



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 Bošanský

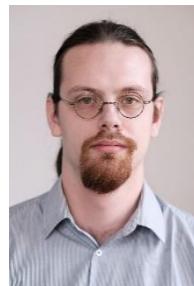


Michal Jakob

Cvičící



Peter Macejko



Petr Tomášek



Petr Váňa



David Fiedler



Jan Mrkos

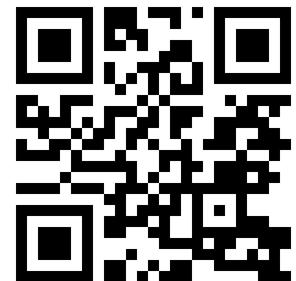


David Milec

Organizace a přehled

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

Bookmark
This Page



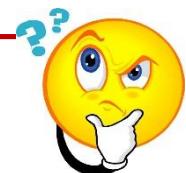
Hodnocení

- 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 jej, 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

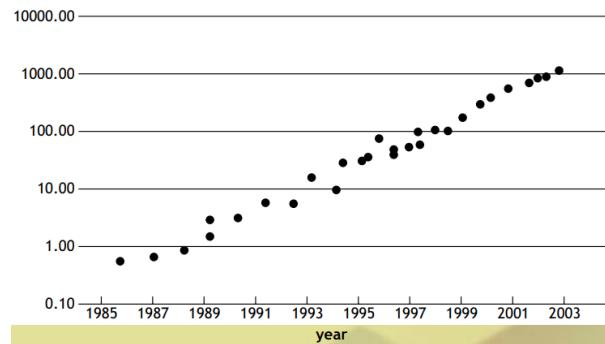
Obrázky převzaty z:

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

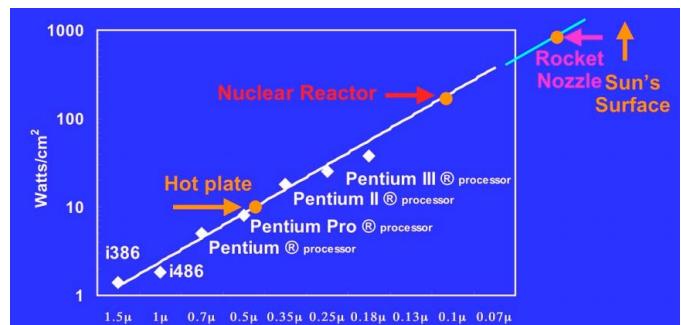
Krátká historie paralelních výpočtů

Nárůst výkonu jednoho procesoru

- Dlouhé roky rostl výpočetní výkon exponenciálně ...



- ... až v letech 2004 nebylo možné dále pouze navýšovat 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

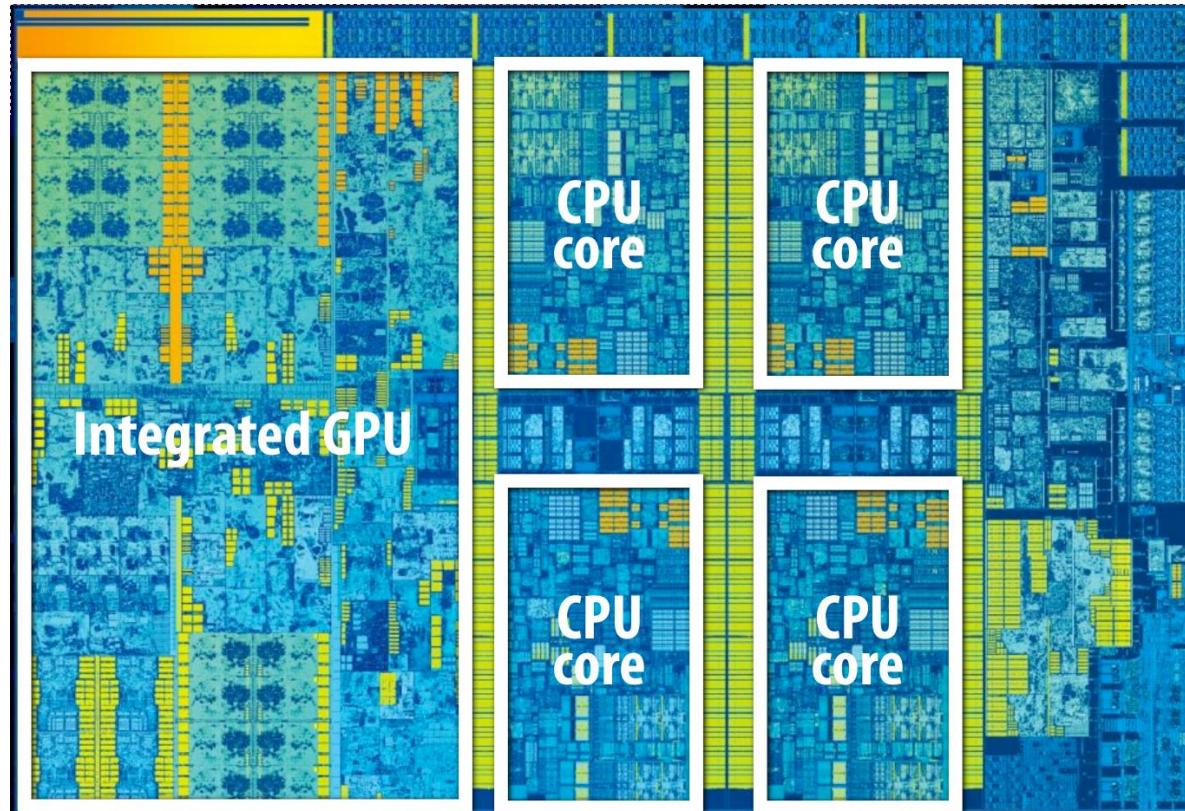
Dnešní paralelní stroje



Krátká historie paralelních výpočtů

Struktura dnešních CPU

- Intel Skylake (2015)
 - i7 – čtyřjádrový CPU + vícejádrový 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

Top 10 positions of the 54th TOP500 in November 2019^[25]

Rank ↴	Rmax Rpeak ↴ (PFLOPS)	Name ↴	Model ↴	Processor	Interconnect ↴	Vendor ↴	Site country, year	Operating system ↴
1 —	148.600 200.795	Summit	IBM Power System AC922	POWER9, Tesla V100	InfiniBand EDR	IBM	Oak Ridge National Laboratory United States, 2018	Linux (RHEL)
2 —	94.640 125.712	Sierra	IBM Power System S922LC	POWER9, Tesla V100	InfiniBand EDR	IBM	Lawrence Livermore National Laboratory United States, 2018	Linux (RHEL)
3 —	93.015 125.436	Sunway TaihuLight	Sunway MPP	SW26010	Sunway ^[26]	NRPCPC	National Supercomputing Center in Wuxi China, 2016 ^[26]	Linux (Raise)
4 —	61.445 100.679	Tianhe-2A	TH-IVB-FEP	Xeon E5-2692 v2, Matrix-2000 ^[27]	TH Express-2	NUDT	National Supercomputing Center in Guangzhou China, 2013	Linux (Kylin)
5 ▲	23.516 38.746	Frontera	Dell C6420	Xeon Platinum 8280 (subsystems with e.g. POWER9 CPUs and Nvidia GPUs were added after official benchmarking ^[11])	InfiniBand HDR	Dell EMC	Texas Advanced Computing Center United States, 2019	Linux (CentOS)
6 ▼	21.230 27.154	Piz Daint	Cray XC50	Xeon E5-2690 v3, Tesla P100	Aries	Cray	Swiss National Supercomputing Centre Switzerland, 2016	Linux (CLE)
7 ▼	20.159 41.461	Trinity	Cray XC40	Xeon E5-2698 v3, Xeon Phi 7250	Aries	Cray	Los Alamos National Laboratory United States, 2015	Linux (CLE)
8 ▼	19.880 32.577	AI Bridging Cloud Infrastructure ^[28]	PRIMERGY CX2550 M4	Xeon Gold 6148, Tesla V100	InfiniBand EDR	Fujitsu	National Institute of Advanced Industrial Science and Technology Japan, 2018	Linux
9 —	19.477 26.874	SuperMUC-NG ^[29]	ThinkSystem SD530	Xeon Platinum 8174 (plus not benchmarked e.g. 32 cloud GPU nodes with Tesla V100 ^[30])	Intel Omni-Path	Lenovo	Leibniz Supercomputing Centre Germany, 2018	Linux (SLES)
10 ▲	18.200 23.047	Lassen	IBM Power System S922LC	POWER9, Tesla V100	InfiniBand EDR	IBM	Lawrence Livermore National Laboratory United States, 2018	Linux (RHEL)

Krátká historie paralelních výpočtů

Výpočetní gridy – a co u nás?

- RCI ČVUT cluster
 - n01-20 CPU nodes: 24 cores/48 threads 3.2GHz (2 x Intel Xeon Scalable Gold 6146), 384GB RAM,
 - n21-n32 GPU nodes: 36 cores/72 threads 2.7GHz (2 x Intel Xeon Scalable Gold 6150), 384GB RAM, 4 x Tesla V100 with NVLink,
 - n33 multi-CPU node: 192 cores/ 384 threads 2.1GHz (8 x Intel Xeon Scalable Platinum 8160), 1536GB RAM



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];
```

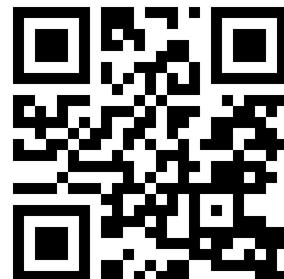
Varianta B

```
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

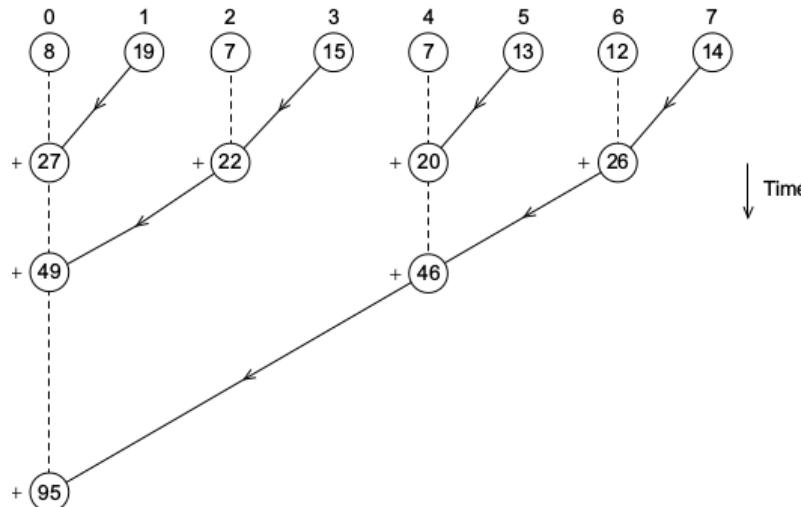
Paralelizace

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
 - Opravovaní 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

PThread 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;
}
```

Pthread 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;
}
```

Pthread vs. C++11 vs. OpenMP

Ochutnávka (OpenMP)

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

const int thread_count = 10;

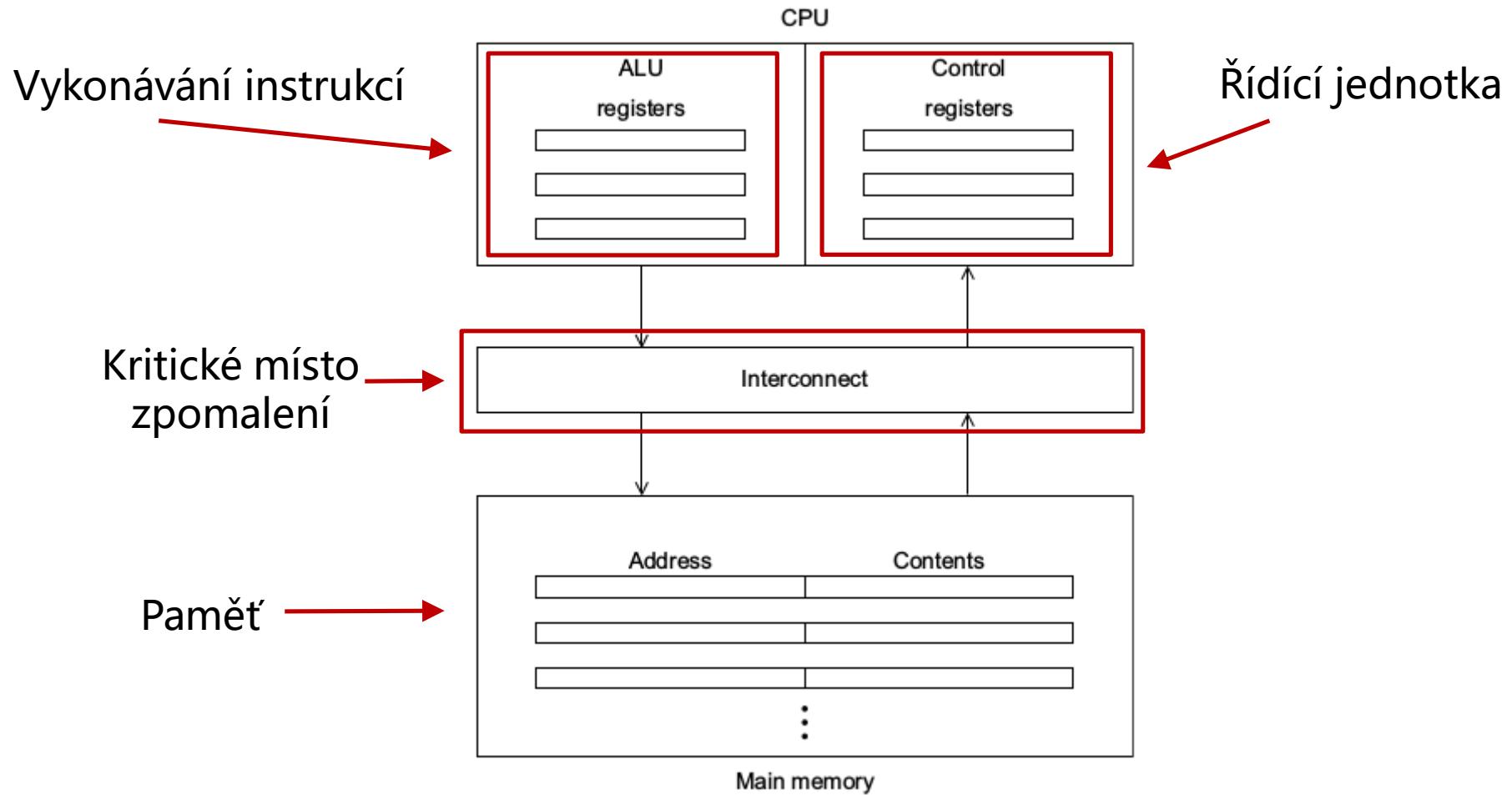
void Hello() {
    int my_rank = omp_get_thread_num();
    int threads = omp_get_num_threads();
    std::cout << "Hello from thread " << my_rank << " of " << threads << std::endl;
}

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 – $7 \times 1000 \times (\text{čas 1 operace; } 1\text{ns})$

Potřebný HW základ

Pipelines

- Paralelizace na úrovni instrukcí (ILP)
- Příklad:
 - Chceme sečíst 2 vektory reálných čísel (float [1000])
 - 1 součet – 7 operací
 - Bez ILP – $7 \times 1000 \times$ (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

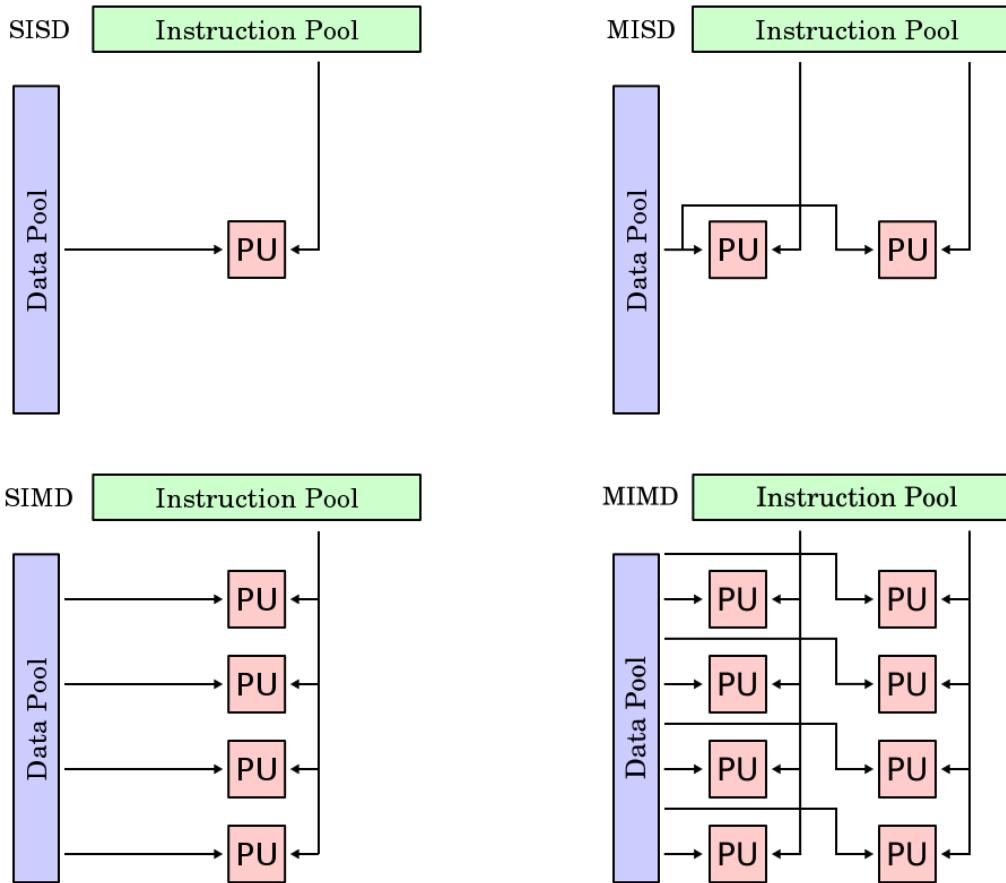
```
for (i=0; i<1000; i++)  
z[i]=x[i]+y[i];
```

