

Paralelní a distribuované výpočty (B4B36PDV)

Branislav Bošanský, Michal Jakob

bosansky@fel.cvut.cz

Artificial Intelligence Center
Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague

Paralelní a Distribuované Výpočty

Paralelní programování

- 1 program
- vícero úloh, které spolupracují pro vyřešení problému
- typicky vlákna, sdílená paměť



Rychleji nalézt řešení

Programování v distribuovaných systémech

- vícero programů
- programy spolupracují pro nalezení řešení
- typicky procesy, výpočetní uzly, distribuovaná paměť



Zvýšit robustnost řešení

Kdo jsme

Přednášející



Branislav Božanský



Michal Jakob

Cvičící



Karel Horák



Peter Macejko



Petr Tomášek



Jan Mrkos

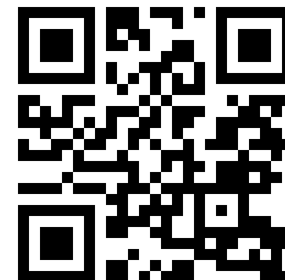


Daniel Fišer

Organizace a přehled

- Paralelní část
 - Paralelní programování jednoduchých algoritmů
 - Vliv různých způsobů paralelizace na rychlost výpočtu
- Distribuovaná část
 - Problémy v distribuovaných systémech (shoda, konzistence dat)
 - Navržení robustních řešení
- CourseWare
 - <https://cw.fel.cvut.cz/wiki/courses/b4b36pdv/start>
- Quizzes
 - <https://goo.gl/a6BEMb>

Bookmark
This Page



Hodnocení

- Domácí úkoly (40%)
 - Malé domácí úkoly (7x)
 - Velké domácí úkoly (2x)
- Praktický test z paralelního programování (20%)
- Teoretický test (40%)

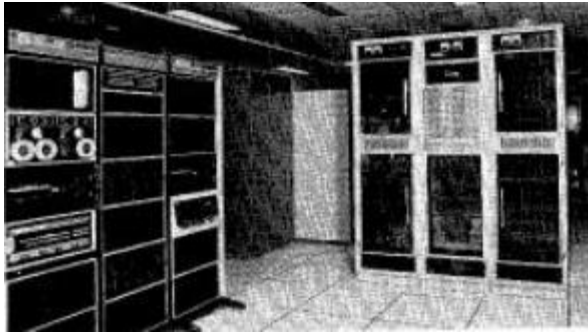
Pro úspěšné ukončení musíte získat alespoň 50% z každé části

Co udělat pro úspěšné zvládnutí PDV?

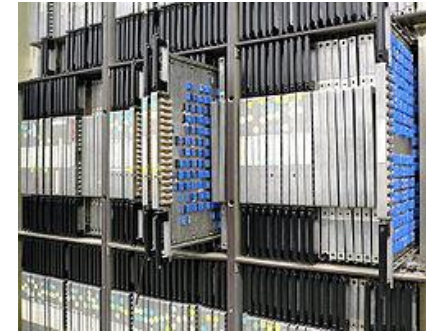
- programovat
 - zkoušejte si kódy z přednášek, upravujte je, analyzujte co se stane
 - nechte si čas na vypracování domácích úkolů
- přemýšlet
 - paralelní / distribuované programy se špatně ladí
 - vícevláknové chyby v debug-módu neodhalíte (můžou pomoci ladící výpisy)
 - pokud program nepracuje jak očekáváte (např. není dostatečně rychlý, výsledek není správný), zastavte se a zamyslete se proč tomu tak je

Krátká historie paralelních výpočtů

1970s-1980s – vědecké výpočty



C.mmp (1971)
16 CPUs



ILLIAC IV
(1975) 64
FPUs, 1 CPUs



Cray X-MP
(1982) parallel
vector CPU

Obrázky převzaty z:

- https://en.wikipedia.org/wiki/ILLIAC_IV
- https://en.wikipedia.org/wiki/Cray_X-MP

Krátká historie paralelních výpočtů

1990s – databáze, superpočítače



Sun Starfire
10000 (1997) 64
UltraSPARC CPUs



Cray T3E
(1996) až 1480
PE

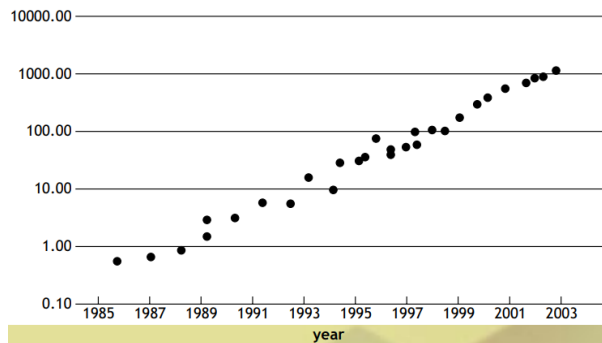
Obrázky převzaty z:

- https://en.wikipedia.org/wiki/Sun_Enterprise
- https://en.wikipedia.org/wiki/Cray_T3E

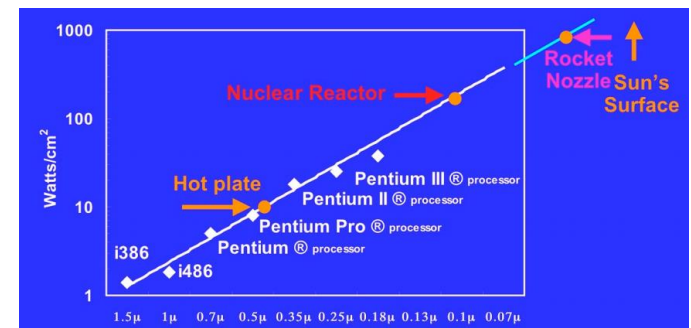
Krátká historie paralelních výpočtů

Nárůst výkonu jednoho procesoru

- Dlouhé roky rostl výpočetní výkon exponenciálně ...



- ... až v letech 2004 nebylo možné dále pouze navyšovat frekvenci



Obrázky převzaty z:

- Olukutun and Hammond, ACM Queue 2005
- <http://www.cs.columbia.edu/~sedwards/classes/2012/3827-spring/advanced-arch-2011.pdf>

Krátká historie paralelních výpočtů

Nárůst výkonu jednoho procesoru

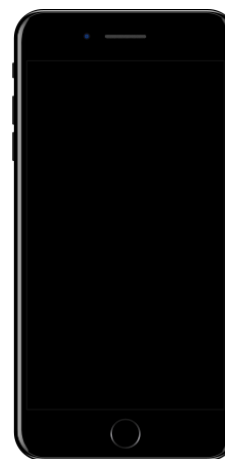
- Pohled programátora:

Jak zrychlím svůj program/algoritmus?

