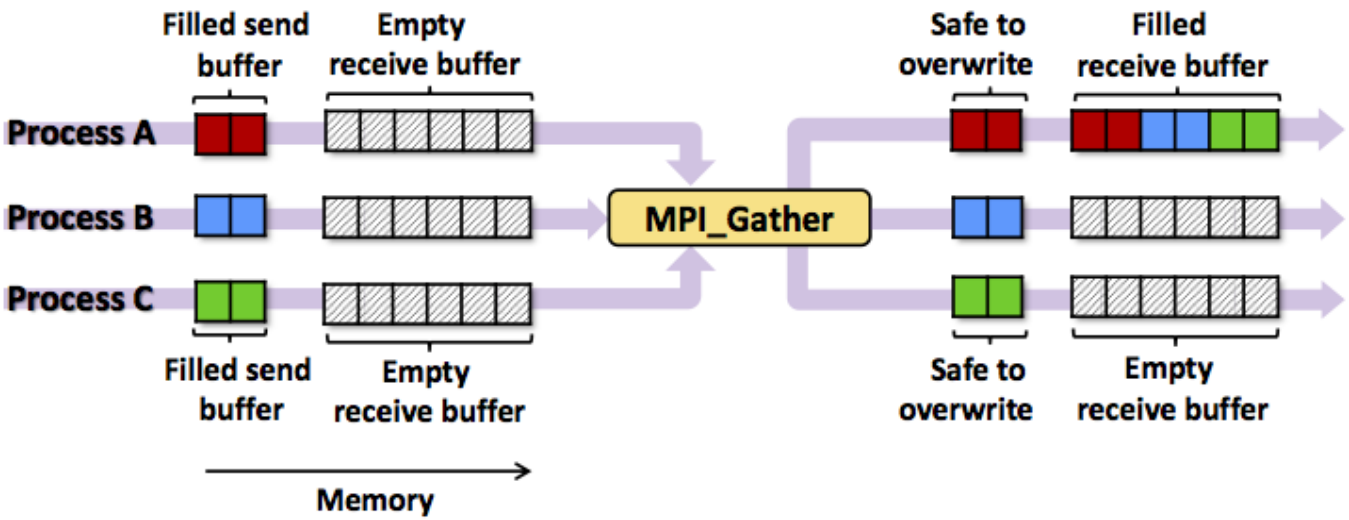
```
#pragma omp parallel sections
{
    #pragma omp section
    { // Tento kód vykoná vlákno A
    }

    #pragma omp section
    { // Tento kód vykoná vlákno B
    }
}
```

- Knihovna se standardizovaným rozhraním pro komunikaci a synchronizaci procesů (s odděleným paměťovým prostorem)
  - Model komunikace založený na zasílání zpráv
  - Komunikace přes sdílenou paměť, nebo síť

- Knihovna se standardizovaným rozhraním pro komunikaci a synchronizaci procesů (s odděleným paměťovým prostorem)
  - Model komunikace založený na zasílání zpráv
  - Komunikace přes sdílenou paměť, nebo síť



- Knihovna se standardizovaným rozhraním pro komunikaci a synchronizaci procesů (s odděleným paměťovým prostorem)
  - Model komunikace založený na zasílání zpráv
  - Komunikace přes sdílenou paměť, nebo síť



ivaverka@fit.vutbr.cz