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

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

Potřebný HW základ

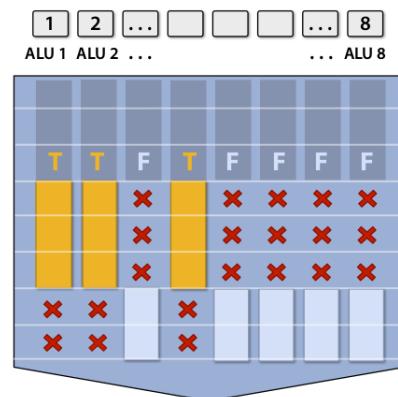
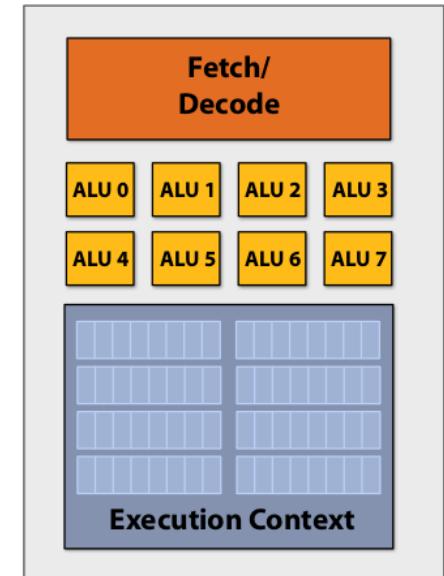
Paralelní hardware – Flynnova taxonomie



Potřebný HW základ

Paralelní hardware – Flynnova taxonomie

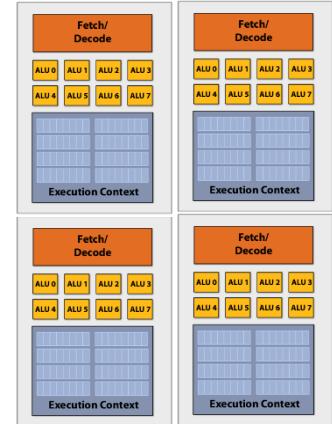
- SIMD (Single Instruction Multiple Data)
 - Jedna řídící jednotka, vícero ALU jednotek
 - Datový paralelismus
 - Vektorové procesory, GPU
 - Běžné jádra CPU podporují SIMD paralelismus
 - instrukce SSE, AVX
- Větvení na SIMD



Potřebný HW základ

Paralelní hardware – Flynnova taxonomie

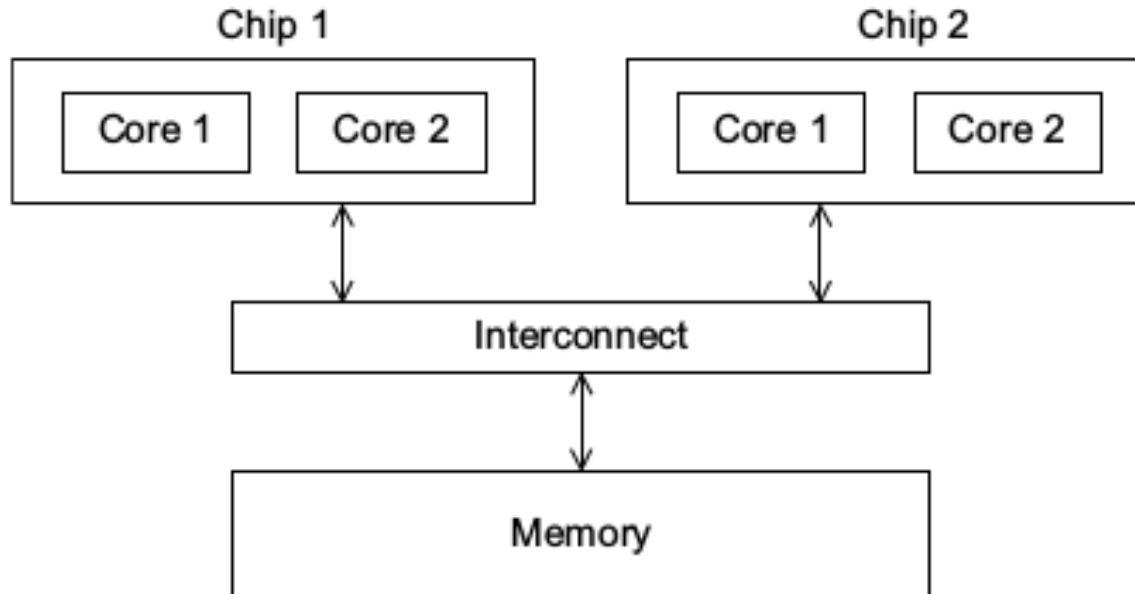
- MIMD (Multiple Instruction Multiple Data)
 - Více-jádrové procesory
 - Různé jádra vykonávají různé instrukce
 - Víceprocesorové počítače
- A co paměť?



Potřebný HW základ

Systémy se sdílenou pamětí

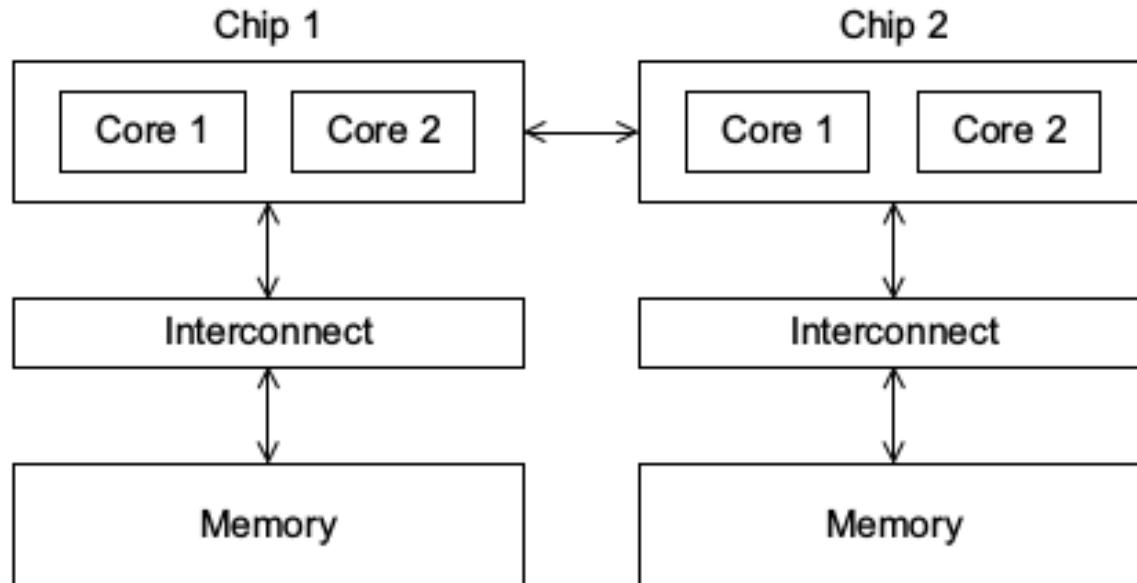
- Uniform Memory Access (UMA)



Potřebný HW základ

Systémy se sdílenou pamětí

- Nonuniform Memory Access (NUMA)

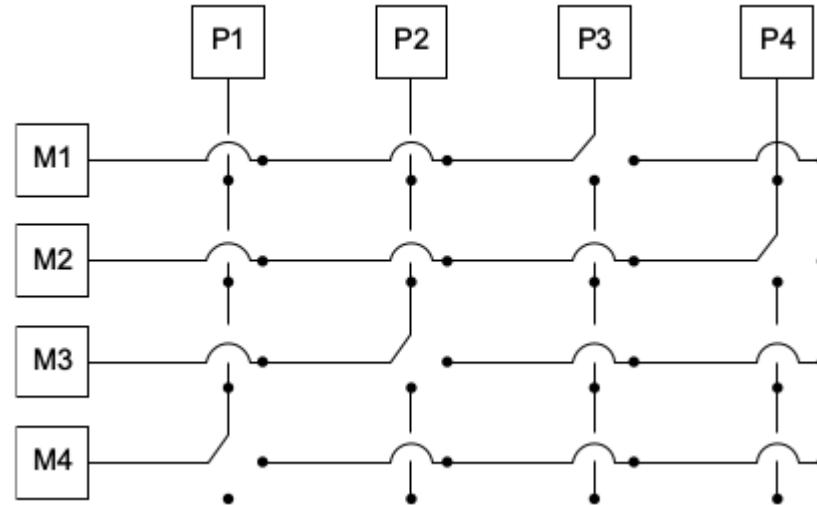


Potřebný HW základ

Systémy se sdílenou pamětí – typy přístupů k paměti

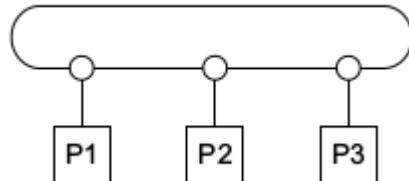
- Sběrnice

- Mřížka



- Sít/Kruh

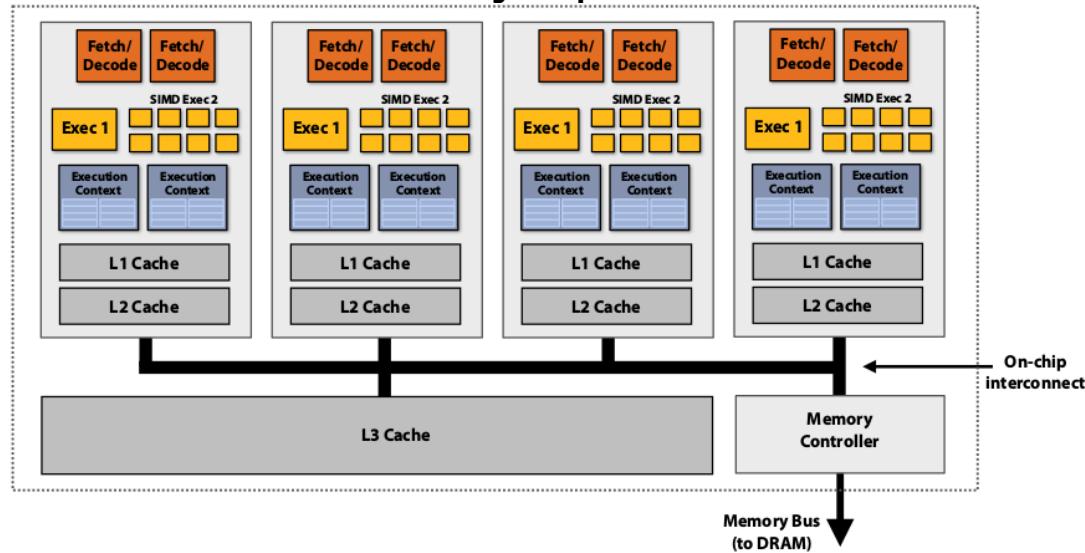
- ...



Potřebný HW základ

Základy Cache

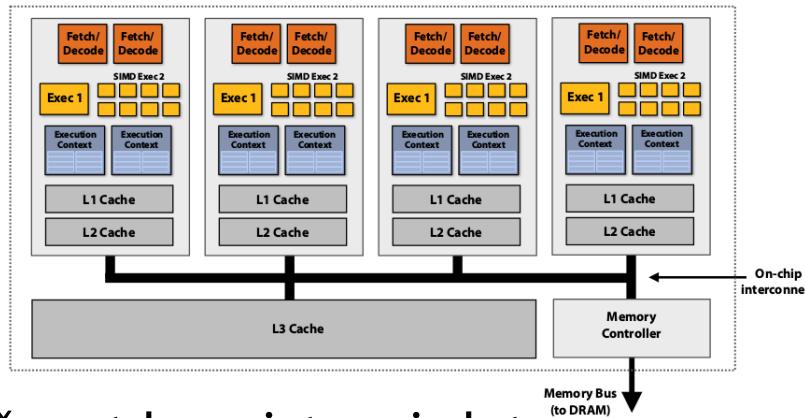
- CPU cache
 - Programy často přistupují k paměti lokálně (lokalita v prostoru a čase)
 - Cache se upravuje po řádcích
- Každé jádro má vlastní cache + existuje společná cache



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

- Vratíme se k příkladu se sčítáním vektoru čísel
 - (tedy budeme sčítat celou část druhých odmocnin)

sčítané pole

id vlákna

pole pro dílčí součty

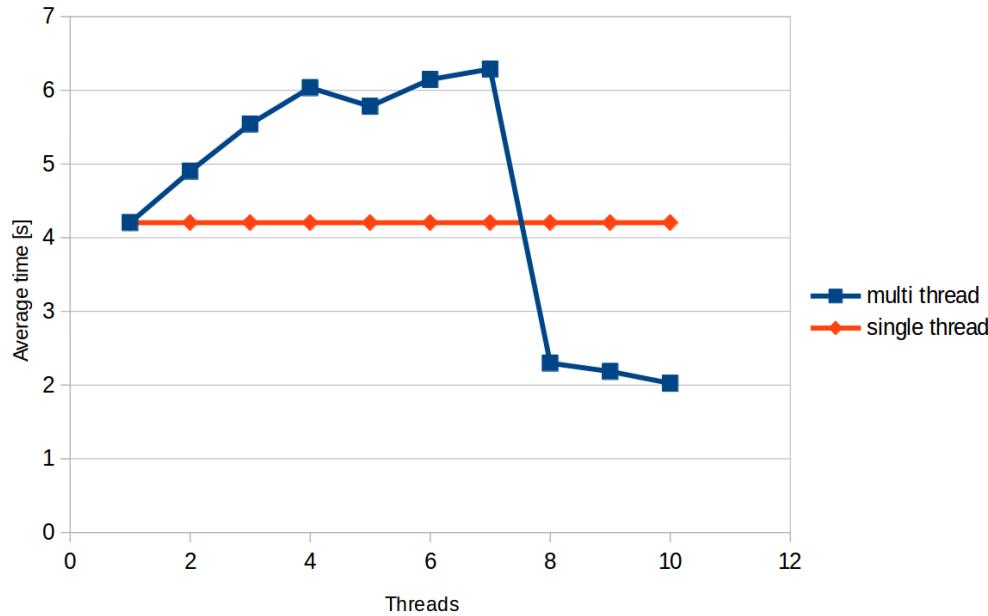
```
long sum(std::vector<int>& vector_to_sum, int thread, std::vector<long>& sums) {
    for (int i=thread; i<SIZE; i += thread_count)
        sums[thread] += sqrt(vector_to_sum[i]);
    for (int j=1; j<log2(thread_count)+1; j++) {
        if ((thread % (int)pow(2,j)) != 0) break;
        int k = (int)pow(2,j-1);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



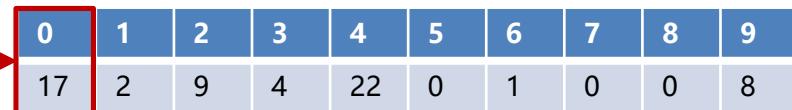
0	1	2	3	4	5	6	7	8	9
17	2	9	4	22	0	1	0	0	8

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;
}
```

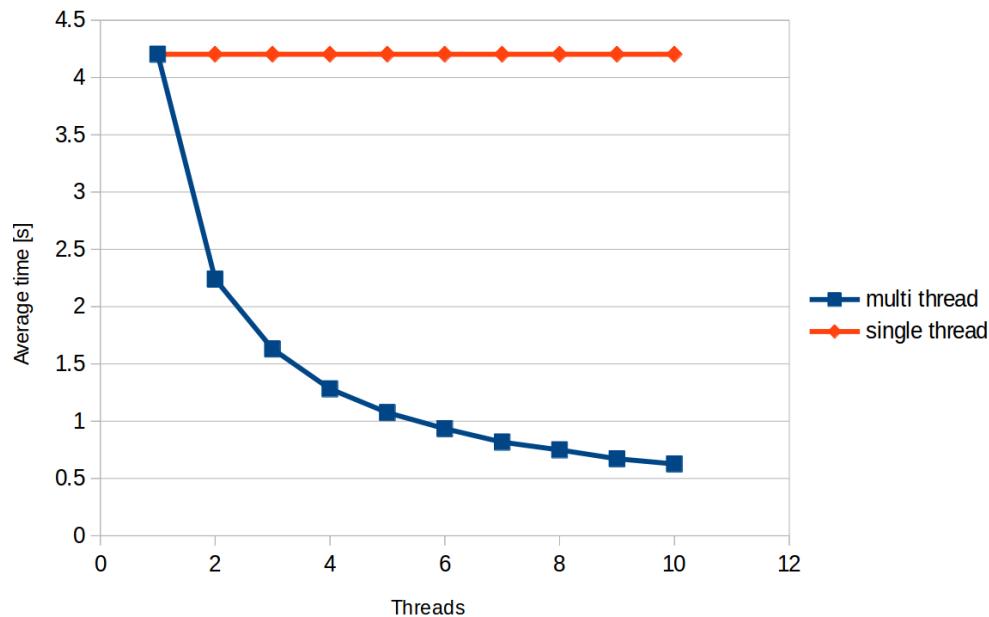
každé vlákno zapisuje
do lokální proměnné

pouze finální výsledek
se zapíše do vektoru

Potřebný HW základ

False Sharing

lokální proměnná – opravdu to pomůže?



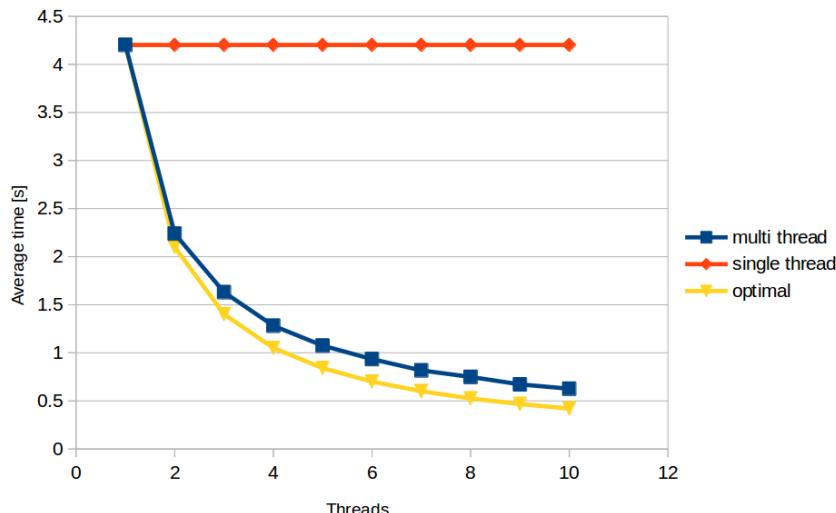
Paralelní programování

Měření zrychlení

Je dané zrychlení dostatečné? Můžeme být rychlejší?