- Odpověď před 2004:
 - počkejte půl roku a kupte nový HW
- Odpověď po 2004:
 - přepsat na paralelní verzi

Krátká historie paralelních výpočtů

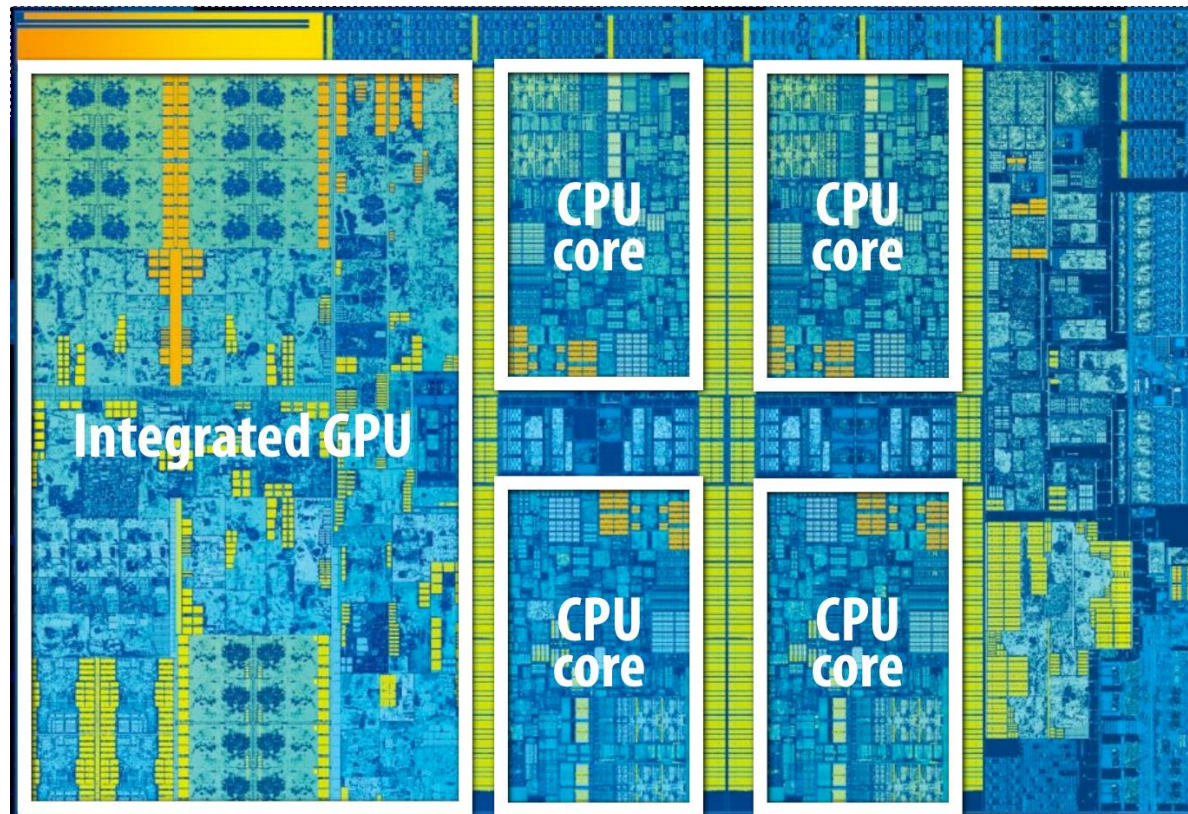
Dnešní paralelní stroje



Krátká historie paralelních výpočtů

Struktura dnešních CPU

- Intel Skylake (2015)
 - i7 – čtyřjádrové CPU + vícejádrové GPU



Krátká historie paralelních výpočtů

Alternativní architektury – Intel Xeon Phi

- Intel Xeon Phi
 - x86-64 architektura (61 CPUs, 244 vláken)



Krátká historie paralelních výpočtů

Alternativní architektury – GPU

- NVIDIA GPUs Pascal (GTX 1070)
 - 128 single-precision ALUs



Krátká historie paralelních výpočtů

Výpočetní gridy – TOP 10 superpočítačů

| Rank | System | Cores | Rmax (TFlop/s) | Rpeak (TFlop/s) | Power (kW) |
|------|--|------------|-------------------|--------------------|---------------|
| 1 | Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi China | 10,649,600 | 93,014.6 | 125,435.9 | 15,371 |
| 2 | Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 3151P , NUDT National Super Computer Center in Guangzhou China | 3,120,000 | 33,862.7 | 54,902.4 | 17,808 |
| 3 | Piz Daint - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 , Cray Inc. Swiss National Supercomputing Centre (CSCS) Switzerland | 361,760 | 19,590.0 | 25,326.3 | 2,272 |
| 4 | Gyokkou - ZettaScaler-2.2 HPC system, Xeon D-1571 16C 1.3GHz, Infiniband EDR, PEZY-SC2 700Mhz , ExaScaler Japan Agency for Marine-Earth Science and Technology Japan | 19,860,000 | 19,135.8 | 28,192.0 | 1,350 |
| 5 | Titan - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x , Cray Inc. DOE/SC/Oak Ridge National Laboratory United States | 560,640 | 17,590.0 | 27,112.5 | 8,209 |
| 6 | Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom , IBM DOE/NNSA/LLNL United States | 1,572,864 | 17,173.2 | 20,132.7 | 7,890 |
| 7 | Trinity - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect , Cray Inc. DOE/NNSA/LANL/SNL United States | 979,968 | 14,137.3 | 43,902.6 | 3,844 |
| 8 | Cori - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect , Cray Inc. DOE/SC/LBNL/NERSC United States | 622,336 | 14,014.7 | 27,880.7 | 3,939 |
| 9 | Oakforest-PACS - PRIMERGY CX1640 M1, Intel Xeon Phi 7250 68C 1.4GHz, Intel Omni-Path , Fujitsu Joint Center for Advanced High Performance Computing Japan | 556,104 | 13,554.6 | 24,913.5 | 2,719 |
| 10 | K computer , SPARC64 VIIIfx 2.0GHz, Tofu interconnect , Fujitsu RIKEN Advanced Institute for Computational Science (AICS) Japan | 705,024 | 10,510.0 | 11,280.4 | 12,660 |

Krátká historie paralelních výpočtů

Výpočetní gridy – a co u nás?