- V optimálním případě se paralelní verze zrychluje propořčně 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
 - Nechť 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



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

Dnešní přednáška

Motivace



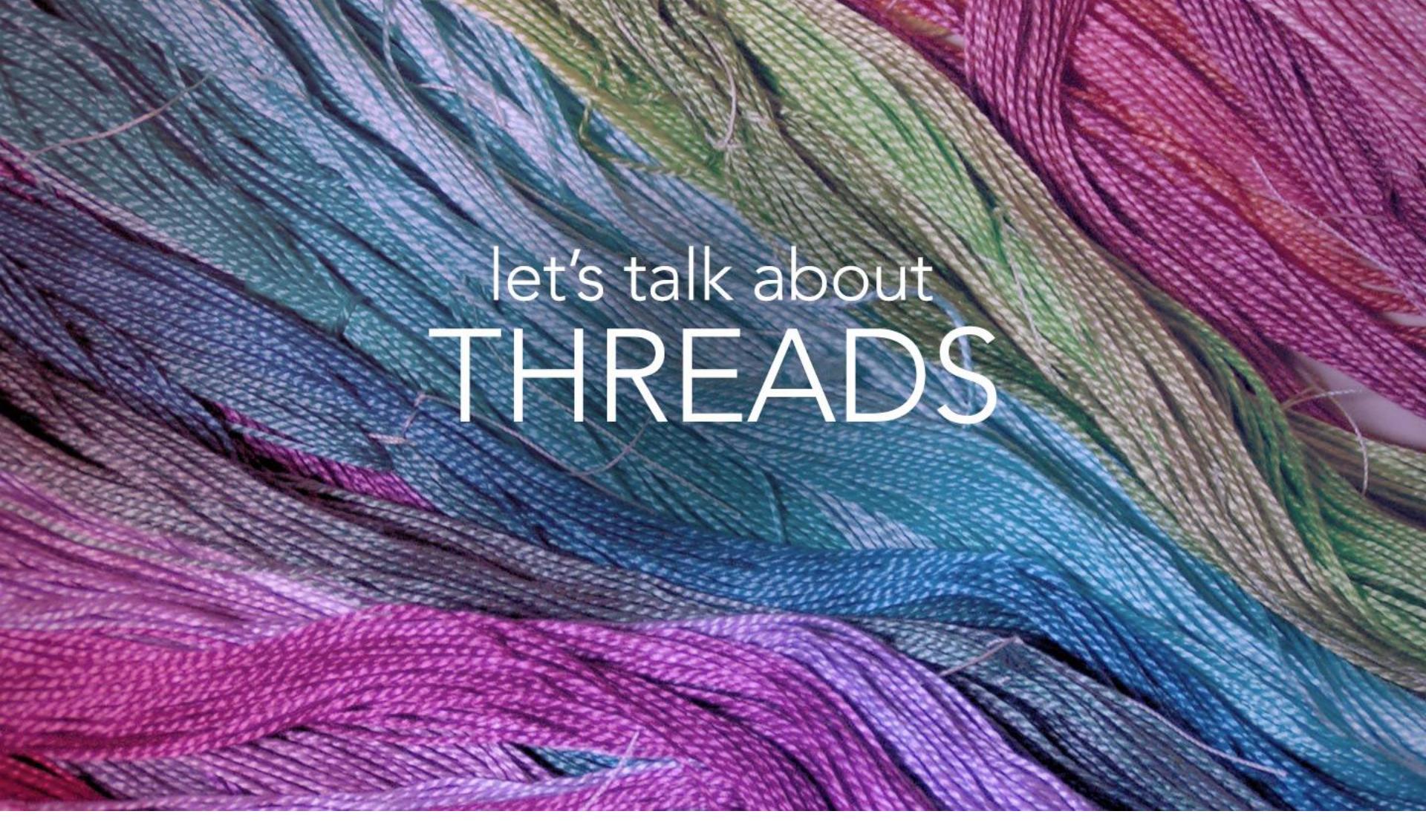
Dnešní přednáška

Nástroje



Dnešní přednáška

Vlákna



let's talk about
THREADS

Vlákna – úvod, spuštění

V C++11 (a dalších)

- V rámci C++ exportovány hlavičkou <thread>
 - std::thread
 - první argument je funkce, kterou bude vlákno vykonávat
 - ostatní argumenty jsou argumenty dané funkce

```
#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;
}
```

Vlákna - ukončení

- V rámci C++ exportovány hlavičkou <thread>
 - std::thread
 - hlavní vlákno musí zavolat buď metodu **join**
 - případně **detach**
 - měli bychom kontrolovat, jestli dané vlákno ještě existuje
 - if (threads[thread].joinable()) ...



Přístup ke sdílené paměti

Synchronizace vláken

- Při současném přístupu ke sdílené proměnné může dojít k uspořádání provedených instrukcí vedoucích k nesprávnému (jinému než očekávanému) výsledku
- Příklad – vyváříme histogram zbytků po dělení čísel:

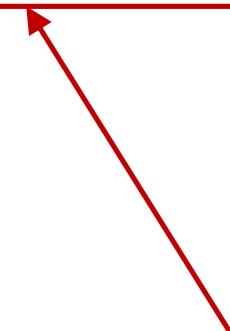
```
void hist(std::vector<int>& vector, int thread, std::vector<int>& histogram) {  
    std::vector<int> local(PARTS);  
    for (int i = thread; i < SIZE; i += thread_count) {  
        local[vector[i] % PARTS]++;  
    }  
  
    for (int i = 0; i < PARTS; i++) {  
        histogram[i] += local[i];  
    }  
}
```

Přístup ke sdílené paměti

Synchronizace vláken

- Při současném přístupu ke sdílené proměnné může dojít k uspořádání provedených instrukcí vedoucích k nesprávnému (jinému než očekávanému) výsledku
- Příklad – vyváříme histogram zbytků po dělení čísel:

```
void hist(std::vector<int>& vector, int thread, std::vector<int>& histogram) {  
    std::vector<int> local(PARTS);  
    for (int i=thread; i<SIZE; i += thread_count) {  
        local[vector[i] % PARTS]++;  
    }  
  
    for (int i=0; i<PARTS; i++) {  
        histogram[i] += local[i];  
    }  
}
```

- 
- std::thread předá referenci pomocí std::ref(histogram)

Přístup ke sdílené paměti

Synchronizace vláken - zámky

- Při současném přístupu ke sdílené proměnné může dojít k uspořádání provedených instrukcí vedoucích k nesprávnému (jinému než očekávanému) výsledku
- Příklad – vyváříme histogram zbytků po dělení čísel:
- Mutex – první řešení

```
std::mutex hist_mutex;

void hist(std::vector<int>& vector, int thread, std::vector<int>& histogram) {
    std::vector<int> local(PARTS);
    for (int i=thread; i<SIZE; i += thread_count) {
        local[vector[i] % PARTS]++;
    }

    hist_mutex.lock();
    for (int i=0; i<PARTS; i++) {
        histogram[i] += local[i];
    }
    hist_mutex.unlock();
}
```

Přístup ke sdílené paměti

Synchronizace vláken - zámky

- Při současném přístupu ke sdílené proměnné může dojít k uspořádání provedených instrukcí vedoucích k nesprávnému (jinému než očekávanému) výsledku
- Příklad – vyváříme histogram zbytků po dělení čísel:
- Mutex – druhé řešení

```
std::vector<std::mutex> hist_part_mutex(PARTS);

void hist(std::vector<int>& vector, int thread, std::vector<int>& histogram) {
    std::vector<int> local(PARTS);
    for (int i=thread; i<SIZE; i += thread_count) {
        local[vector[i] % PARTS]++;
    }

    for (int i=0; i<PARTS; i++) {
        hist_part_mutex[i].lock();
        histogram[i] += local[i];
        hist_part_mutex[i].unlock();
    }
}
```

Přístup k více proměnným

- Co když potřebujeme výlučný přístup ke dvěma (nebo více) proměnným
 - např. chci mezivýsledek z operací, které se provádějí
 - zamknu 1., zamknu 2.

Přístup k více proměnným

- Co když potřebujeme výlučný přístup ke dvěma (nebo více) proměnným
 - např. chci mezivýsledek z operací, které se provádějí
 - zamknu 1., zamknu 2.
 - možný deadlock!



Přístup k více proměnným

- Co když potřebujeme výlučný přístup ke dvěma (nebo více) proměnným
 - např. chci mezivýsledek z operací, které se provádějí
 - zamknu 1., zamknu 2.
 - možný deadlock!



- musíme zamknout oba zámky současně

```
void thread_operation() {
    std::lock(mutex1,mutex2);
    ...
    complicated_task();
    ...
    mutex1.unlock();
    mutex2.unlock();
}
```

Deadlock

- Deadlocky mohou vzniknout pokud:
 1. Každý zámek může vlastnit maximálně jedno vlákno
 2. Vlákno aktuálně vlastní (má zamčený) alespoň jeden zámek a požaduje zamknout alespoň jeden další
 3. Není možné odebrat vlastnictví zámku
 4. Existuje cyklická závislost mezi vlákny



Deadlock vznikne velice snadno

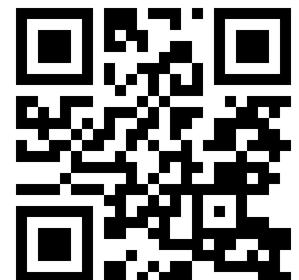
Deadlock

- Deadlocky mohou vzniknout pokud:
 1. Každý zámek může vlastnit maximálně jedno vlákno
 2. Vlákno aktuálně vlastní (má zamčený) alespoň jeden zámek a požaduje zamknout alespoň jeden další
 3. Není možné odebrat vlastnictví zámku
 4. Existuje cyklická závislost mezi vlákny



Deadlock vznikne velice snadno

<https://goo.gl/a6BEMb>



Odemykání zámků

- Zamknuté zámky je dobré odemykat
- A co v případě výjimky?
 - Musíme zajistit odemknutí při všech možných ukončeních

```
void operation() {
    mutex.lock();
    try {
        ...
        complicated_task();
        ...
    } catch (std::string e) {
        mutex.unlock();
        throw e;
    }
    mutex.unlock();
}
```

Automatická správa zámků

- Lock_guard mutex – RAI správa zámků (Resource acquisition is initialization)
 - Konstruktor automaticky volá lock() na zámku, destrukturor zámek odemyká

```
void operation() {
    std::lock_guard<std::mutex> guard(mutex);
    try {
        ...
        complicated_task();
        ...
    } catch (std::string e) {
        throw e;
    }
}
```

- A co když chceme mít v lock_guard 2 zámky?

Automatická správa zámků

Vícero proměnných

- Lock_guard mutex
 - Konstruktor automaticky volá lock() na zámku, destruktur zámek odemyká
 - A co když chceme mít v lock_guard 2 zámky?

```
std::mutex m1;
std::mutex m2;

void f(int id) {
    std::lock(m1, m2);
    std::lock_guard<std::mutex> lock1(m1, std::adopt_lock);
    std::lock_guard<std::mutex> lock2(m2, std::adopt_lock);
    std::cout << "Thread " << id << " says hi." << std::endl;
// we do not need to unlock
}

int main(int argc, char* argv[]) {
    std::thread t1(f, 1);
    std::thread t2(f, 2);

    t1.join();
    t2.join();
}
```

Časově omezené čekání

- Pokud nechceme, aby se vlákno při pokusu o zamknutí zámku zablokovalo, můžeme využít časově omezených metod
 - časové zámky (timed_mutex)

```
std::timed_mutex m;
const int THREADS = 10;
const std::chrono::milliseconds timeout(100);
const std::chrono::milliseconds timeout2(20);

void f(int id) {
    if (m.try_lock_for(timeout)) {
        std::cout << "Thread " << id << " is computing stuff." << std::endl;
        std::this_thread::sleep_for(timeout2);
        m.unlock();
    } else {
        std::cout << "Thread " << id << " is skipping." << std::endl;
    }
}

int main(int argc, char* argv[]) {
    std::vector<std::thread> threads;

    for (int i=0; i<THREADS; i++)
        threads.push_back(std::thread(f, i));

    for (int i=0; i<THREADS; i++)
        threads[i].join();
}
```

Opakovane zamykani

- Co když už máme naimplementovaných několik thread-safe metod, které bychom chtěli zavolat z jiné thread-safe metody?
 - Mějme operace nad prvky matice add, divide

```
void add_number(std::vector<std::vector<int>>& matrix, int row, int column, int value) {  
    std::lock_guard<std::mutex> l(mutexes[row][column]);  
    matrix[row][column] += value;  
}  
  
void divide_by_number(std::vector<std::vector<int>>& matrix, int row, int column, int value) {  
    std::lock_guard<std::mutex> l(mutexes[row][column]);  
    matrix[row][column] = matrix[row][column] / value;  
}
```

- Když bychom měli jinou metodu, která tyto metody volá, zámky již budou zamknuté ...

Opakovane zamykani

- Co když už máme naimplementovaných několik thread-safe metod, které bychom chtěli zavolat z jiné thread-safe metody?
 - Mějme operace nad prvky matice add, divide

```
void add_number(std::vector<std::vector<int>>& matrix, int row, int column, int value) {  
    std::lock_guard<std::mutex> l(mutexes[row][column]);  
    matrix[row][column] += value;  
}  
  
void divide_by_number(std::vector<std::vector<int>>& matrix, int row, int column, int value) {  
    std::lock_guard<std::mutex> l(mutexes[row][column]);  
    matrix[row][column] = matrix[row][column] / value;  
}
```

- Když bychom měli jinou metodu, která tyto metody volá, zámky již budou zamknuté ...
- Můžeme použít **recursive_mutex**

Recursive Lock

- recursive_lock

```
void add_number(std::vector<std::vector<int>>& matrix, int row, int column, int value) {
    std::lock_guard<std::recursive_mutex> l(mutexes[row][column]);
    matrix[row][column] += value;
}

void divide_by_number(std::vector<std::vector<int>>& matrix, int row, int column, int value) {
    std::lock_guard<std::recursive_mutex> l(mutexes[row][column]);
    matrix[row][column] = matrix[row][column] / value;
}

//there is a one dimension padding in the matrix so that we do not need to check boundaries
void average_with_neighbours(std::vector<std::vector<int>>& matrix, int row, int column) {
    std::lock(mutexes[row][column-1], mutexes[row][column], mutexes[row][column+1], mutexes[row-1][column], mutexes[row+1][column]);
    std::cout << "thread is averaging " << row << "," << column << " old_value: " << matrix[row][column];
    int v = (matrix[row][column-1] + matrix[row][column+1] + matrix[row-1][column] + matrix[row+1][column])/8;
    divide_by_number(matrix, row, column, 2);
    add_number(matrix, row, column, v);
    std::cout << " new_value: " << matrix[row][column] << std::endl;
    for (int i=-1; i<=1; i++)
        for (int j=-1; j<=1; j++) {
            if (i*j != 0) continue;
            mutexes[row+i][column+j].unlock();
        }
}
```

Atomické proměnné

- Zámky znamenají práci navíc
- Někdy není nutné je použít
 - **Atomické proměnné** umožňují provedení atomických operací, limitují možné optimalizace kompilátoru
 - umožňují základní operace bez zámků – typicky rychlejší
 - **atomic<type>** (např. atomic<int> (=atomic_int), ...)
- Vratíme se k histogramu

```
void hist_atomic(std::vector<int>& vector, int thread, std::vector<std::atomic_int*>& histogram) {
    std::vector<int> local(PARTS);
    for (int i=thread; i<SIZE; i += thread_count) {
        local[vector[i] % PARTS]++;
    }

    for (int i=0; i<PARTS; i++) {
        histogram[i] += local[i];
    }
}
```

- při současném přístupu k vícero proměnným musíme použít zámky

Zpracování výsledků z jiných vláken

- Při inicializaci přes std::thread, vlákna nic nevracejí
 - dílčí výsledky jsou předávané přes sdílené proměnné
- Co když chceme vytvořit několik vláken pro pomocné úkoly a jejich výsledky zpracovat?
 - struktura future a metoda std::async

```
#include <thread>
#include <future>
#include <iostream>

int foo() {
    std::cout << "I'm a thread" << std::endl;
    return 42;
}

int main() {
    auto future = std::async(std::launch::async, foo);
    std::cout << future.get(); // Red arrow points here
    return 0;
}
```

- future.get() blokuje aktivní vlákno a čeká na výsledek
- lze volat pouze jednou
- future.wait() čeká, ale nezkonzumuje výsledek

Zpracování výsledků z jiných vláken (2)

- Destruktor std::future vytvořené pomocí std::async je blokující

```
std::async(std::launch::async, foo);  
std::async(std::launch::async, foo2);
```

- funkce foo2 se spustí pouze po skončení vlákna s metodou foo
- pokud v metodě zavoláme

```
auto future = std::async(std::launch::async, foo);
```

- při ukončení metody se bude čekat na ukončení vláken

Zpracování výsledků z jiných vláken (2)

- Destruktor std::future vytvořené pomocí std::async je blokující

```
std::async(std::launch::async, foo);  
std::async(std::launch::async, foo2);
```

- funkce foo2 se spustí pouze po skončení vlákna s metodou foo
- pokud v metodě zavoláme

```
auto future = std::async(std::launch::async, foo);
```

- při ukončení metody se bude čekat na ukončení vláken

- Futures také umožňují časově-omezené čekání na výsledek
 - Má smysl pokud hlavní vlákno aktivně pracuje a průběžně kontroluje dostupnost dílčích výsledků z asynchronně puštěných vláken

Zpracování výsledků z jiných vláken (3)

- Futures také umožňují časově-omezené čekání na výsledek

```
#include <thread>
#include <future>
#include <iostream>
#include <chrono>

int function(int duration) {
    std::this_thread::sleep_for(std::chrono::seconds(duration));
    return duration*4-2;
}

int main() {
    auto f1 = std::async(std::launch::async, function, 5);
    auto f2 = std::async(std::launch::async, function, 3);

    auto timeout = std::chrono::nanoseconds(10);
    while(f1.valid() || f2.valid()) {
        if(f1.valid() && f1.wait_for(timeout) == std::future_status::ready) {
            std::cout << "Task1 is done with result " << f1.get() << std::endl;
        }
        if(f2.valid() && f2.wait_for(timeout) == std::future_status::ready) {
            std::cout << "Task2 is done with result " << f2.get() << std::endl;
        }
        std::cout << "Work in the main thread." << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }

    return 0;
}
```

Podmínkové proměnné

- Podmínkové proměnné (Condition variables) slouží ke komunikaci mezi vlákny
 - např. vlákno chce předat zprávu „dílčí výsledek je již připraven“
 - typický příklad – fronta úkolů ke zpracování (Producer-Consumer)
 - 1 (nebo víc) vlákno generuje úkoly
 - další vlákna je zpracovávají
 - zpracovávající vlákna musí dostat notifikaci o tom, že další úkol je připraven ke zpracování
 - podmíněná proměnná navázaná na zámek fronty úkolů
 - nechť má fronta úkolů omezenou velikost (např. aby nám nepřetekla paměť)
 - čekající vlákno je notifikováno metodou `notify_one()` (případně `notify_all()`)

Podmínkové proměnné

```
int front = 0;
int rear = 0;
int count = 0;

std::vector<std::pair<int,int>> buffer;
std::mutex lock;
std::condition_variable not_full;
std::condition_variable not_empty;

void add_task(int row, int column){
    std::unique_lock<std::mutex> l(lock);
    not_full.wait(l, [this](){return count != MAXPOOL; });

    buffer[rear] = std::pair<int,int>(row,column);
    rear = (rear + 1) % MAXPOOL;
    count++;
    l.unlock();
    not_empty.notify_one();
}

std::tuple<int,int,int> execute_task(){
    std::unique_lock<std::mutex> l(lock);

    not_empty.wait(l, [this](){return count != 0; });

    std::pair<int,int> square = buffer[front];
    front = (front + 1) % MAXPOOL;
    count--;

    l.unlock();
    not_full.notify_one();

    int result = average_with_neighbours(*matrix, square.first, square.second);
    return std::tuple<int,int,int>(result,square.first,square.second);
}
```

Budoucnost

V C++17 (a dalších)

- Další změny a podpora paralelismu v C++17
 - ...
 - Parallel versions of STL algorithms
 - ...



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

Dnešní přednáška

Motivace



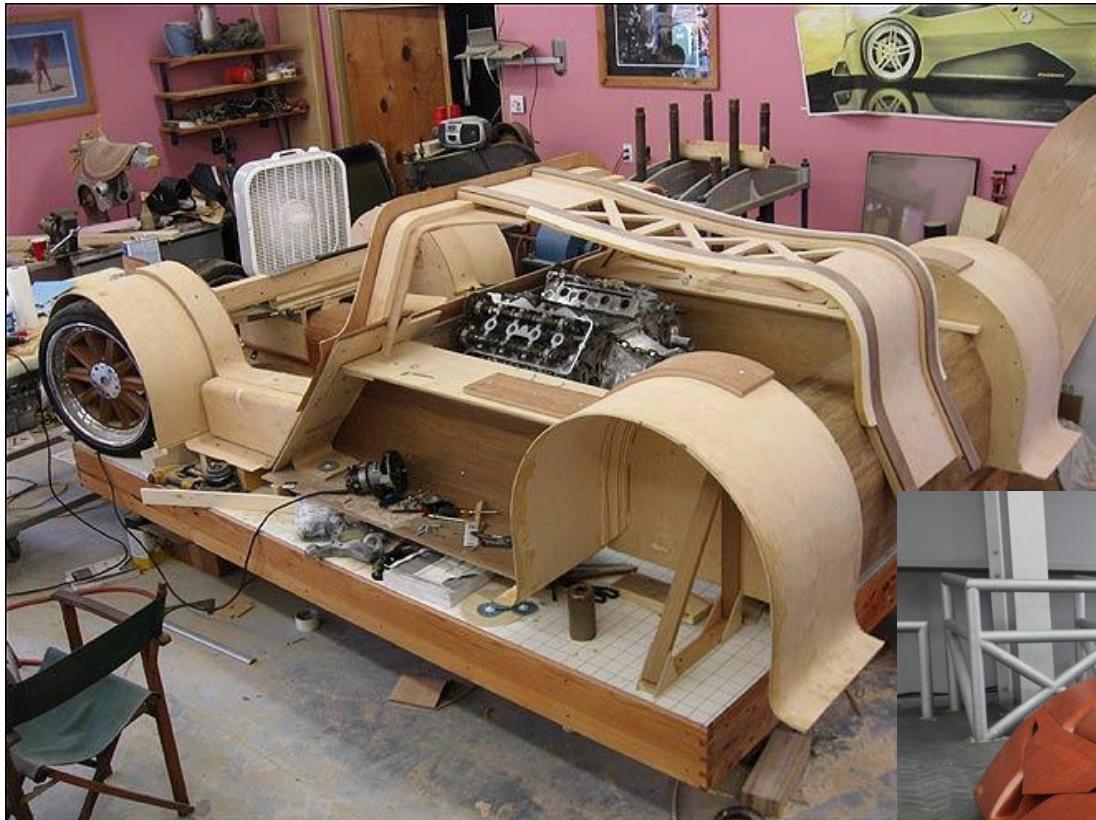
Dnešní přednáška

Motivace



Dnešní přednáška

Motivace



Dnešní přednáška

Další nástroje



Dnešní přednáška

OpenMP



<http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf>

OpenMP – úvod, spuštění

V C++11 (podporuje C,C++)

- V rámci C/C++ exportovány hlavičkou „omp.h“
 - rozšíření kompilátoru (různé kompilátory podporují různé verze OpenMP)
 - lze přistoupit pomocí #pragma omp
 - základní použití pro paralelizaci for cyklu

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

int main(int argc, char* argv[]) {
    std::cout << "Hello from the main thread\n";

    #pragma omp parallel for
    for (int i=0; i<10; i++)
        std::cout << "Item " << i << " is processed by thread" << omp_get_thread_num() << std::endl;

    return 0;
}
```

OpenMP – úvod, spuštění

V C++11 (podporuje C,C++)

- **#pragma omp parallel for** je pouze zkratkou za

- ```
#pragma omp parallel
{
 #pragma omp for
 for (int i=0; i<MAX; i++)
}
```

```
int main(int argc, char* argv[]) {
#pragma omp parallel
{
 std::cout << "this is a line printed by thread" << omp_get_thread_num() << std::endl;

#pragma omp for
 for (int i=0; i<10; i++)
 std::cout << "for-loop line " << i << " printed by thread" << omp_get_thread_num() << std::endl;

}
std::cout << "Hello from the main thread\n";
return 0;
}
```

# OpenMP – úvod, spuštění

V C++11 (podporuje C,C++)

- **#pragma omp parallel for** je pouze zkratkou za
  - `#pragma omp parallel {`
  - `#pragma omp for`
  - `for (int i=0; i<MAX; i++)`
  - `}`
- vytvoří tým vláken, které vykonávají blok

```
int main(int argc, char* argv[]) {
 #pragma omp parallel
 {
 std::cout << "this is a line printed by thread" << omp_get_thread_num() << std::endl;

 #pragma omp for
 for (int i=0; i<10; i++)
 std::cout << "for-loop line " << i << " printed by thread" << omp_get_thread_num() << std::endl;

 }
 std::cout << "Hello from the main thread\n";
 return 0;
}
```

# OpenMP – úvod, spuštění

V C++11 (podporuje C,C++)

- **#pragma omp parallel for** je pouze zkratkou za

- `#pragma omp parallel` ←
  - vytvoří tým vláken, které vykonávají blok
- `#pragma omp for` ←
  - vezme následující for cyklus a rozdělí jej mezi vlákna v týmu

```
int main(int argc, char* argv[]) {
 #pragma omp parallel
 {
 std::cout << "this is a line printed by thread" << omp_get_thread_num() << std::endl;

 #pragma omp for
 for (int i=0; i<10; i++)
 std::cout << "for-loop line " << i << " printed by thread" << omp_get_thread_num() << std::endl;
 }

 std::cout << "Hello from the main thread\n";
 return 0;
}
```

# OpenMP – úvod, spuštění

V C++11 (podporuje C,C++)

- **#pragma omp parallel for** je pouze zkratkou za

- `#pragma omp parallel`
- `{`
- `#pragma omp for`
- `for (int i=0; i<MAX; i++)`
- `}`
- vytvoří tým vláken, které vykonávají blok
- vezme následující for cyklus a rozdělí jej mezi vlákna v týmu
- vlákna se připojí k hlavnímu vláknu (join)

```
int main(int argc, char* argv[]) {
 #pragma omp parallel
 {
 std::cout << "this is a line printed by thread" << omp_get_thread_num() << std::endl;

 #pragma omp for
 for (int i=0; i<10; i++)
 std::cout << "for-loop line " << i << " printed by thread" << omp_get_thread_num() << std::endl;

 }

 std::cout << "Hello from the main thread\n";
 return 0;
}
```

# OpenMP – základní způsob práce

## Blok parallel

- **#pragma omp parallel**
  - Spustí N vláken, kde N bývá (typicky) počet jader/paralelně zpracovatelných vláken (např. 4 na 2-jádrovém CPU s HT)
  - Pokud chceme počet vláken upravit, použijeme **num\_threads(<číslo>)**

```
const int thread_count = 2;
int main(int argc, char* argv[]) {
 #pragma omp parallel num_threads(thread_count)
 {
 std::cout << "this is a line printed by thread" << omp_get_thread_num() << std::endl;

 #pragma omp for
 for (int i=0; i<10; i++)
 std::cout << "for-loop line " << i << " printed by thread" << omp_get_thread_num() << std::endl;

 }

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

 return 0;
}
```

# OpenMP – základní způsob práce

## Blok parallel

- **#pragma omp parallel**

- Spustí N vláken, kde N bývá (typicky) počet jader/paralelně zpracovatelných vláken (např. 4 na 2-jádrovém CPU s HT)
- Pokud chceme počet vláken upravit, použijeme **num\_threads(<číslo>)**

```
const int thread_count = 2;
int main(int argc, char* argv[]) {
 #pragma omp parallel num_threads(thread_count)
 {
 std::cout << "this is a line printed by thread" << omp_get_thread_num() << std::endl;

 #pragma omp for
 for (int i=0; i<10; i++)
 std::cout << "for-loop line " << i << " printed by thread" << omp_get_thread_num() << std::endl;

 }

 std::cout << "Hello from the main thread\n";
 return 0;
}
```

- vypíše se 2x



- pro jedno i se vypíše pouze jednou
- každé vlákno vypíše 5

# OpenMP – základní způsob práce

## Blok for

- Rozdělí iterace na bloky, které vlákna zpracují
  - Není garantované pořadí, ve kterém se jednotlivé iterace provedou
- Existuje několik možností pro úpravu způsobu rozvržení iterací
  - Statické (**static**) – iterace se zařadí do bloků, bloky se přiřadí vláknům (výchozí možnost; bloky jsou přibližně stejně velké)
  - Dynamické (**dynamic**) – iterace se zařadí do bloků (jejich velikost lze ovlivnit, výchozí hodnota je 1), vlákna si vždy vyžádají blok ke zpracování
  - Guided – podobně jak dynamické, ale velikost bloků se postupně zmenšuje
  - Auto – bude zvoleno automaticky
  - Runtime – lze ovlivnit za běhu nastavením proměnné v prostředí
- Pokud chceme, aby se nějaká část cyklu vykonala přesně v pořadí iterací, můžeme použít modifikátor **ordered**
- Pokud máme více vnořených cyklů, můžeme je paralelizovat použitím modifikátoru **collapse(<počet\_for\_cyklů>)**

# OpenMP – základní způsob práce

## Blok for

```
#pragma omp parallel for schedule(static) num_threads(4)
for (int i=0; i<20; i++) {
 std::this_thread::sleep_for(std::chrono::milliseconds((i%2)*i*300));
 std::cout << "for-loop line " << i << " printed by thread" << omp_get_thread_num() << std::endl;
}
```

```
#pragma omp parallel for schedule(dynamic) num_threads(4)
for (int i=0; i<20; i++) {
 std::this_thread::sleep_for(std::chrono::milliseconds((i%2)*i*300));
 std::cout << "for-loop line " << i << " printed by thread" << omp_get_thread_num() << std::endl;
}
```

```
#pragma omp parallel for schedule(guided) num_threads(4)
for (int i=0; i<20; i++) {
 std::this_thread::sleep_for(std::chrono::milliseconds((i%2)*i*300));
 std::cout << "for-loop line " << i << " printed by thread" << omp_get_thread_num() << std::endl;
}
```

# OpenMP – základní způsob práce

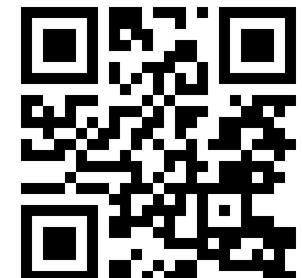
## Blok for

```
#pragma omp parallel for schedule(static) num_threads(4)
for (int i=0; i<20; i++) {
 std::this_thread::sleep_for(std::chrono::milliseconds((i%2)*i*300));
 std::cout << "for-loop line " << i << " printed by thread" << omp_get_thread_num() << std::endl;
}
```

```
#pragma omp parallel for schedule(dynamic) num_threads(4)
for (int i=0; i<20; i++) {
 std::this_thread::sleep_for(std::chrono::milliseconds((i%2)*i*300));
 std::cout << "for-loop line " << i << " printed by thread" << omp_get_thread_num() << std::endl;
}
```

```
#pragma omp parallel for schedule(guided) num_threads(4)
for (int i=0; i<20; i++) {
 std::this_thread::sleep_for(std::chrono::milliseconds((i%2)*i*300));
 std::cout << "for-loop line " << i << " printed by thread" << omp_get_thread_num() << std::endl;
}
```

- Která varianta bude nejrychlejší?



<https://goo.gl/a6BEMb>

# OpenMP – základní způsob práce

## Blok sections

- Můžeme paralelizovat pouze for cykly?
  - Můžeme paralelizovat libovolné metody/úkoly.
  - Použijeme sekce (**sections**)

# OpenMP – základní způsob práce

## Blok sections

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

const int thread_count = 2;

void method(const int& i) {
 int my_rank = omp_get_thread_num();
 int thread_count = omp_get_num_threads();
 std::cout << "Hello from method " << i << " by thread " << my_rank << std::endl;
}

int main(int argc, char* argv[]) {

#pragma omp parallel num_threads(thread_count)
{
 method(1);
#pragma omp sections
 {
#pragma omp section
 {
 method(2);
 method(3);
 }
#pragma omp section
 { method(4); }
 }
}
return 0;
}
```

- Můžeme paralelizovat pouze for cykly?
  - Můžeme paralelizovat libovolné metody/úkoly.
  - Použijeme sekce (**sections**)

# OpenMP – základní způsob práce

## Blok sections

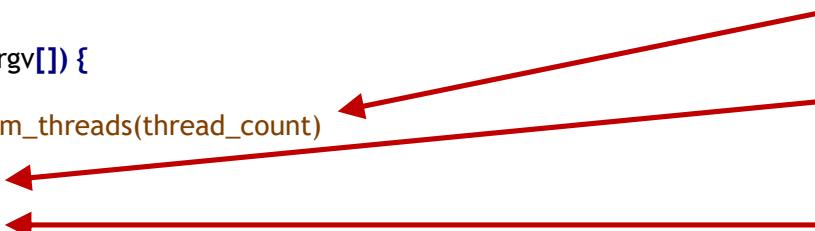
```
#include <iostream>
#include <vector>
#include "omp.h"

const int thread_count = 2;

void method(const int& i) {
 int my_rank = omp_get_thread_num();
 int thread_count = omp_get_num_threads();
 std::cout << "Hello from method " << i << " by thread " << my_rank << std::endl;
}

int main(int argc, char* argv[]) {
 #pragma omp parallel num_threads(thread_count)
 {
 method(1);
 #pragma omp sections
 {
 #pragma omp section
 {
 method(2);
 method(3);
 }
 #pragma omp section
 { method(4); }
 }
 }
 return 0;
}
```

- Můžeme paralelizovat pouze for cykly?
  - Můžeme paralelizovat libovolné metody/úkoly.
  - Použijeme sekce (**sections**)



- vytvoří se tým 2 vláken
- každé vlákno vykoná method(1)
- sekce se rozdělí mezi vlákna v týmu
- každá z method(2)-(4) se vykoná pouze 1x
- method(2) a method(3) se musí vykonat sekvenčně
- 2 sekce mohou být vykonané paralelně

# OpenMP – základní způsob práce

## Blok tasks

- Můžeme paralelizovat pouze for cykly?
  - Můžeme paralelizovat libovolné metody/úkoly.
  - Použijeme úkoly (**tasks**)

```
const int thread_count = 4;

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

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

# OpenMP – základní způsob práce

## Blok tasks

- Můžeme paralelizovat pouze for cykly?
  - Můžeme paralelizovat libovolné metody/úkoly.
  - Použijeme úkoly (**tasks**)

```
const int thread_count = 4;

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

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

- vytvoří se tým 4 vláken
- pouze 1 vlákno bude vykonávat blok *single*
- (jiné) 1 vlákno bude vykonávat task Hello

# OpenMP – základní způsob práce

## Synchronizace vláken

- Můžeme vynutit čekání vláken
  - barrier
  - některé bloky (na konci) obsahují implicitní bariéru
    - parallel, sections, for, single
    - pokud chceme implicitní bariéru zrušit, použijeme nowait

# OpenMP – základní způsob práce

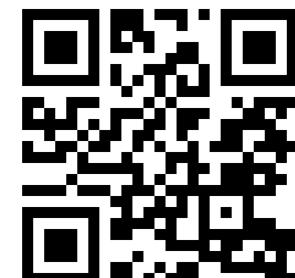
## Synchronizace vláken

- Můžeme vynutit čekání vláken
  - barrier
  - některé bloky (na konci) obsahují implicitní bariéru
    - parallel, sections, for, single
    - pokud chceme implicitní bariéru zrušit, použijeme nowait

```
std::mutex m;
#pragma omp parallel num_threads(2)
{
#pragma omp single
{
 m.lock();
#pragma omp task
{
 m.lock();
 Hello();
 m.unlock();
}
}
Hello();
m.unlock();
}
```

Co udělá tento kód?