- IT4Innovations (www.it4i.cz)
 - 180x 16 Core CPUs, 23x Kepler GPUs, 4x Xeon Phi
 - 1008x 2x12 Core CPUs
 - komerční výpočty, lze zažádat a získat výpočetní čas pro výzkum
- Metacentrum
 - spojení výpočetních prostředků akademické sítě
 - volně dostupné pro akademické pracovníky, studenty
 - mnoho dostupných strojů (CPU, GPU, Xeon Phi)
 - <https://metavo.metacentrum.cz/pbsmon2/hardware>

Vliv architektury

Cache

- Proč je důležité vědět o architektuře?
 - Uvažme příklad násobení matice vektorem

```
int x[MAXIMUM], int y[MAXIMUM], int A[MAXIMUM*MAXIMUM]
```

Varianta A

```
for ( int i = 0; i < MAXIMUM ; i ++)  
  for ( int j = 0; j < MAXIMUM ; j ++)  
    y[i] += A->at(i * MAXIMUM + j)*x[j];
```

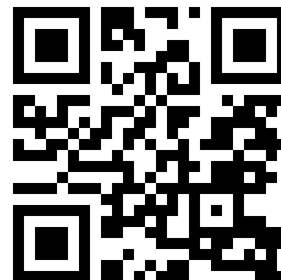
Varianta B

```
for ( int j = 0; j < MAXIMUM ; j ++)  
  for ( int i = 0; i < MAXIMUM ; i ++)  
    y [i] += A->at(i * MAXIMUM + j)*x[j];
```

Který kód bude rychlejší?



<https://goo.gl/a6BEMb>



Vliv architektury

Cache

```
for ( int i = 0; i < MAXIMUM ; i ++)  
  for ( int j = 0; j < MAXIMUM ; j ++)  
    y[i] += A->at(i * MAXIMUM + j)*x[j];
```



```
for ( int j = 0; j < MAXIMUM ; j ++)  
  for ( int i = 0; i < MAXIMUM ; i ++)  
    y [i] += A->at(i * MAXIMUM + j)*x[j];
```



- Pole jsou v paměti uložena sekvenčně (po řádcích)
- CPU při přístupu k $A[0][0]$ načte do cache vícero hodnot (cache line)

| Cache Line | Elements of A | | | |
|------------|---------------|-----------|-----------|-----------|
| 0 | $A[0][0]$ | $A[0][1]$ | $A[0][2]$ | $A[0][3]$ |
| 1 | $A[1][0]$ | $A[1][1]$ | $A[1][2]$ | $A[1][3]$ |
| 2 | $A[2][0]$ | $A[2][1]$ | $A[2][2]$ | $A[2][3]$ |
| 3 | $A[3][0]$ | $A[3][1]$ | $A[3][2]$ | $A[3][3]$ |

- Při přístupu k $A[1][0]$ se změní celý řádek

V rámci paralelních programů může k podobným problémům docházet častěji

Paralelizace

Jednoduchý příklad

- Suma vektoru čísel

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | ... | ... | 5×10^9 |
|----|---|---|---|----|---|---|-----|-----|-----------------|
| 17 | 2 | 9 | 4 | 22 | 0 | 1 | | | 8 |

Jak paralelizovat?

- Mějme 4 jádra – každé jádro může sečíst čtvrtinu vektoru, pak sečteme částečné součty

| Vláken | 1 | 2 | 3 | 4 |
|--------|--------|--------|--------|--------|
| Čas | 0.389s | 0.262s | 0.258s | 0.244s |

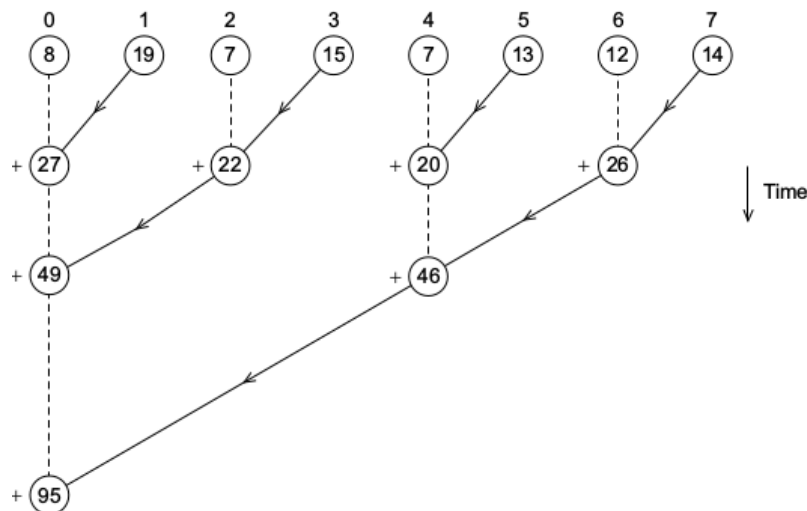
Paralelizace

Jednoduchý příklad

- Suma vektoru čísel

Co když máme tisíce jader?

- Pokud částečné součty sčítá pouze jedno jádro, kód není velmi efektivní



Hlavní cíl paralelní části

- Paralelizace úkolů / dat
 - Rozdělení úkolu na jiné součásti a jejich paralelizace
 - Rozdělení dat a jejich (téměř) stejné paralelní zpracování
 - Opravování písemky (rozdělení po otázkách/studentech)
- Komunikace a synchronizace mezi vlákny/procesy
 - Přístup ke společné paměti



Získat základní informace a prostor pro praktické zkušenosti v oblasti programování efektivních paralelních programů

Přehled paralelní části

- Základní úvod
 - Vlákna, synchronizace, mutexy
 - Pthread (již by jste měli znát), C++11 thready
- OpenMP
 - nadstavba nad C kompilátorem pro zjednodušení implementace paralelních programů
- Techniky dekompozice
- Datové struktury umožňující přístup vícero vláken
- Základní paralelní řadící algoritmy a vektorové instrukce
- Základní paralelní maticové algoritmy

Pthready vs. C++11 vs. OpenMP

Ochutnávka (pthreads)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

const int thread_count = 10;
void* Hello(void* rank);

int main(int argc, char* argv[]) {
    long thread;
    pthread_t *thread_handles;
    thread_handles = (pthread_t*)malloc(thread_count * sizeof(pthread_t));
    for (thread = 0; thread < thread_count; thread++)
        pthread_create(&thread_handles[thread], NULL,
                      Hello, (void *) thread);
    printf("Hello from the main thread\n");
    for (thread = 0; thread < thread_count; thread++)
        pthread_join(thread_handles[thread], NULL);

    free(thread_handles);
    return 0;
}

void* Hello(void* rank) {
    long my_rank = (long) rank;
    printf("Hello from thread %ld of %d\n", my_rank, thread_count);
    return NULL;
}
```

Pthready vs. C++11 vs. OpenMP

Ochutnávka (C++11)