<https://goo.gl/a6BEMb>



# OpenMP – základní způsob práce

## Synchronizace vláken

- Můžeme vynutit čekání vláken
  - barrier
  - některé bloky (na konci) obsahují implicitní bariéru
    - parallel, sections, for, single
    - pokud chceme implicitní bariéru zrušit, použijeme nowait

```
std::mutex m;
#pragma omp parallel num_threads(2)
{
 #pragma omp single
 {
 m.lock();
 #pragma omp task
 {
 m.lock(); // ←
 Hello();
 m.unlock();
 } // ←
 Hello();
 m.unlock();
 }
}
```

- dojde k deadlocku, jelikož druhé vlákno, které chce znamknout **m** čeká na první vlákno, které zámek vlastní první vlákno čeká na ukončení druhého vlákna

# OpenMP – základní způsob práce

## Synchronizace vláken

- Můžeme vynutit čekání vláken
  - barrier
  - některé bloky (na konci) obsahují implicitní bariéru
    - parallel, sections, for, single
    - pokud chceme implicitní bariéru zrušit, použijeme nowait

```
std::mutex m;
#pragma omp parallel num_threads(2)
{
 #pragma omp single nowait
 {
 m.lock();
 #pragma omp task
 {
 m.lock();
 Hello();
 m.unlock();
 }
 }
 Hello();
 m.unlock();
}
```

# OpenMP – přístup k proměnným

## Privátní a sdílené proměnné

- V rámci bloku `omp parallel` můžeme určovat viditelnost proměnných
  - `shared(<proměnná1>, <proměnná2>,...)`
    - sdílené proměnné
  - `private(<proměnná1>, <proměnná2>,...)`
    - privátní proměnné
    - vytvoří se nenainicializovaná lokální kopie

# OpenMP – přístup k proměnným

## Privátní a sdílené proměnné

- V rámci bloku `omp parallel` můžeme určovat viditelnost proměnných
  - `shared(<proměnná1>, <proměnná2>,...)`
    - sdílené proměnné
  - `private(<proměnná1>, <proměnná2>,...)`
    - privátní proměnné
    - vytvoří se nenainicializovaná lokální kopie

```
const int thread_count = 2;
int main(int argc, char* argv[]) {
 int a = 42, b = 1;
#pragma omp parallel num_threads(thread_count) shared(a) private(b)
 {
 b = omp_get_thread_num() + 2;
#pragma omp critical
 {
 a = a / b;
 std::cout << "Variable 'b' = " << b << " by thread " << (omp_get_thread_num()) << std::endl;
 std::cout << "Variable 'a' = " << a << " by thread " << (omp_get_thread_num()) << std::endl;
 }
 }
 std::cout << "Variable 'a' = " << a << " after omp " << std::endl;
 std::cout << "Variable 'b' = " << b << " after omp " << std::endl;
 return 0;
}
```

# OpenMP – přístup k proměnným

## Privátní a sdílené proměnné

- V rámci bloku `omp parallel` můžeme určovat viditelnost proměnných
  - `firstprivate(<proměnná1>, <proměnná2>,...)`
    - lokální kopie proměnné se nainicializuje dle původní hodnoty

```
const int thread_count = 2;
int main(int argc, char* argv[]) {
 int a = 42, b = 1;
#pragma omp parallel num_threads(thread_count) shared(a) firstprivate(b)
 {
 b = omp_get_thread_num() + b + 2;
#pragma omp critical
 {
 a = a / b;
 std::cout << "Variable 'b' = " << b << " by thread " << (omp_get_thread_num()) << std::endl;
 std::cout << "Variable 'a' = " << a << " by thread " << (omp_get_thread_num()) << std::endl;
 }
 }
 std::cout << "Variable 'a' = " << a << " after omp " << std::endl;
 std::cout << "Variable 'b' = " << b << " after omp " << std::endl;
 return 0;
}
```

# OpenMP – přístup k proměnným

## Privátní a sdílené proměnné

- V rámci OpenMP můžeme využít známých technik pro synchronizaci přístupu k sdíleným proměnným
  - kritické sekce - #pragma omp critical([<název\_sekce>])
    - pouze 1 vlákno může být v kritické sekci
  - zámky
    - existuje `omp_lock_t`
    - lze použít standardní (C++11) mutexy
  - atomické operace
    - `#pragma omp atomic`
  - `flush(<proměnná1>, ...)`
    - zabezpečí synchronizaci sdílených proměnných
    - před každým čtením, po každém zápisu
    - pomalé z důvodu zabezpečení konzistence

# OpenMP – přístup k proměnným

## Privátní a sdílené proměnné

- pro agregaci výsledků lze použít **reduction variable**
  - **#pragma omp parallel reduction(<operace>:<proměnná>)**
  - přístup ke sdílené proměnné
  - zabezpečí atomickou exekuci, nicméně pouze jedné předem definované operace
  - vhodné pro agregaci parciálních výsledků dílčích úkolů

# OpenMP – přístup k proměnným

## Privátní a sdílené proměnné

- pro agregaci výsledků lze použít **reduction variable**
  - **#pragma omp parallel reduction(<operace>:<proměnná>)**
  - přístup ke sdílené proměnné
  - zabezpečí atomickou exekuci, nicméně pouze jedné předem definované operace
  - vhodné pro agregaci parciálních výsledků dílčích úkolů

```
long x=0;
#pragma omp parallel for num_threads(thread_count) reduction(+:x)
 for (int i=0; i<SIZE; i++) {
 x += vector_to_sum[i];
 }
```

# OpenMP – přístup k proměnným

## Privátní a sdílené proměnné

- pro agregaci výsledků lze použít **reduction variable**
  - **#pragma omp parallel reduction(<operace>:<proměnná>)**
  - přístup ke sdílené proměnné
  - zabezpečí atomickou exekuci, nicméně pouze jedné předem definované operace
  - vhodné pro agregaci parciálních výsledků dílčích úkolů

```
long x=0;
#pragma omp parallel for num_threads(thread_count) reduction(+:x)
 for (int i=0; i<SIZE; i++) {
 x += vector_to_sum[i];
 }
```

- vytvoří lokální kopii proměnné
- na konci použije definovanou operaci pro sjednocení parciálních výsledků ze všech vláken

# OpenMP – přístup k proměnným

## Privátní a sdílené proměnné

- pro agregaci výsledků lze použít **reduction variable**
  - co když chceme agregovat výsledky v poli (případně v jiné datové struktuře)
  - můžeme si definovat vlastní operace redukce
    - `#pragma omp declare reduction(name:type:expression) initializer(expression)`
    - např. se může hodit aggregace (suma) čísel ve vektoru

```
#pragma omp declare reduction(vec_int_plus : std::vector<int> : \
 std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>()) \
 initializer(omp_priv = omp_orig)
```

# OpenMP – přístup k proměnným

## Privátní a sdílené proměnné

- pro agregaci výsledků lze použít **reduction variable**
  - co když chceme agregovat výsledky v poli (případně v jiné datové struktuře)
  - můžeme si definovat vlastní operace redukce
    - `#pragma omp declare reduction(name:type:expression) initializer(expression)`
    - např. se může hodit agregace (suma) čísel ve vektoru

```
#pragma omp declare reduction(vec_int_plus : std::vector<int> : \
 std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>()) \
 initializer(omp_pri = omp_orig)
```

výslední (výstupní) proměnná

lokální kopie proměnné (vstupní z hlediska redukce)

# OpenMP – přístup k proměnným

## Privátní a sdílené proměnné

- pro agregaci výsledků lze použít **reduction variable**
  - co když chceme agregovat výsledky v poli (případně v jiné datové struktuře)
  - můžeme si definovat vlastní operace redukce
    - `#pragma omp declare reduction(name:type:expression) initializer(expression)`
    - např. se může hodit agregace (suma) čísel ve vektoru

```
#pragma omp declare reduction(vec_int_plus : std::vector<int> : \
 std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>())) \
 initializer(omp_pri = omp_orig)
```

výslední (výstupní) proměnná

inicializace výstupní proměnné

lokální kopie proměnné (vstupní z hlediska redukce)

původní hodnota proměnné, na kterou aplikujeme redukci

# OpenMP – histogram

```
long nolocks(std::vector<int>& vector, std::vector<int>& histogram) {
 int j;
#pragma omp parallel for private(j) shared(vector,histogram) num_threads(thread_count)
 for (int i=0; i<SIZE; i++) {
 j = vector[i] % PARTS;
 histogram[j]++;
 }
}
```

```
long local_locks(std::vector<int>& vector, std::vector<int>& histogram) {
 int j;
#pragma omp parallel for private(j) shared(vector,histogram) num_threads(thread_count)
 for (int i=0; i<SIZE; i++) {
 j = vector[i] % PARTS;
 hist_part_mutex[j].lock();
 histogram[j]++;
 hist_part_mutex[j].unlock();
 }
}
```

# OpenMP – histogram

```
long local_atomic(std::vector<int>& vector, std::vector<int>& histogram) {
 int j;
#pragma omp parallel for private(j) shared(vector,histogram) num_threads(thread_count)
 for (int i=0; i<SIZE; i++) {
 j = vector[i] % PARTS;
#pragma omp atomic
 histogram[j]++;
 }
}
```

```
long local_reduction(std::vector<int>& vector, std::vector<int>& histogram) {
#pragma omp declare reduction(vec_int_plus : std::vector<int> : \
 std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>()) \
 initializer(omp_priv = omp_orig)
 int j;
#pragma omp parallel for private(j) shared(vector) num_threads(thread_count) reduction(vec_int_plus:histogram)
 for (int i=0; i<SIZE; i++) {
 j = vector[i] % PARTS;
 histogram[j]++;
 }
}
```



# Paralelní a distribuované výpočty (B4B36PDV)

**Branislav Bošanský, Michal Jakob**

[bosansky@fel.cvut.cz](mailto:bosansky@fel.cvut.cz)

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

# Dnešní přednáška

## Motivace



# Dnešní přednáška

## Konkurentní datové struktury

Vícero vláken chce přistupovat ke společné datové struktuře

- např. zásobník, fronta, spojový seznam
- binární vyhledávací strom, jiné vyhledávací stromy
- haldy
- ...

# Dnešní přednáška

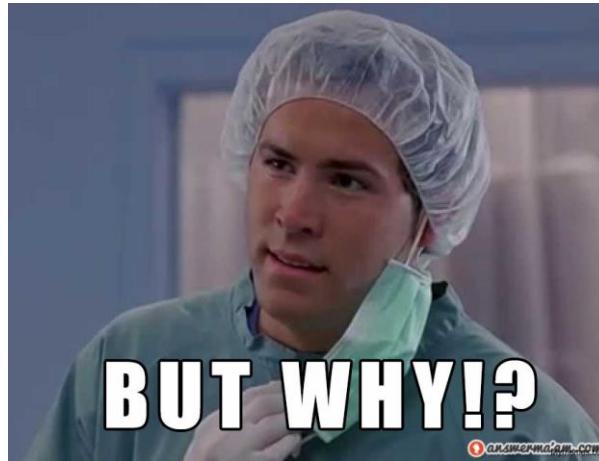
## Konkurentní datové struktury

- Potřebujeme zabezpečit korektní operace s datovou strukturou
  - Vlákna chtějí současně vkládat, mazat, nebo vyhledávat

# Dnešní přednáška

## Konkurentní datové struktury

- Potřebujeme zabezpečit korektní operace s datovou strukturou
  - Vlákna chtějí současně vkládat, mazat, nebo vyhledávat



- Máme zámky, strukturu zamkneme a máme garantovanou konzistenci ...

# Dnešní přednáška

## Konkurentní datové struktury

- Potřebujeme zabezpečit korektní operace s datovou strukturou
  - Vlákna chtějí současně vkládat, mazat, nebo vyhledávat
- Máme zámky, strukturu zamkneme a máme garantovanou konzistenci ...
- Chceme navrhnut co nejfektivnější práci více vláken nad společnou datovou strukturou
- Dnes si ukážeme, jak můžeme konzistenci zajistit bez zámků (tzv. lock-free datové struktury)

# Konkurentní datové struktury

Co chceme dosáhnout?

- Hlavní myšlenka

Vlákna optimisticky předpokládají, že vše bude v pořádku  
(modifikace DS budou konzistentní)

- ... ale nemůžeme se na to spolehnout, takže

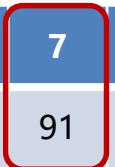
V případě detekce nekonzistence ji vlákno vyřeší opraví

# Konkurentní datové struktury

## Příklad 1 – vyhledání maxima

- Vyhledání maxima v seznamu čísel
  - Chci najít maximální hodnotu a index na kterém se nachází
  - V případě rovnosti, chci co možná největší index

| 1 | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 |
|---|----|----|----|----|----|----|----|----|----|
| 3 | 12 | 42 | 91 | 74 | 24 | 91 | 66 | 19 | 32 |



- Společná datová struktura
  - Dvě čísla – maximální hodnota & index

# Konkurentní datové struktury

## Příklad 1 – vyhledání maxima

- Vyhledání maxima v seznamu čísel
  - Chci najít maximální hodnotu a index na kterém se nachází
  - V případě rovnosti, chci co možná největší index

| 1 | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 |
|---|----|----|----|----|----|----|----|----|----|
| 3 | 12 | 42 | 91 | 74 | 24 | 91 | 66 | 19 | 32 |



- Společná datová struktura
  - Dvě čísla – maximální hodnota & index
  - Jedno číslo – index aktuální maximální hodnoty
- Jak na to?

# Konkurentní datové struktury

## Příklad 1

- Vyhledání maxima v seznamu čísel
  - Chci najít maximální hodnotu a index na kterém se nachází
  - V případě rovnosti, chci co možná největší index
- První řešení - zámky

| 1 | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 |
|---|----|----|----|----|----|----|----|----|----|
| 3 | 12 | 42 | 91 | 74 | 24 | 91 | 66 | 19 | 32 |



# Konkurentní datové struktury

## Příklad 1

- Vyhledání maxima v seznamu čísel
  - Chci najít maximální hodnotu a index na kterém se nachází
  - V případě rovnosti, chci co možná největší index
- První řešení - zámky

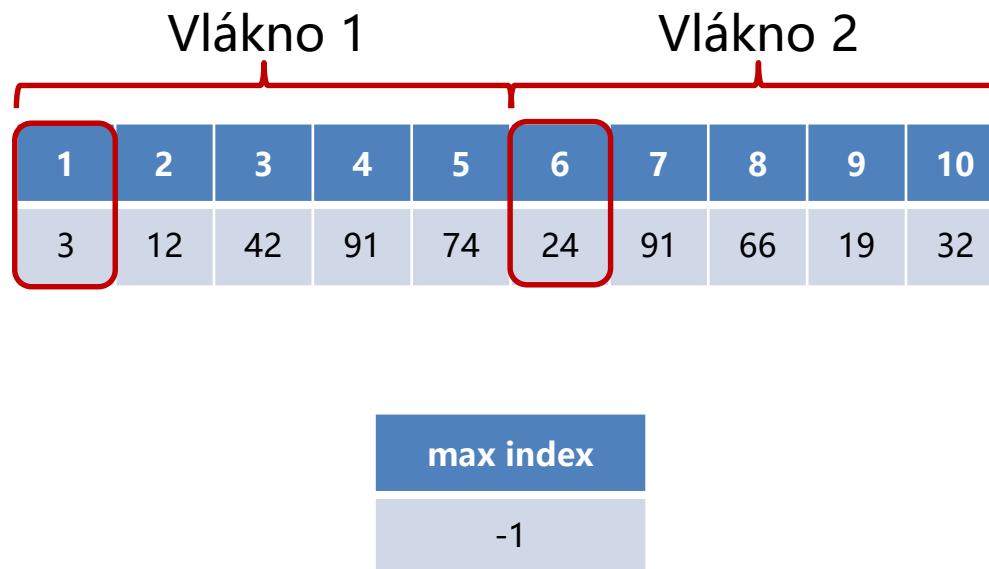
| Vlákno 1 |    |    |    |    | Vlákno 2 |    |    |    |    |
|----------|----|----|----|----|----------|----|----|----|----|
| 1        | 2  | 3  | 4  | 5  | 6        | 7  | 8  | 9  | 10 |
| 3        | 12 | 42 | 91 | 74 | 24       | 91 | 66 | 19 | 32 |

max index  
-1

# Konkurentní datové struktury

## Příklad 1

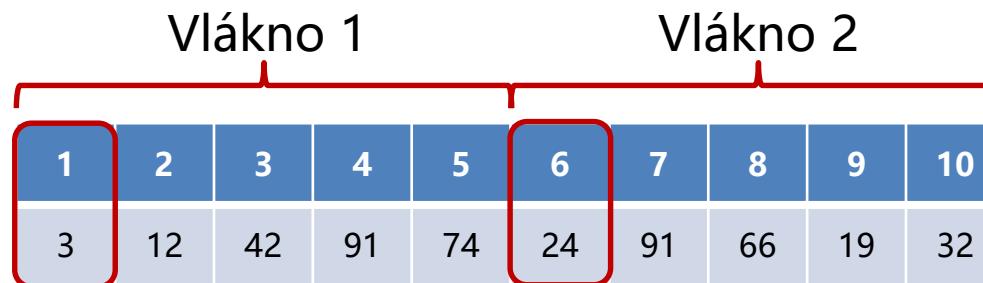
- Vyhledání maxima v seznamu čísel
    - Chci najít maximální hodnotu a index na kterém se nachází
    - V případě rovnosti, chci co možná největší index
  - První řešení - zámky



# Konkurentní datové struktury

## Příklad 1

- Vyhledání maxima v seznamu čísel
  - Chci najít maximální hodnotu a index na kterém se nachází
  - V případě rovnosti, chci co možná největší index
- První řešení - zámky

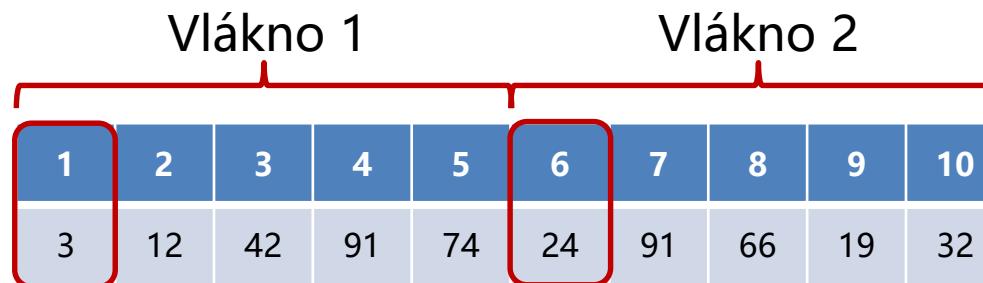


- vlákno 1 porovná hodnotu
- pokud je aktuální hodnota větší
  - zamkne max index
  - provede úpravu
  - odemkne max index

# Konkurentní datové struktury

## Příklad 1

- Vyhledání maxima v seznamu čísel
  - Chci najít maximální hodnotu a index na kterém se nachází
  - V případě rovnosti, chci co možná největší index
- První řešení - zámky



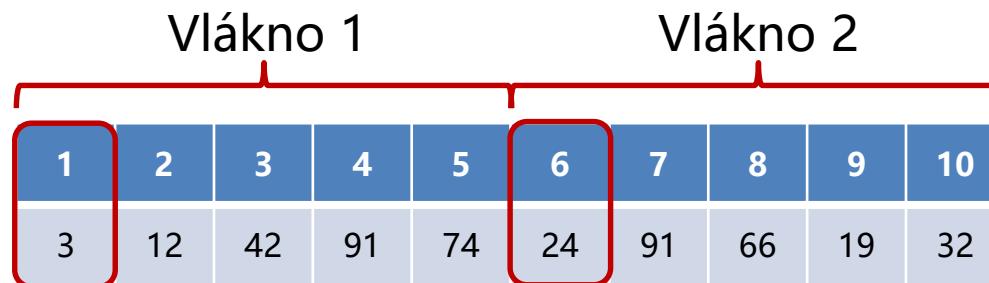
- vlákno 2 porovná hodnotu
- pokud je aktuální hodnota větší
  - zamkne max index
  - provede úpravu
  - odemkne max index



# Konkurentní datové struktury

## Příklad 1

- Vyhledání maxima v seznamu čísel
  - Chci najít maximální hodnotu a index na kterém se nachází
  - V případě rovnosti, chci co možná největší index
- První řešení - zámky



- vlákno 2 porovná hodnotu
- pokud je aktuální hodnota větší
  - zamkne max index
  - provede úpravu
  - odemkne max index

Jak to bude fungovat?