```
#include <iostream>
#include <thread>
#include <vector>

const int thread_count = 10;
void Hello(long my_rank);

int main(int argc, char* argv[]) {
    std::vector<std::thread> threads;
    for (int thread=0; thread < thread_count; thread++) {
        threads.push_back(std::thread(Hello, thread));
    }

    std::cout << "Hello from the main thread\n";

    for (int thread=0; thread < thread_count; thread++) {
        threads[thread].join();
    }

    return 0;
}

void Hello(long my_rank) {
    std::cout << "Hello from thread " << my_rank << " of " << thread_count << std::endl;
}
```


Pthready vs. C++11 vs. OpenMP

Ochutnávka (OpenMP)

```
#include <iostream>
#include <vector>
#include "omp.h"

const int thread_count = 10;

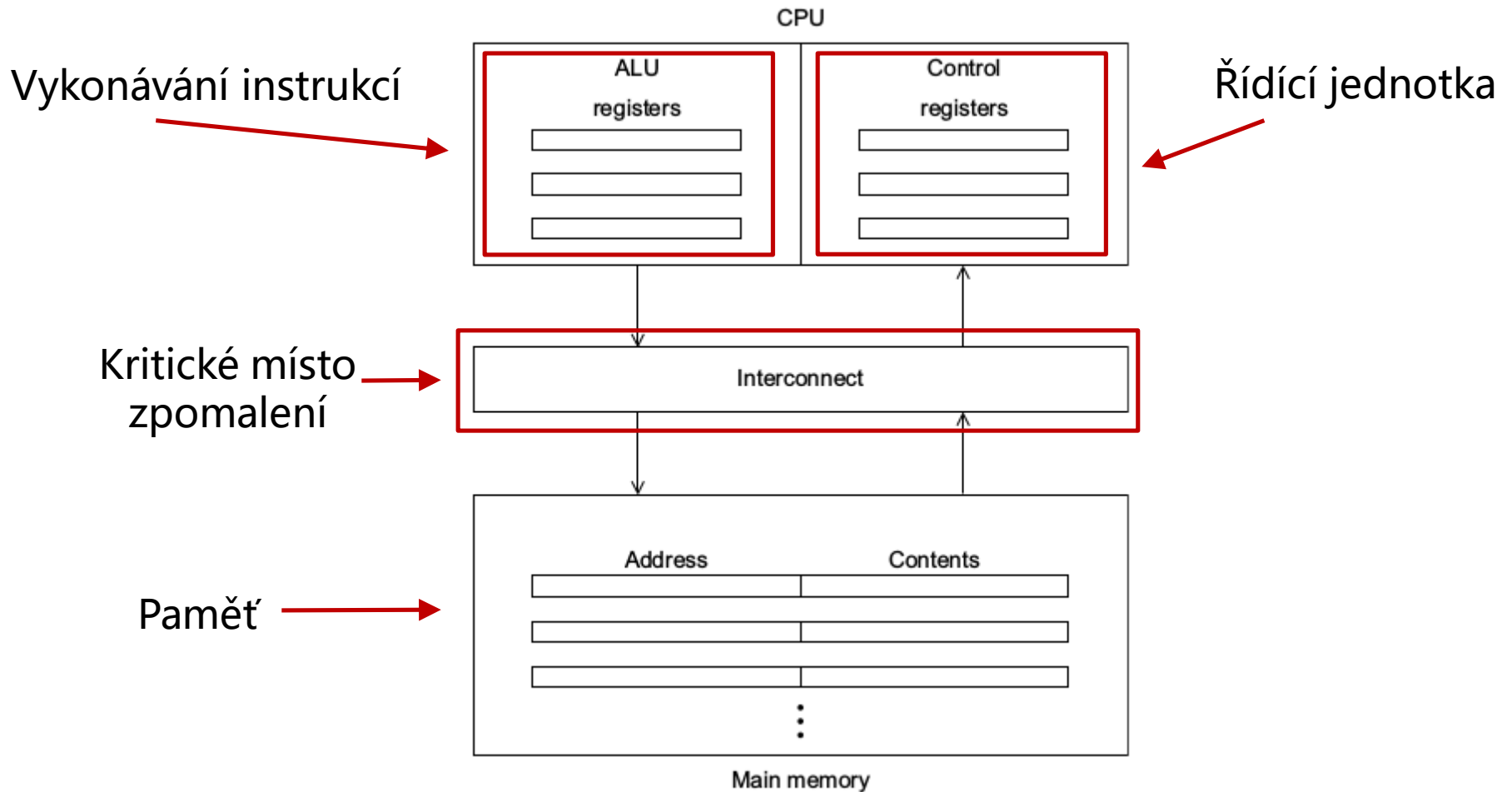
void Hello() {
    int my_rank = omp_get_thread_num();
    int threads = omp_get_num_threads();
    std::cout << "Hello from thread " << my_rank << " of " << threads << std::endl;
}

int main(int argc, char* argv[]) {
    #pragma omp parallel num_threads(thread_count)
    Hello();
    return 0;
}
```

- nutno překládat s přepínačem `-fopenmp`
 - (např. `g++ -fopenmp openmp-hello.cpp -o openmp-hello`)

Potřebný HW základ

Von Neumannova architektura



Potřebný HW základ

Pipelines

- Paralelizace na úrovni instrukcí (ILP)
- Příklad:
 - Chceme sečíst 2 vektory reálných čísel (float [1000])
 - 1 součet – 7 operací
 - Fetch
 - Porovnání exponentů
 - Posun
 - Součet
 - Normalizace
 - Zaokrouhlení
 - Uložení výsledku
 - Bez ILP – $7 \times 1000 \times (\text{čas 1 operace; 1 ns})$

Potřebný HW základ

Pipelines

- Paralelizace na úrovni instrukcí (ILP)
- Příklad:
 - Chceme sečíst 2 vektory reálných čísel (float [1000])
 - 1 součet – 7 operací
 - Bez ILP – 7x1000x(čas 1 operace; 1ns)
 - S ILP (a 7 jednotek) – 1005 ns

Table 2.3 Pipelined Addition. Numbers in the Table Are Subscripts of Operands/Results

| Time | Fetch | Compare | Shift | Add | Normalize | Round | Store |
|------|-------|---------|-------|-----|-----------|-------|-------|
| 0 | 0 | | | | | | |
| 1 | 1 | 0 | | | | | |
| 2 | 2 | 1 | 0 | | | | |
| 3 | 3 | 2 | 1 | 0 | | | |
| 4 | 4 | 3 | 2 | 1 | 0 | | |
| 5 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 6 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| 999 | 999 | 998 | 997 | 996 | 995 | 994 | 993 |
| 1000 | | 999 | 998 | 997 | 996 | 995 | 994 |
| 1001 | | | 999 | 998 | 997 | 996 | 995 |
| 1002 | | | | 999 | 998 | 997 | 996 |
| 1003 | | | | | 999 | 998 | 997 |
| 1004 | | | | | | 999 | 998 |
| 1005 | | | | | | | 999 |

Potřebný HW základ

Superskalární procesory

- Současné vyhodnocení vícero instrukcí
 - uvažme cyklus

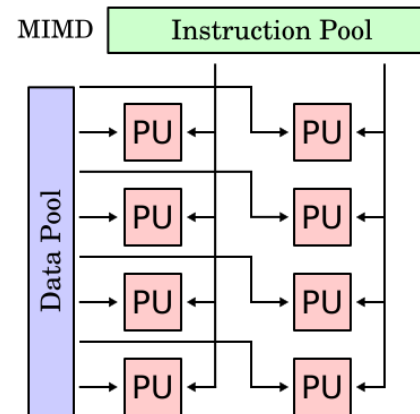
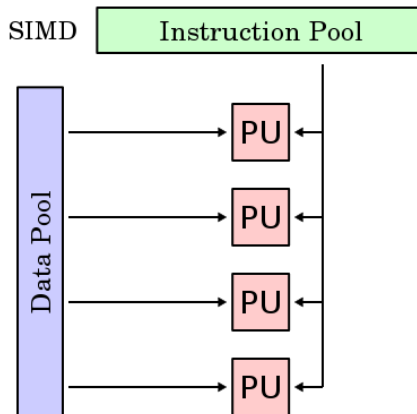
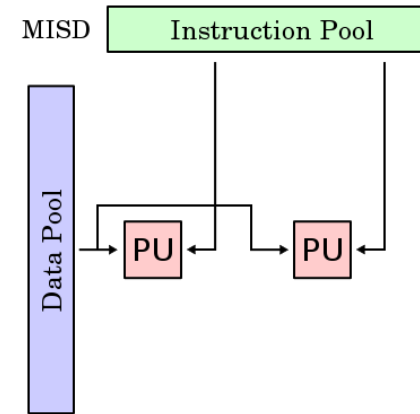
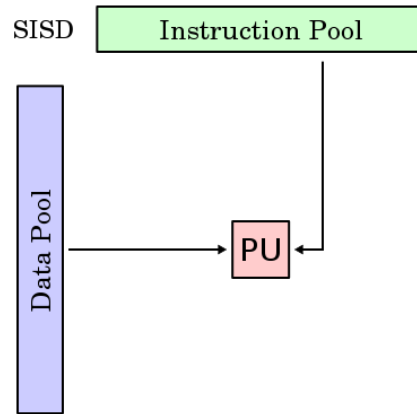
```
for (i=0; i<1000; i++)  
  z[i]=x[i]+y[i];
```

- jedna jednotka může počítat z[0], druhá z[1], ...
- Speklativní vyhodnocení