# Konkurenční datové struktury

## Příklad 1

- Může vzniknout nekonzistence

| Vlákno 1 |    |    |    |    | Vlákno 2 |    |    |    |    |
|----------|----|----|----|----|----------|----|----|----|----|
| 1        | 2  | 3  | 4  | 5  | 6        | 7  | 8  | 9  | 10 |
| 3        | 12 | 42 | 91 | 74 | 24       | 91 | 66 | 19 | 32 |

- vlákno 1 porovná hodnotu
- pokud je aktuální hodnota větší
  - zamkne max index
  - provede úpravu na hodnotu 1
  - odemkne max index
- vlákno 2 porovná hodnotu
- pokud je aktuální hodnota větší
  - zamkne max index
  - provede úpravu na hodnotu 6
  - odemkne max index



# Konkurenční datové struktury

## Příklad 1

- Může vzniknout nekonzistence

| Vlákno 1 |    |    |    |    | Vlákno 2 |    |    |    |    |
|----------|----|----|----|----|----------|----|----|----|----|
| 1        | 2  | 3  | 4  | 5  | 6        | 7  | 8  | 9  | 10 |
| 3        | 12 | 42 | 91 | 74 | 24       | 91 | 66 | 19 | 32 |

- vlákno 1 porovná hodnotu
- pokud je aktuální hodnota větší
  - zamkne max index
  - provede úpravu na hodnotu 1
  - odemkne max index
- vlákno 2 porovná hodnotu
- pokud je aktuální hodnota větší
  - zamkne max index
  - provede úpravu na hodnotu 6
  - odemkne max index

Výsledek může být nesprávný

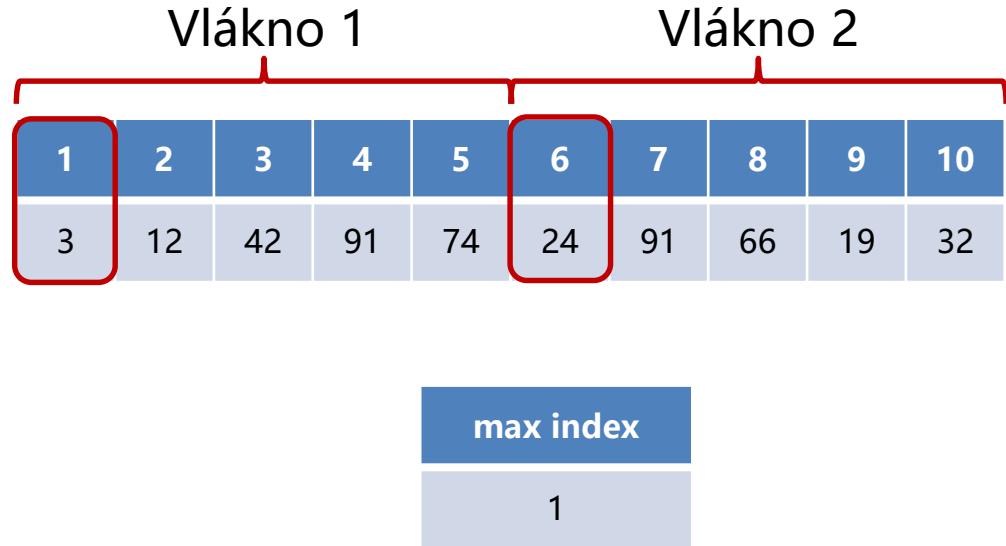


# Konkurentní datové struktury

## Příklad 1

- možné řešení:
  - vlákna budou zamykat max index před kontrolou – pomalé ☹
  - po získání zámku vlákno opět zkонтroluje jestli je update aktuální

- vlákno 1 porovná hodnotu
- pokud je aktuální hodnota větší
  - zamkne max index
  - zkонтroluje, jestli je aktuální hodnota stále větší
  - provede úpravu na hodnotu 1
  - odemkne max index



# Konkurenční datové struktury

## Příklad 1

- Jde to i bez zámků?
- K nekonzistenci může dojít mezi kontrolou jestli je aktuální hodnota větší než v maximu a případnou výměnou
  - Nechť je aktuální hodnota **max index == 2** a vlákno 2 testuje podmínu
  - Výměnu vykoná pouze tehdy, pokud je **max index** pořád 2

| Vlákno 1 |    | Vlákno 2 |    |
|----------|----|----------|----|
| 1        | 2  | 3        | 4  |
| 3        | 12 | 42       | 91 |

|    |    |
|----|----|
| 5  | 6  |
| 74 | 24 |

|    |    |    |    |
|----|----|----|----|
| 7  | 8  | 9  | 10 |
| 91 | 66 | 19 | 32 |

Můžeme použít atomické proměnné

max index

2

# Compare and Swap

- Atomická operace **compare and swap** (CAS)
  - Atomicky porovná jestli hodnota proměnné odpovídá očekávané hodnotě a pokud ano, provede změnu na novou hodnotu
- V C++, `compare_exchange_strong(expected, new)`

```
std::atomic_int atomic_max_index;

void find_max_cas(std::vector<int> & vector) {
#pragma omp parallel for num_threads(thread_count) shared(vector, atomic_max_index)
 for (int i=0; i<SIZE; ++i) {
 int tmp_index = atomic_max_index.load();
 while ((vector[i] > vector[tmp_index] || (vector[i] == vector[tmp_index] && i > tmp_index)) &&
 !atomic_max_index.compare_exchange_strong(tmp_index,i)) {
 tmp_index = atomic_max_index.load();
 }
 }
}
```

# Compare and Swap

- Atomická operace **compare and swap** (CAS)
  - Atomicky porovná jestli hodnota proměnné odpovídá očekávané hodnotě a pokud ano, provede změnu na novou hodnotu
- V C++, `compare_exchange_strong(expected, new)`

podmínka pro maximum

```
std::atomic_int atomic_max_index;

void find_max_cas(std::vector<int> & vector) {
#pragma omp parallel for num_threads(thread_count) shared(vector, atomic_max_index)
 for (int i=0; i<SIZE; ++i) {
 int tmp_index = atomic_max_index.load();
 while ((vector[i] > vector[tmp_index] || (vector[i] == vector[tmp_index] && i > tmp_index)) &&
 !atomic_max_index.compare_exchange_strong(tmp_index,i)) {
 tmp_index = atomic_max_index.load();
 }
 }
}

hodnota indexu
```

# Compare and Swap

## Příklad 1

- Atomická operace **compare and swap** (CAS)
  - Atomicky porovná jestli hodnota proměnné odpovídá očekávané hodnotě a pokud ano, provede změnu na novou hodnotu
- V C++, `compare_exchange_strong(expected, new)`

```
std::atomic_int atomic_max_index;

void find_max_cas(std::vector<int> & vector) {
#pragma omp parallel for num_threads(thread_count) shared(vector, atomic_max_index)
 for (int i=0; i<SIZE; ++i) {
 int tmp_index = atomic_max_index.load();
 while ((vector[i] > vector[tmp_index] || (vector[i] == vector[tmp_index] && i > tmp_index)) &&
 !atomic_max_index.compare_exchange_strong(tmp_index,i)) {
 tmp_index = atomic_max_index.load();
 }
 }
}
```

compare and swap – pokud je v proměnné **atomic\_max\_index** hodnota **tmp\_index** (kterou jsme použili), tak provedeme update

# Compare and Swap

## Příklad 1

```
std::atomic_int atomic_max_index;

void find_max_cas(std::vector<int> & vector) {
#pragma omp parallel for num_threads(thread_count) shared(vector, atomic_max_index)
 for (int i=0; i<SIZE; ++i) {
 int tmp_index = atomic_max_index.load();
 while ((vector[i] > vector[tmp_index] || (vector[i] == vector[tmp_index] && i > tmp_index)) &&
 !atomic_max_index.compare_exchange_strong(tmp_index,i)) {
 tmp_index = atomic_max_index.load();
 }
 }
}
```



výsledek je **true** pokud se  
operace povede, **false** v opačném  
případě

# Compare and Swap

## Příklad 1

```
std::atomic_int atomic_max_index;

void find_max_cas(std::vector<int> & vector) {
#pragma omp parallel for num_threads(thread_count) shared(vector, atomic_max_index)
 for (int i=0; i<SIZE; ++i) {
 int tmp_index = atomic_max_index.load();
 while ((vector[i] > vector[tmp_index] || (vector[i] == vector[tmp_index] && i > tmp_index)) &&
 !atomic_max_index.compare_exchange_strong(tmp_index,i)) {
 tmp_index = atomic_max_index.load();
 }
 }
}
```

Pokud je výsledek **false**, jiné vlákno mezičasem změnilo **atomic\_max\_index**. Musíme aktualizovat hodnotu **tmp\_index** a provést kontrolu znovu.

výsledek je **true** pokud se operace povede, **false** v opačném případě

# Compare and Swap

## Příklad 1

```
std::atomic_int atomic_max_index;

void find_max_cas(std::vector<int> & vector) {
#pragma omp parallel for num_threads(thread_count) shared(vector, atomic_max_index)
 for (int i=0; i<SIZE; ++i) {
 int tmp_index = atomic_max_index.load();
 while ((vector[i] > vector[tmp_index] || (vector[i] == vector[tmp_index] && i > tmp_index)) &&
 !atomic_max_index.compare_exchange_strong(tmp_index,i)) {
 tmp_index = atomic_max_index.load();
 }
 }
}
```

Pokud je výsledek **false**, jiné vlákno mezičasem změnilo **atomic\_max\_index**. Musíme aktualizovat hodnotu **tmp\_index** a provést kontrolu znovu.

výsledek je **true** pokud se operace povede, **false** v opačném případě

**Kontrola je ve while cyklu!**  
(nekonzistence se může vyskytnout opakovaně)

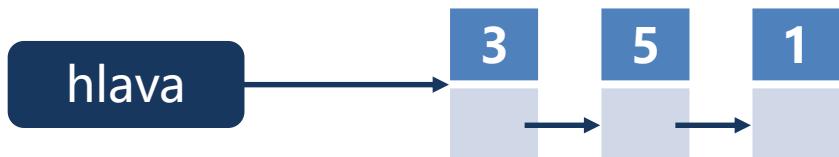
# Konkurentní datové struktury

## Příklad 2 – zásobník

### Zásobník

```
struct Node {
 int value = 0;
 Node* successor = nullptr;

 Node(int _value, Node* _successor) : value(_value), successor(_successor) {}
};
```

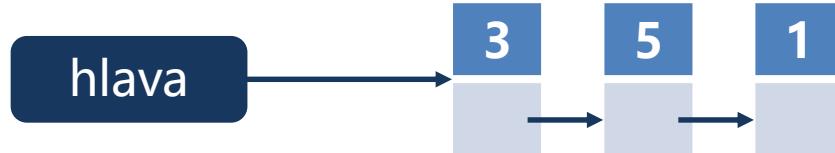


Kritické místo je při vkládání a odebírání do/z vrcholu zásobníku

# Konkurentní datové struktury

## Příklad 2

Zásobník



Řešení pomocí zámků

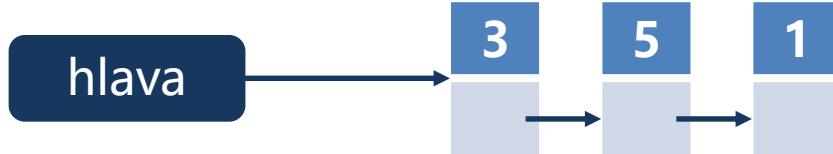
```
void add_to_stack_locks(int new_value) {
 m.lock();
 head = new Node(new_value, head);
 m.unlock();
}

int pop_from_stack_locks() {
 m.lock();
 if (head == nullptr) {
 m.unlock();
 throw std::out_of_range("The stack is empty.");
 return -1;
 } else {
 Node* tmp = head;
 int val = head->value;
 head = head->successor;
 delete tmp;
 m.unlock();
 return val;
 }
}
```

# Konkurenční datové struktury

## Příklad 2

Řešení pomocí atomických proměnných



```
std::atomic<Node*> head2;

void add_to_stack_cas(int new_value) {
 Node* p = new Node(new_value, head2.load());
 while (!head2.compare_exchange_strong(p->successor, p)) {
 p->successor = head2.load();
 }
}

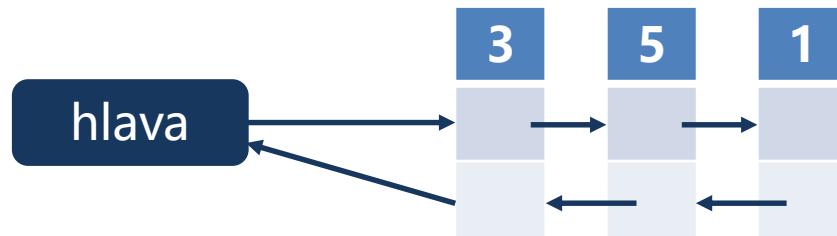
int pop_from_stack_cas() {
 if (head2.load() == nullptr) {
 throw std::out_of_range("The stack is empty.");
 return -1;
 } else {
 Node* h = head2.load();
 while (!head2.compare_exchange_strong(h, h->successor)) {
 h = head2.load();
 }
 int val = h->value;
 delete h;
 return val;
 }
}
```

Jak to bude fungovat?

# Konkurentní datové struktury

## Příklad 3

- Obousměrný spojový seznam



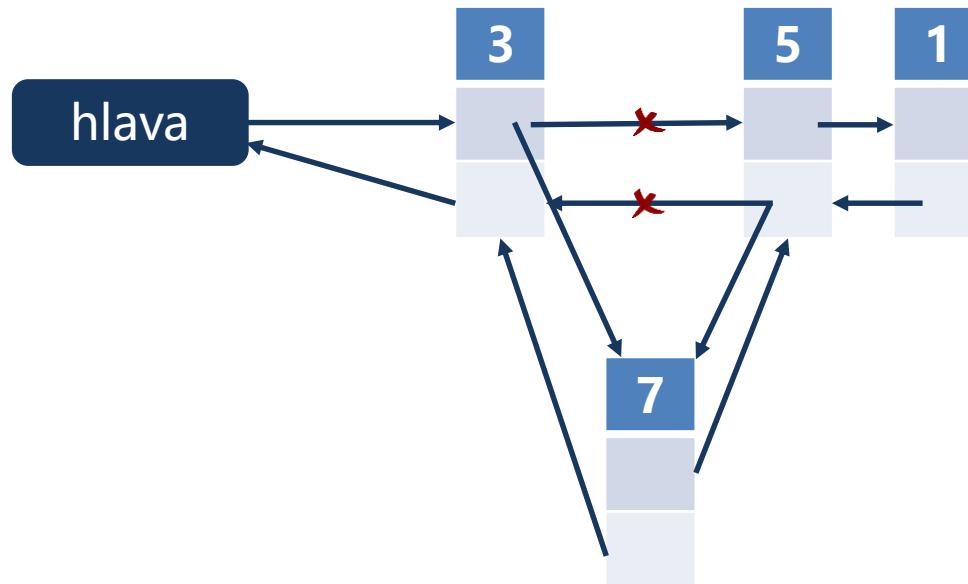
Operace přidání



# Konkurentní datové struktury

## Příklad 3

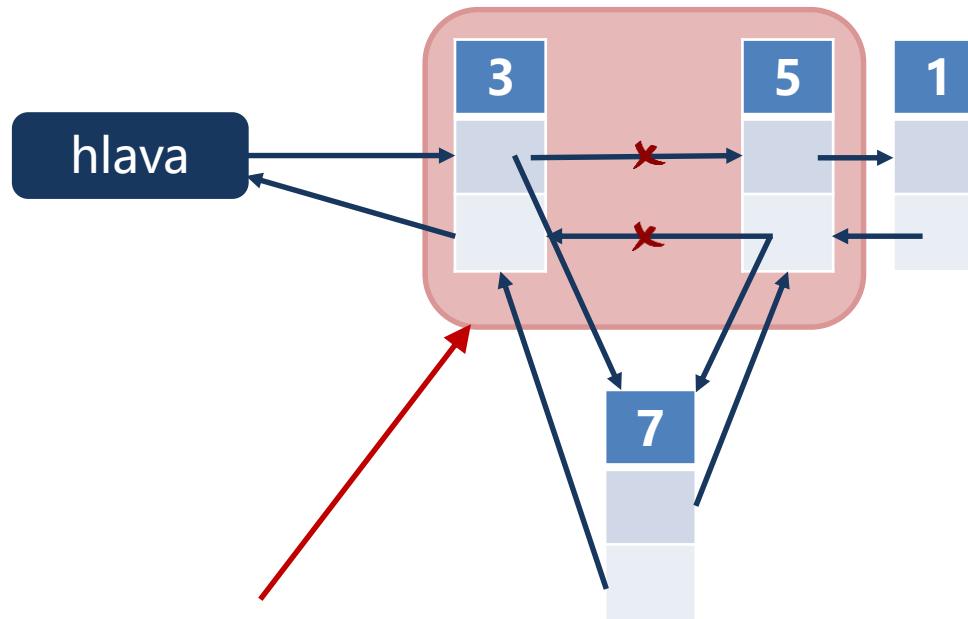
- Operace přidání pro obousměrný spojový seznam



# Konkurentní datové struktury

## Příklad 3

- Operace přidání pro obousměrný spojový seznam

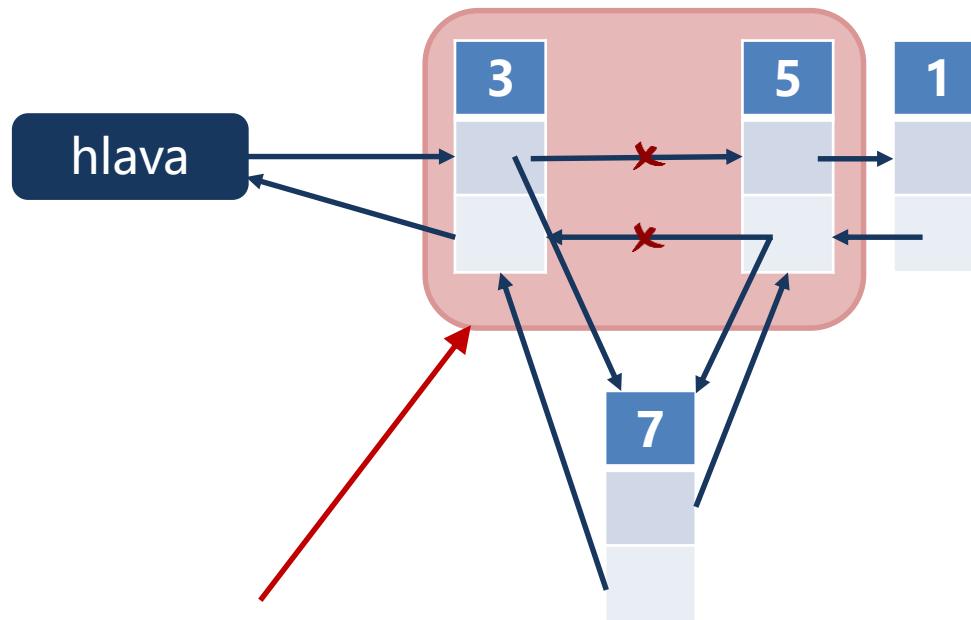


musíme zamknout oba prvky

# Konkurentní datové struktury

## Příklad 3

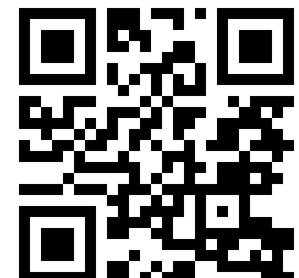
- Operace přidání pro obousměrný spojový seznam



musíme zamknout oba prvky

Musíme zamykat oba současně nebo stačí vždy nejdřív prvek blíž k hlavě a pak jeho následovníka?