```
z = x + y;  
if (z > 0)  
  w = x;  
else  
  w = y;
```

Potřebný HW základ

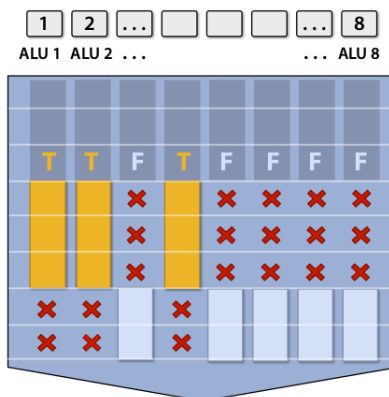
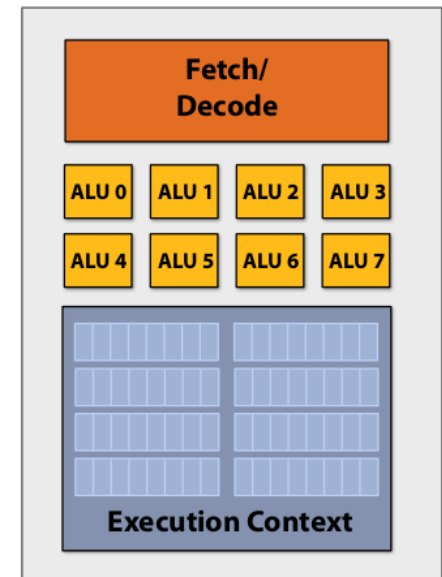
Paralelní hardware – Flynnova taxonomie



Potřebný HW základ

Paralelní hardware – Flynnova taxonomie

- SIMD (Single Instruction Multiple Data)
 - Jedna řídicí jednotka, vícero ALU jednotek
 - Datový paralelizmus
 - Vektorové procesory, GPU
 - Běžné jádra CPU podporují SIMD paralelizmus
 - instrukce SSE, AVX
- Větvení na SIMD

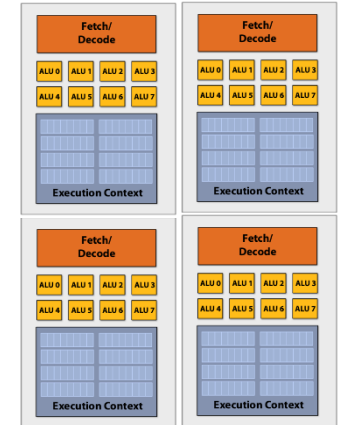


Potřebný HW základ

Paralelní hardware – Flynnova taxonomie

- MIMD (Multiple Instruction Multiple Data)
 - Více-jádrové procesory
 - Různá jádra vykonávají různé instrukce
 - Víceprocesorové počítače

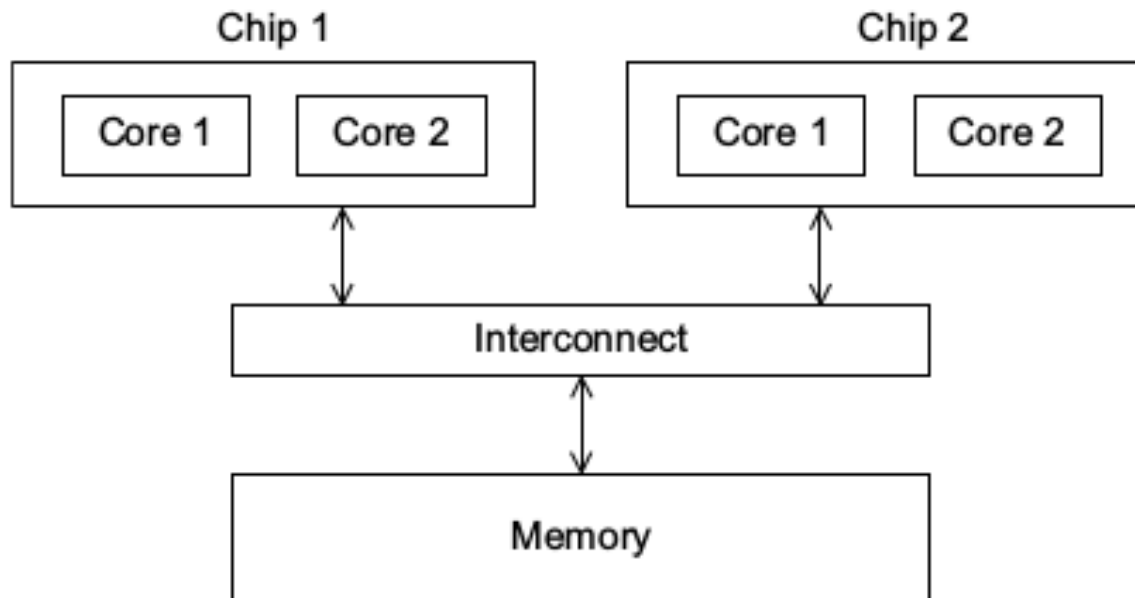
- A co paměť?



Potřebný HW základ

Systemy se sdílenou pamětí

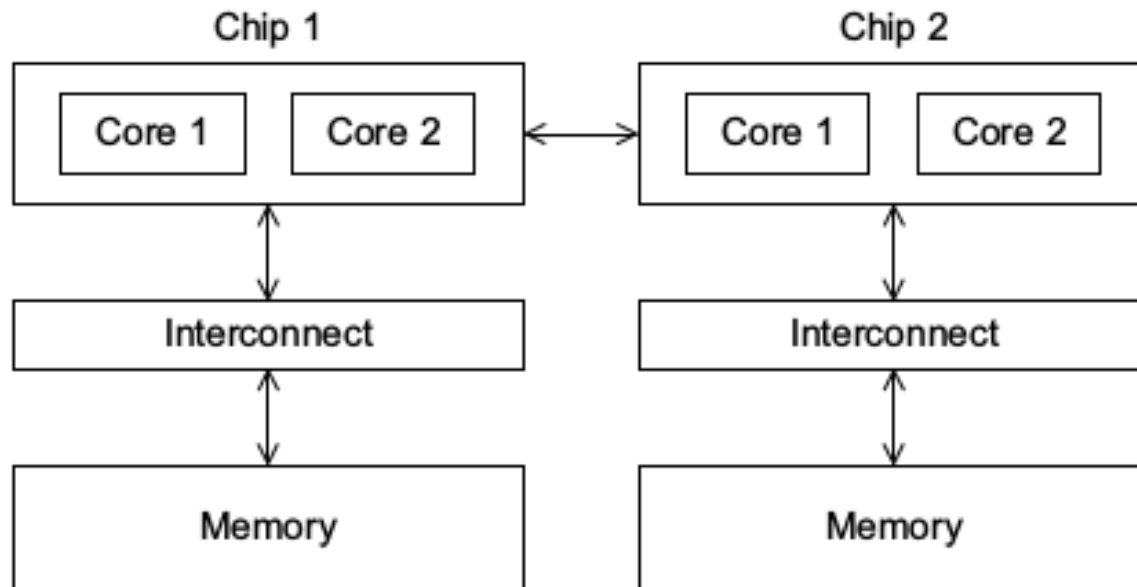
- Uniform Memory Access (UMA)



Potřebný HW základ

Systemy se sdílenou pamětí

- Nonuniform Memory Access (NUMA)

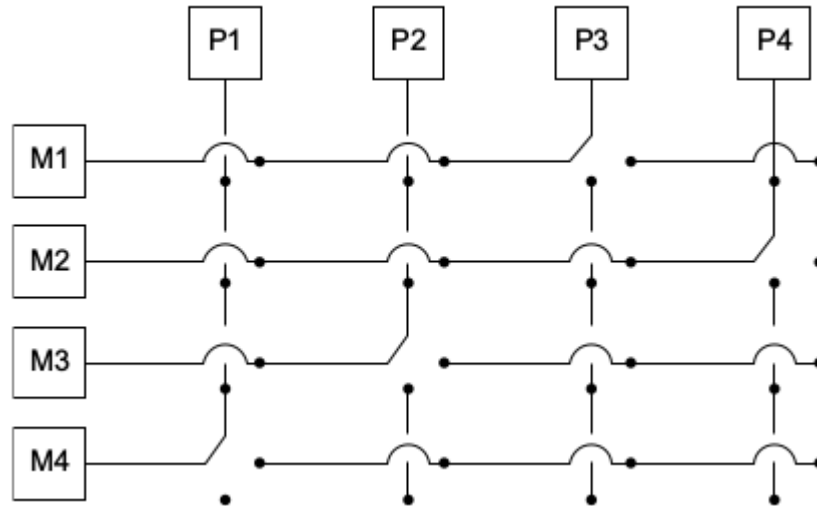


Potřebný HW základ

Systemy se sdílenou pamětí – typy přístupů k paměti

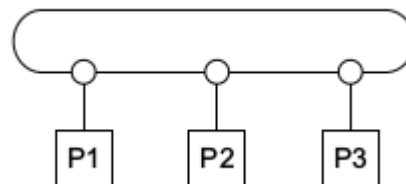
- Sběrnice

- Mřížka



- Síť/Kruh

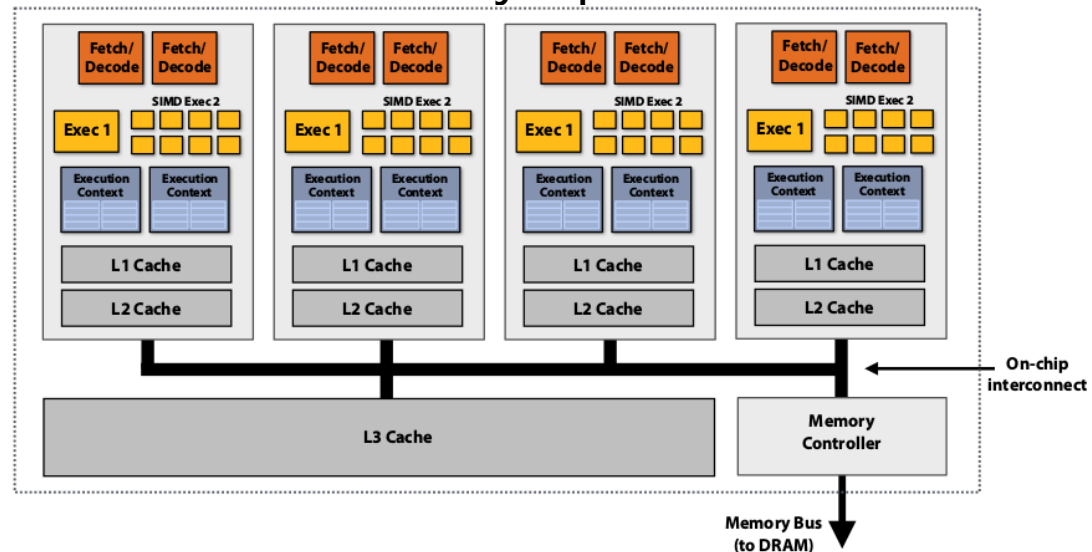
- ...



Potřebný HW základ

Základy Cache

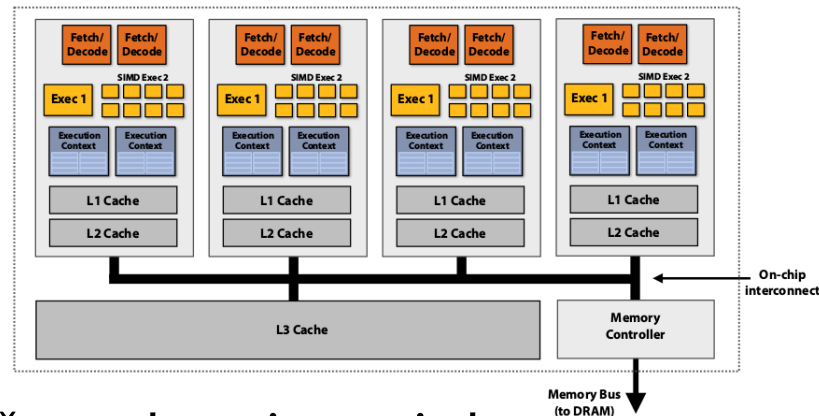
- CPU cache
 - Programy často přistupují k paměti lokálně (lokalita v prostoru a čase)
 - Cache se upravuje po řádcích
- Každé jádro má vlastní cache + existuje společná cache



A co když jádra přistupují ke stejné adrese?

Potřebný HW základ

Distributed memory



- Musíme udržovat konzistenci dat



V dnešních moderních CPU je nutno řešit řadu paralelních a distribuovaných problémů

Pokud budeme implementovat naše algoritmy bez ohledu na architekturu, zrychlení nemusí být dostatečné

Pokročilejší příklad

- Vraťme se k příkladu se sčítáním vektoru čísel
 - (teď budeme sčítat celou část druhých odmocnin)

sčítané pole

id vlákna

pole pro dílčí součty

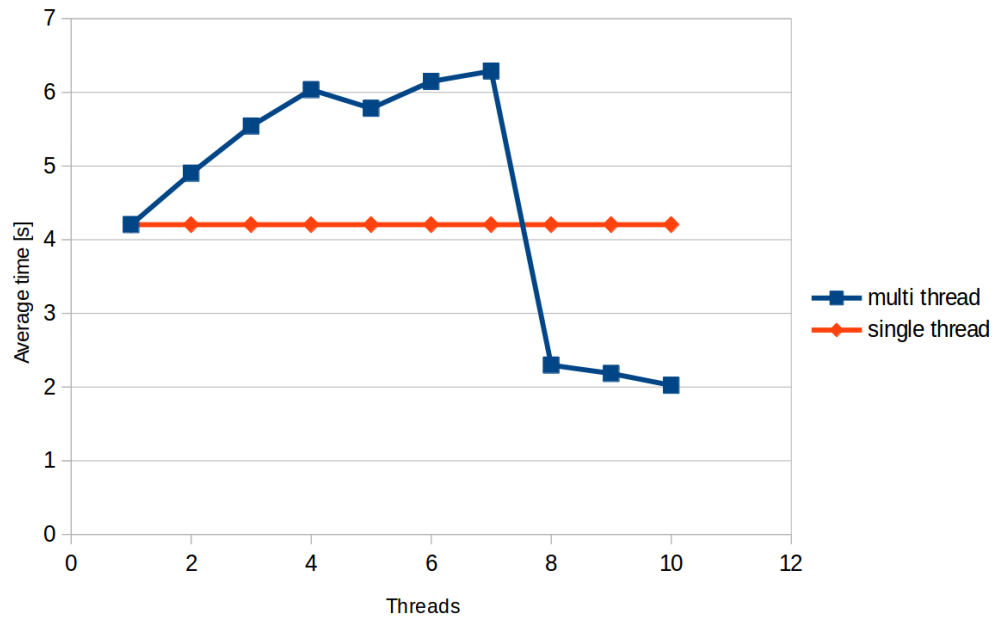
```
long sum(std::vector<int>& vector_to_sum, int thread, std::vector<long>& sums) {  
    for (int i=thread; i<SIZE; i += thread_count)  
        sums[thread] += sqrt(vector_to_sum[i]);  
  