<https://goo.gl/a6BEMb>



# Konkurentní datové struktury

## Příklad 3

- Řešení pomocí zámků

```
struct Node {
 std::mutex m;
 int value = 0;
 Node* successor = nullptr;
 Node* predecessor = nullptr;
 Node(int _value, Node* _predecessor, Node* _successor) :
 value(_value), predecessor(_predecessor), successor(_successor) {}
};
```

# Konkurentní datové struktury

## Příklad 3

- Řešení pomocí zámků

```
Node* add_to_list_after(Node* _previous_node, int _new_value) {
 assert (_previous_node != nullptr);
 _previous_node->m.lock();

 Node* new_successor = _previous_node->successor;
 bool is_there_successor = new_successor != nullptr;

 if (is_there_successor) {
 new_successor->m.lock();
 }

 Node* new_node = new Node(_new_value, _previous_node, new_successor);

 if (is_there_successor)
 _previous_node->successor->predecessor = new_node;
 _previous_node->successor = new_node;

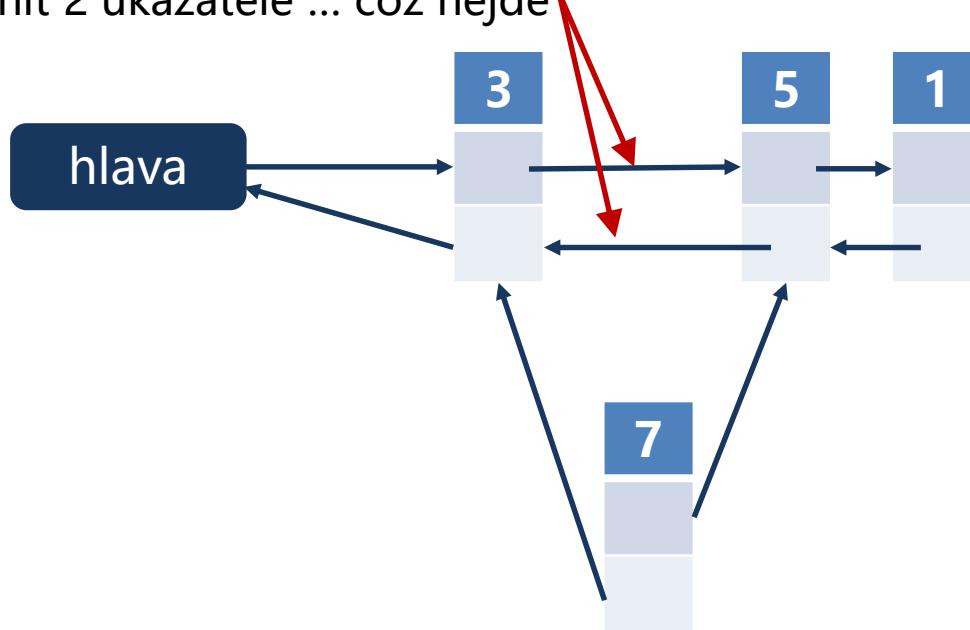
 if (is_there_successor)
 new_successor->m.unlock();
 _previous_node->m.unlock();
 return new_node;
}
```

# Konkurenční datové struktury

## Příklad 3

- Jak řešit pomocí atomických operací?

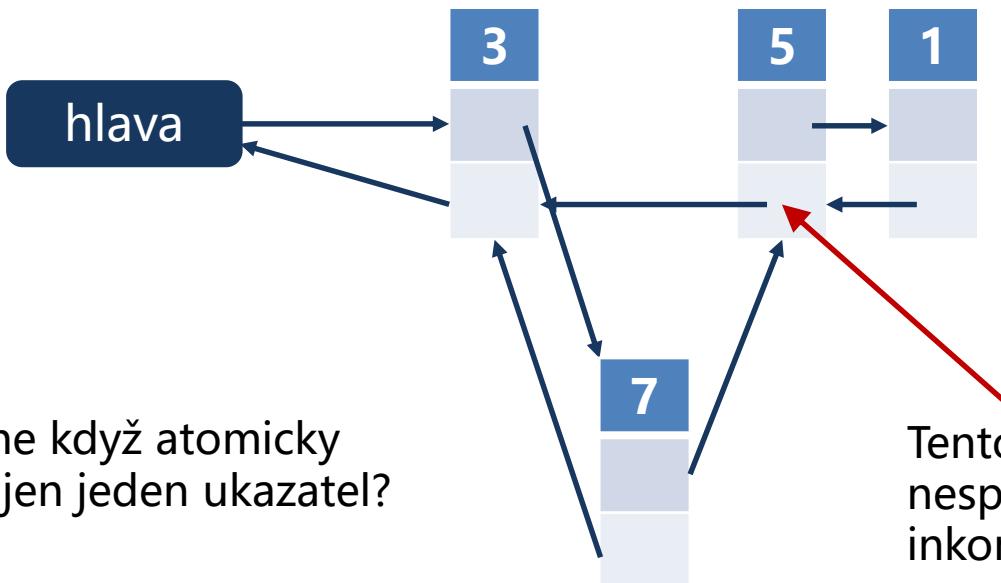
potřebovali bychom atomicky změnit 2 ukazatele ... což nejde



# Konkurenční datové struktury

## Příklad 3

- Jak řešit pomocí atomických operací?



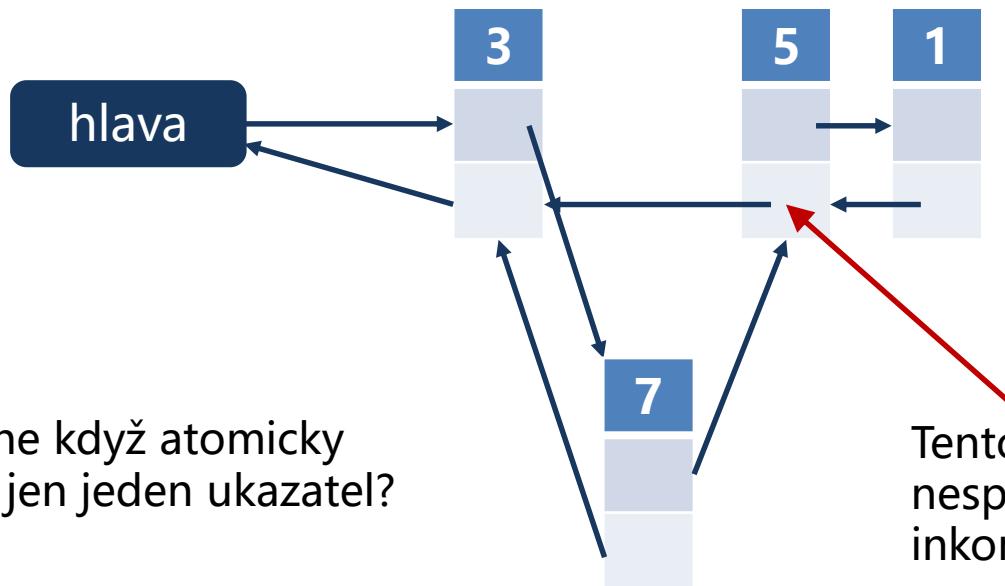
Co se stane když atomicky změníme jen jeden ukazatel?

Tento ukazatel je nesprávný. Kdy nám tato inkonzistence může vadit?

# Konkurenční datové struktury

## Příklad 3

- Jak řešit pomocí atomických operací?



Co se stane když atomicky změníme jen jeden ukazatel?

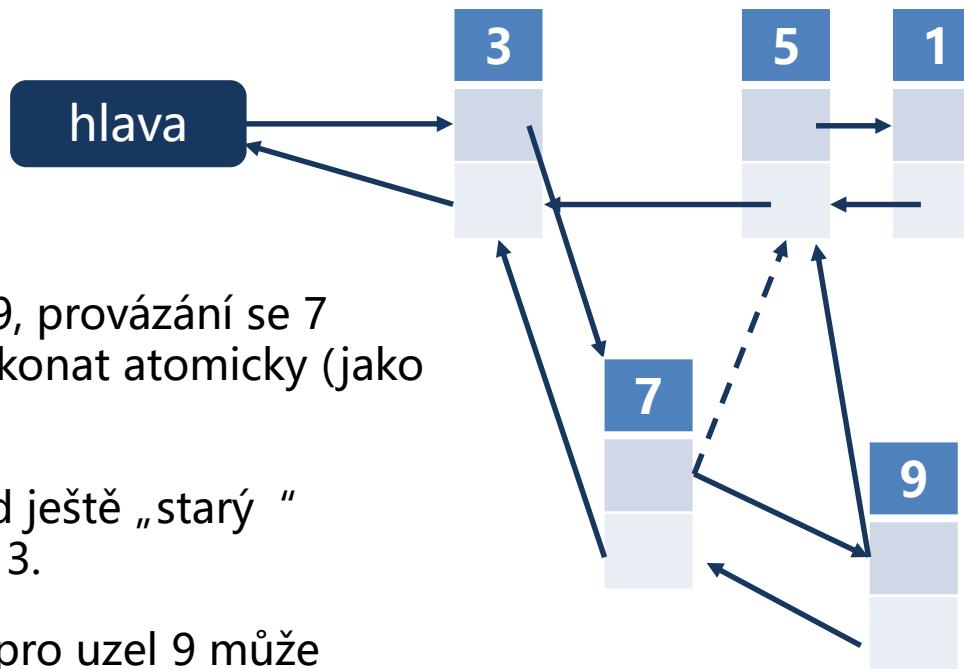
Tento ukazatel je nesprávný. Kdy nám tato inkonzistence může vadit?

Když chceme přidávat mezi 7 a 5.

# Konkurentní datové struktury

## Příklad 3

- Jak řešit pomocí atomických operací?



Přidáváme 9, provázání se 7 můžeme vykonat atomicky (jako předtím).

U 5 je pořád ještě „starý“ ukazatel na 3.

Dokončení pro uzel 9 může počkat, až bude v 5 správný ukazatel (na 7).

# Konkurentní datové struktury

## Příklad 3

- Řešení pomocí CAS

```
struct AtomicNode {
 int value = 0;
 std::atomic<AtomicNode*> successor;
 std::atomic<AtomicNode*> predecessor;

 AtomicNode(int value) {
 successor.store(nullptr);
 predecessor.store(nullptr);
 }

 AtomicNode(int _value, AtomicNode* _predecessor, AtomicNode* _successor) : value(_value) {
 successor.store(_successor);
 predecessor.store(_predecessor);
 }
};
```

# Konkurentní datové struktury

## Příklad 3

- Řešení pomocí CAS

```
AtomicNode* atomic_add_to_list_after(AtomicNode* _previous_node, int _new_value) {
 assert (_previous_node != nullptr);

 AtomicNode* old_successor = _previous_node->successor;
 AtomicNode* new_node = new AtomicNode(_new_value, _previous_node, old_successor);

 while (!_previous_node->successor.compare_exchange_strong(old_successor, new_node)) {
 old_successor = _previous_node->successor;
 new_node->successor.store(old_successor);
 }

 if (old_successor != nullptr) {
 while (!old_successor->predecessor.compare_exchange_strong(_previous_node, new_node))
 ;
 }

 return new_node;
}
```

# Konkurentní datové struktury

## Příklad 3

- Řešení pomocí CAS

```
AtomicNode* atomic_add_to_list_after(AtomicNode* _previous_node, int _new_value) {
 assert (_previous_node != nullptr);

 AtomicNode* old_successor = _previous_node->successor;
 AtomicNode* new_node = new AtomicNode(_new_value, _previous_node, old_successor);

 while (!_previous_node->successor.compare_exchange_strong(old_successor, new_node)) {
 old_successor = _previous_node->successor;
 new_node->successor.store(old_successor);
 }

 if (old_successor != nullptr) {
 while (!old_successor->predecessor.compare_exchange_strong(_previous_node, new_node))
 ;
 }

 return new_node;
}
```

Změna ukazatele v  
prvním prvku (před  
vkládaným uzlem).

# Konkurentní datové struktury

## Příklad 3

- Řešení pomocí CAS

```
AtomicNode* atomic_add_to_list_after(AtomicNode* _previous_node, int _new_value) {
 assert (_previous_node != nullptr);

 AtomicNode* old_successor = _previous_node->successor;
 AtomicNode* new_node = new AtomicNode(_new_value, _previous_node, old_successor);

 while (!_previous_node->successor.compare_exchange_strong(old_successor, new_node)) {
 old_successor = _previous_node->successor;
 new_node->successor.store(old_successor);
 }

 if (old_successor != nullptr) {
 while (!old_successor->predecessor.compare_exchange_strong(_previous_node, new_node))
 ;
 }

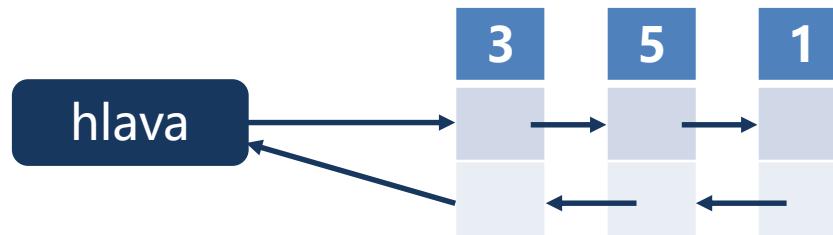
 return new_node;
}
```

Změna ukazatele v  
druhého prvku (za  
vkládaným uzlem).

# Konkurenční datové struktury

## Příklad 3

- Obousměrný spojový seznam



Operace mazání – komplexnější

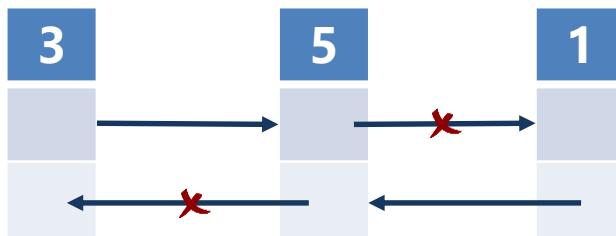
Musíme označit které uzly (ukazatele) budou smazány

# Konkurenční datové struktury

## Příklad 3

- Mazání v obousměrném spojovém seznamu

1.



Provedeme následující atomické operace:

- Označíme ukazatele mazaného uzlu jako „ke smazání“

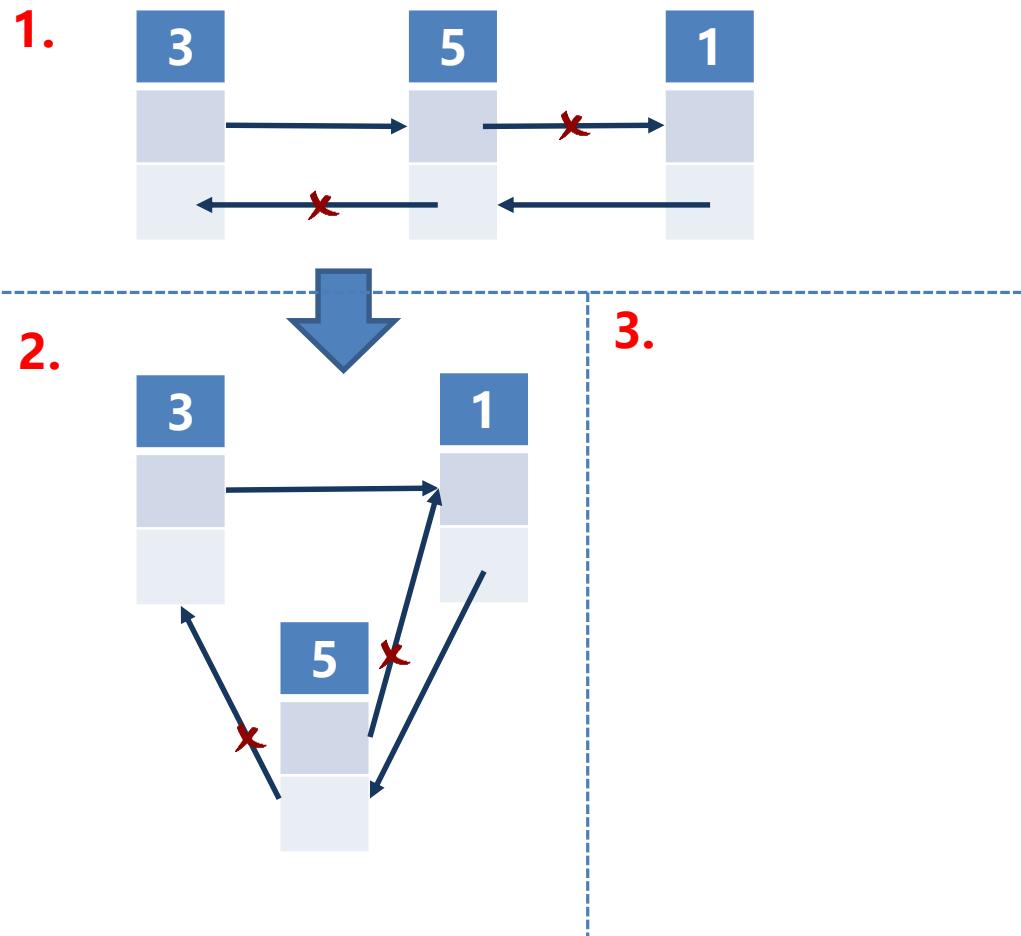
2.

3.