    for (int j=1; j<log2(thread_count)+1; j++) {  
        if ((thread % (int)pow(2,j)) != 0) break;  
        int k = (int)pow(2,j-1);  
        if ((thread + k) >= thread_count) break;  
        if (threads[thread + k].joinable()) threads[thread + k].join();  
        sums[thread] += sums[thread + k];  
    }  
}
```

logaritmický
součet dílčích
výsledků

každé vlákno zapisuje na
vlastní index pole

Pokročilejší příklad

Jak nám to bude fungovat?




Nic moc :(

Pokročilejší příklad

Kde je chyba?

```
long sum(std::vector<int>& vector_to_sum, int thread, std::vector<long>& sums) {  
    for (int i=thread; i<SIZE; i += thread_count)  
        sums[thread] += sqrt(vector_to_sum[i]);  
  
    for (int j=1; j<log2(thread_count)+1; j++) {  
        if ((thread % (int)pow(2,j)) != 0) break;  
        int k = (int)pow(2,j-1);  
        if ((thread + k) >= thread_count) break;  
        if (threads[thread + k].joinable()) threads[thread + k].join();  
        sums[thread] += sums[thread + k];  
    }  
}
```

každé vlákno zapisuje na
vlastní index pole



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|----|---|---|---|----|---|---|---|---|---|
| 17 | 2 | 9 | 4 | 22 | 0 | 1 | 0 | 0 | 8 |

Pokročilejší příklad

Kde je chyba?

```
long sum(std::vector<int>& vector_to_sum, int thread, std::vector<long>& sums) {  
    for (int i=thread; i<SIZE; i += thread_count)  
        sums[thread] += sqrt(vector_to_sum[i]);  
  
    for (int j=1; j<log2(thread_count)+1; j++) {  
        if ((thread % (int)pow(2,j)) != 0) break;  
        int k = (int)pow(2,j-1);  
        if ((thread + k) >= thread_count) break;  
        if (threads[thread + k].joinable()) threads[thread + k].join();  
        sums[thread] += sums[thread + k];  
    }  
}
```

- vlákno 0 upraví hodnotu
- jenže vlákno 0 má celý vektor **sums** v cache jádra
- a podobně i jiné vlákna
- při změně 1 hodnoty se musí zabezpečit konzistence

| | | | | | | | | | |
|----|---|---|---|----|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 17 | 2 | 9 | 4 | 22 | 0 | 1 | 0 | 0 | 8 |

False Sharing

False Sharing

možné řešení

```
long sum_local(std::vector<int>& vector_to_sum, int thread, std::vector<long>& sums) {  
    long local = 0;  
    for (int i=thread; i<SIZE; i += thread_count) {  
        local += sqrt(vector_to_sum[i]);  
    }  
    sums[thread] = local;  
  
    for (int j=1; j<log2(thread_count)+1; j++) {  
        if ((thread % (int)pow(2,j)) != 0) break;  
        int k = (int)pow(2,j-1);  
        if ((thread + k) >= thread_count) break;  
        if (threads[thread + k].joinable()) threads[thread + k].join();  
        local += sums[thread + k];  
    }  
    sums[thread] = local;  
}
```

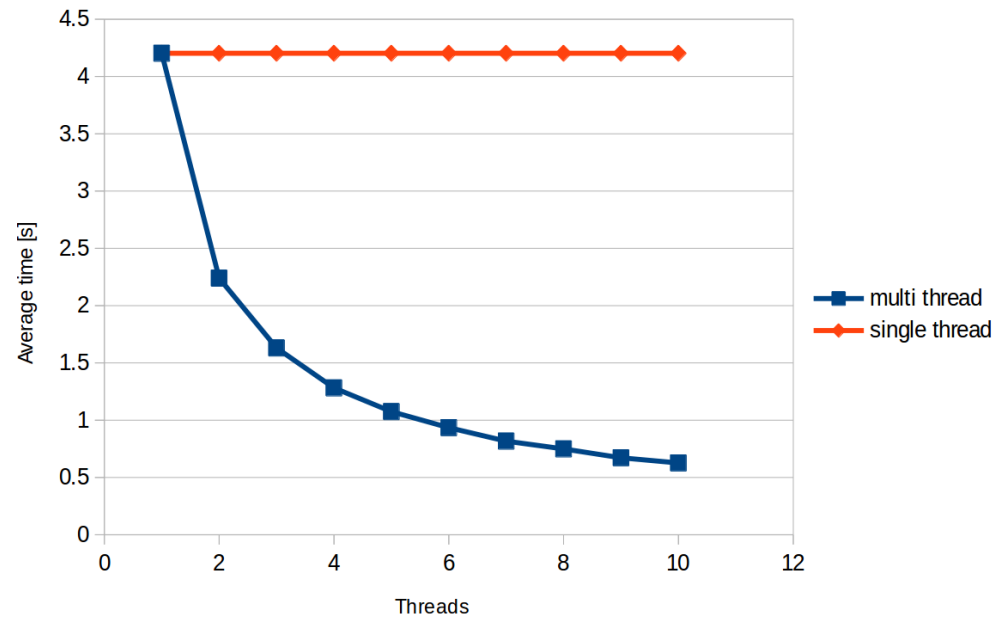
každé vlákno zapisuje
do lokální proměnné

pouze finální výsledek
se zapíše do vektoru

Potřebný HW základ

False Sharing

lokální proměnná – opravdu to pomůže?



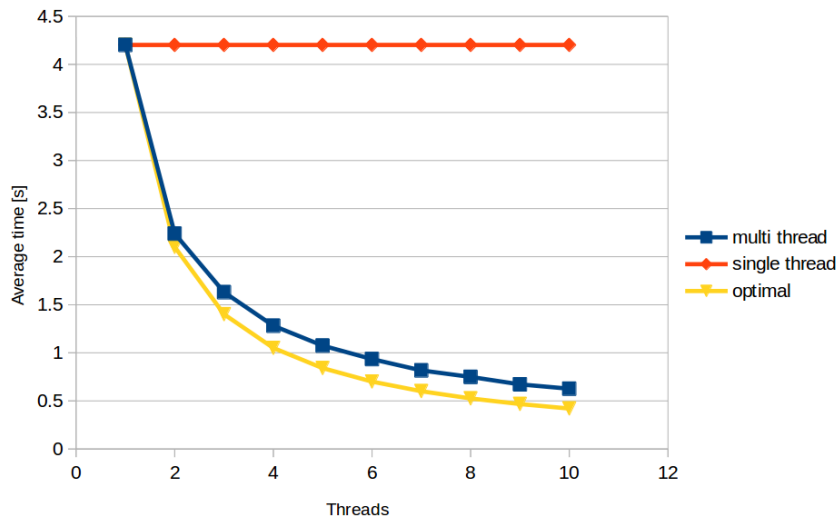
Paralelní programování

Měření zrychlení

Je dané zrychlení dostatečné? Můžeme být rychlejší?

- V optimálním případě se paralelní verze zrychluje proporcčně s počtem jader

| Vláken | 1 | 2 | 3 | 4 |
|--------|---|-----|-----|-----|
| Čas | x | x/2 | x/3 | x/4 |



Často vyjádřeno jako zrychlení:

$$S = \frac{T_{serial}}{T_{parallel}}$$

Paralelní programování

Měření zrychlení

Můžeme se vždy dostat k lineárnímu zrychlení?

- Paralelní verze algoritmů mají (téměř) vždy další režii
 - spouštění vláken
 - zámky
 - synchronizace
 - ...
- Program/algoritmus často vyžaduje určitou sériovou část
 - Necht' jsme schopni přepsat 90% kódu s lineárním zrychlením
 - $$S = \frac{T_{serial}}{0.9 \times \frac{T_{serial}}{p} + 0.1 \times T_{serial}} \leq \frac{T_{serial}}{0.1 \times T_{serial}}$$
 - To znamená, že pokud sériový program trvá 20 sekund, nikdy nedosáhneme zrychlení větší než 10

Amdahlův zákon