# Konkurenční datové struktury

## Příklad 3

- Mazání v obousměrném spojovém seznamu



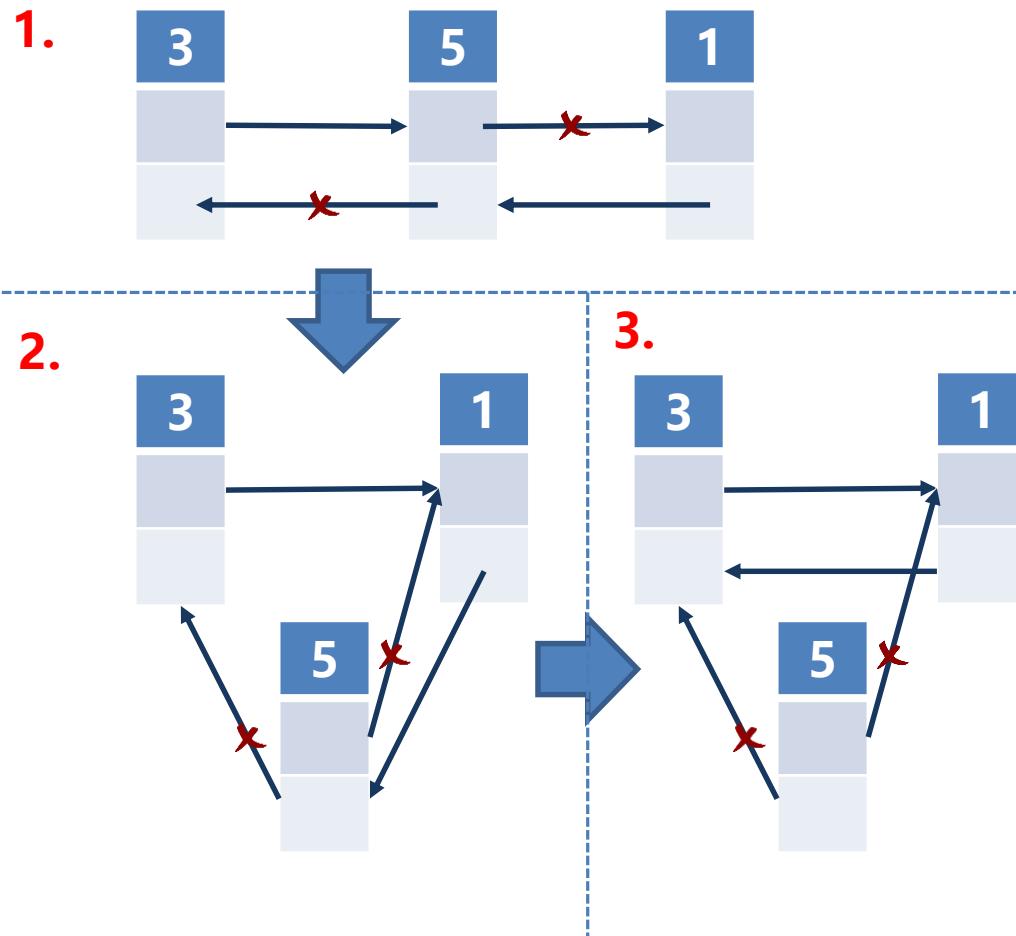
Provedeme následující atomické operace:

- Označíme ukazatele mazaného uzlu jako „ke smazání“
- Převedeme ukazatel předchůdce

# Konkurenční datové struktury

## Příklad 3

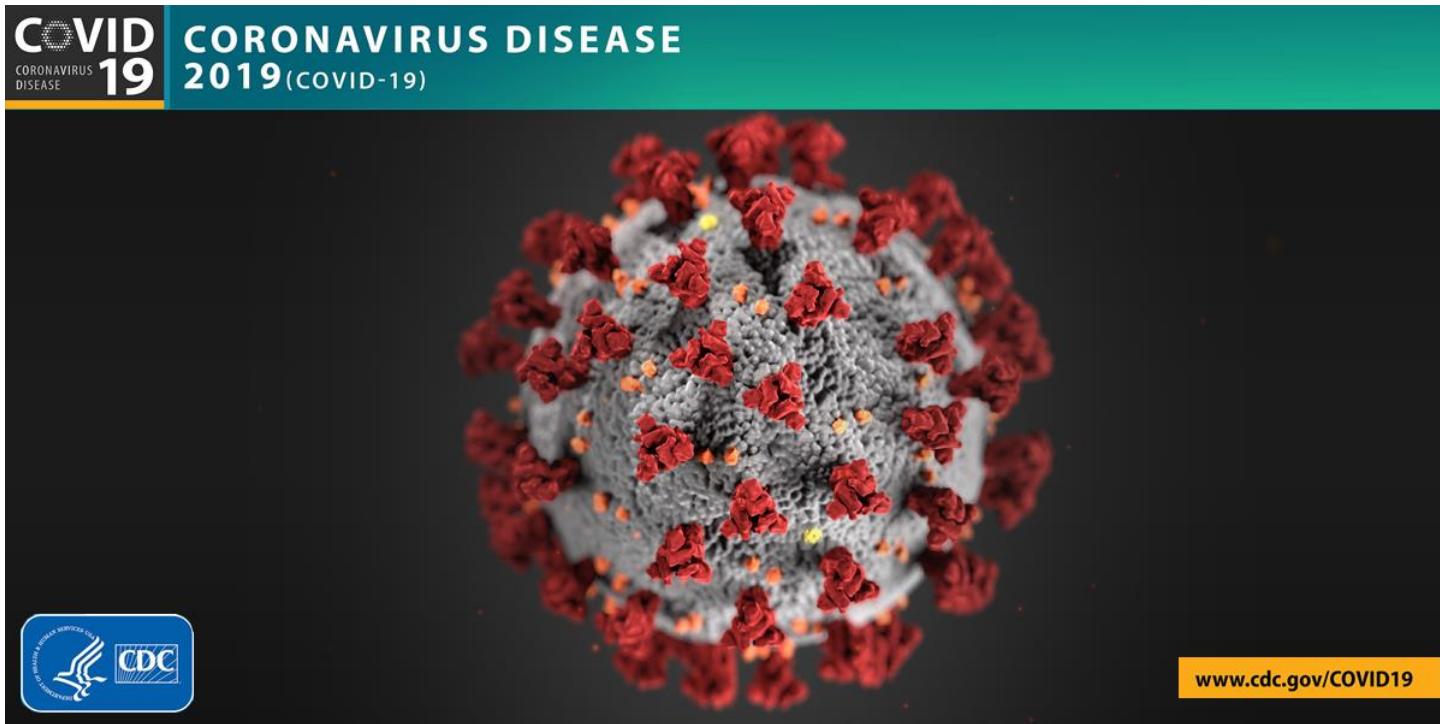
- Mazání v obousměrném spojovém seznamu



Provedeme následující atomické operace:

- Označíme ukazatele mazaného uzlu jako „ke smazání“
- Převedeme ukazatel předchůdce
- Převedeme ukazatel následovníka

# Paralelní a distribuované výpočty (B4B36PDV)



- Přihlašujte se pod Google účtem
- Pokud je to možné, používejte sluchátka
- Pokud nemluvíte, vypněte si mikrofón



# Paralelní a distribuované výpočty (B4B36PDV)

**Branislav Bošanský, Michal Jakob**

[bosansky@fel.cvut.cz](mailto:bosansky@fel.cvut.cz)

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

# Dnešní přednáška

## Motivace



# Dnešní přednáška

## Techniky paralelizace

Chci paralelizovat algoritmus XY

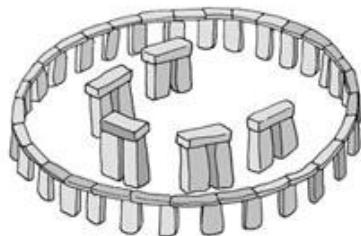


Jak na to?

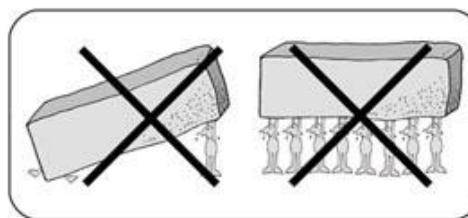
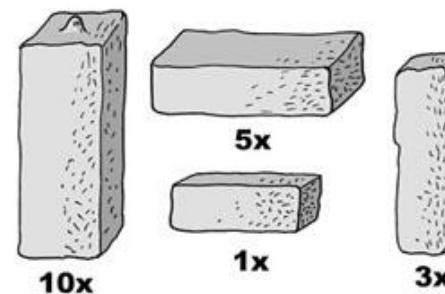
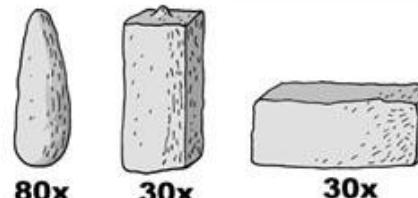
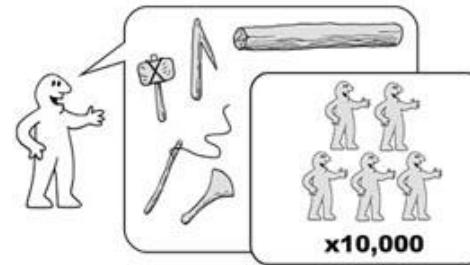
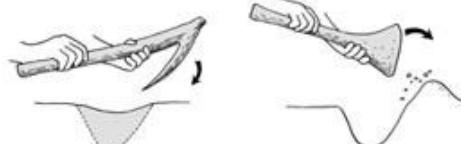
# Dnešní přednáška

Postup – Jak na to?

# HĚNJ



1



# Paralelní programování

Co chceme dosáhnout

- Potřebujeme se rozhodnout jak budeme úlohu **dekomponovat**, jak budeme **úkoly rozdělovat** a jakým způsobem zabezpečit celkovou orchestraci
- Klíčové cíle
  - **Vybalancování** – aby každé vlákno vykonávalo (přibližně) stejnou práci
  - **Minimalizace komunikace** – aby vlákna na sebe nemusely čekat
  - **Minimalizace duplicitní/zbytečné práce** – aby vlákna nepočítali něco, co by se nepočítalo bez paralelizace
- Neexistuje univerzální návod, musíte vždy přemýšlet jak dané cíle naplnit pro konkrétní úlohu

# Paralelní programování

Náhled

Problém

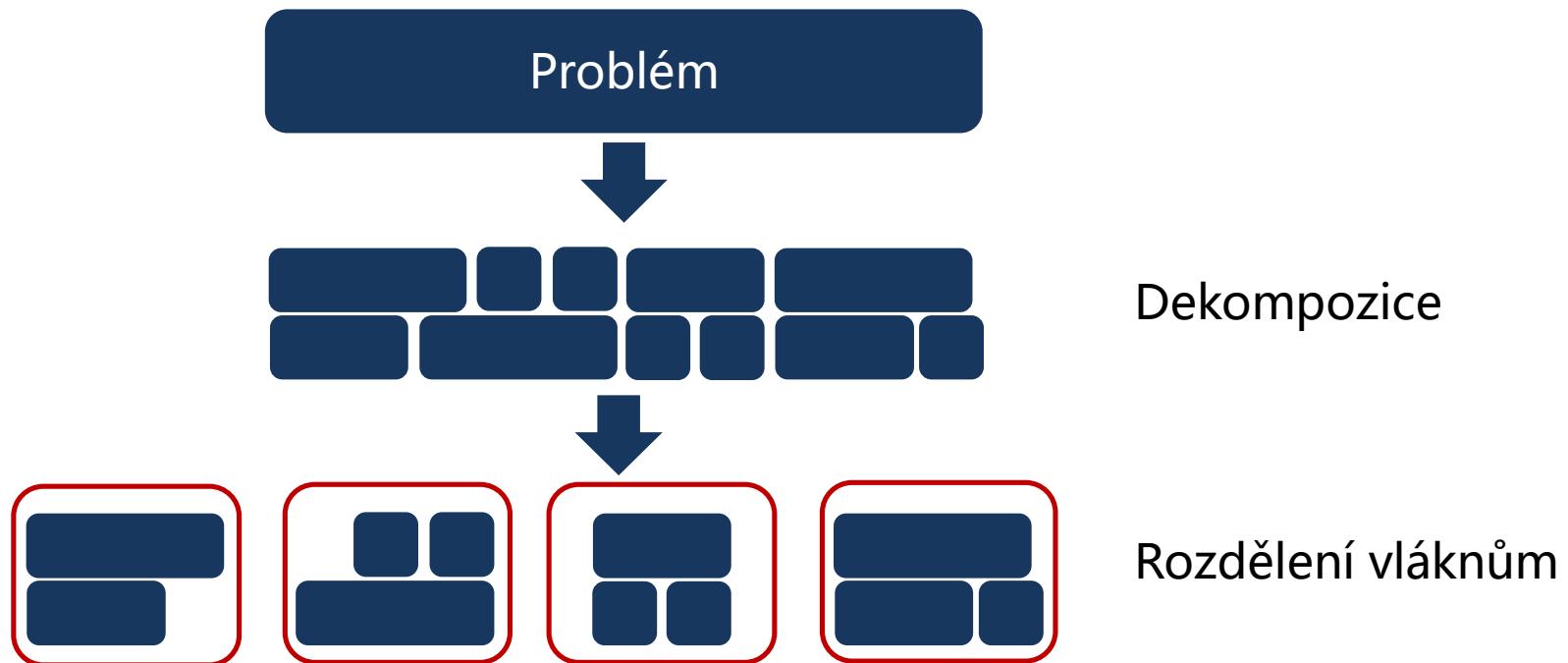
# Paralelní programování

## Náhled



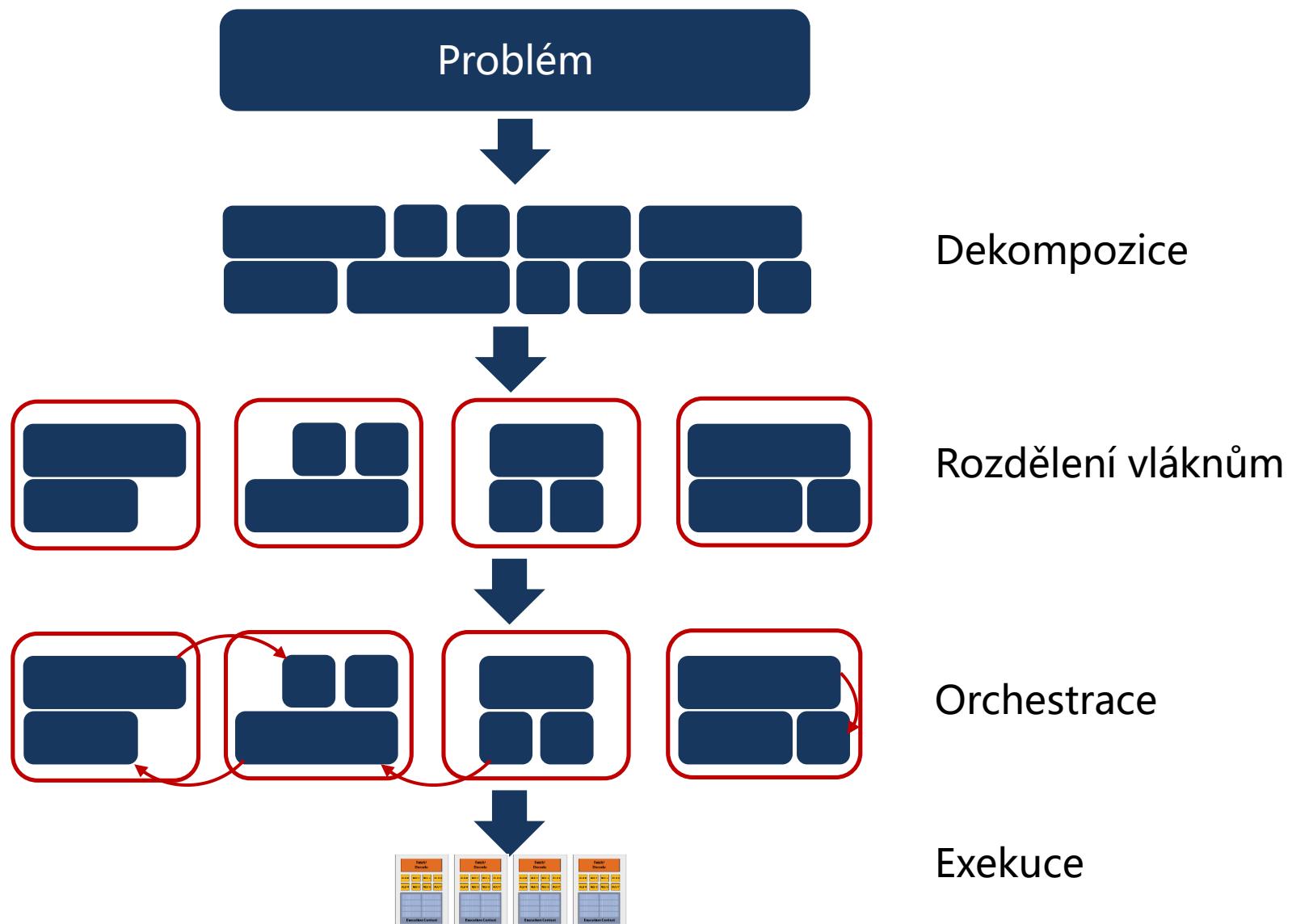
# Paralelní programování

## Náhled



# Paralelní programování

## Náhled

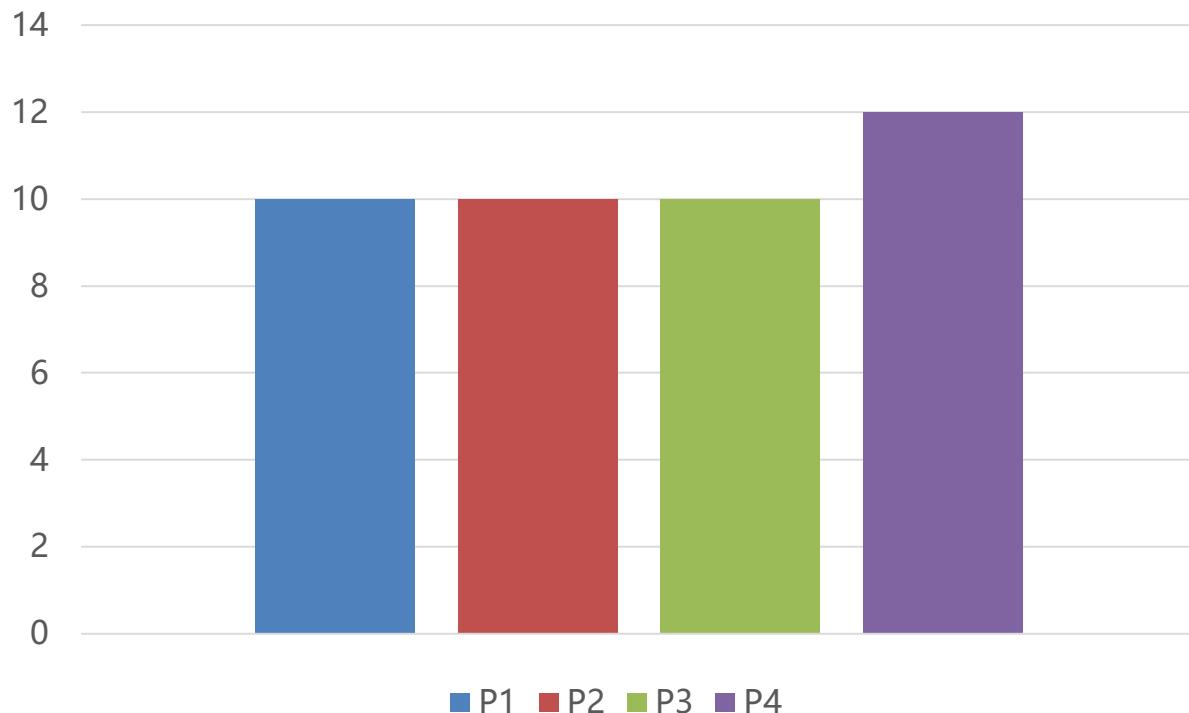


# Paralelní programování

## Balancování

- Ideálně chceme, aby všechna vlákna/jádra pracovaly a skončily současně

Čas výpočtu (s) pro jednotlivé vlákna/procesory

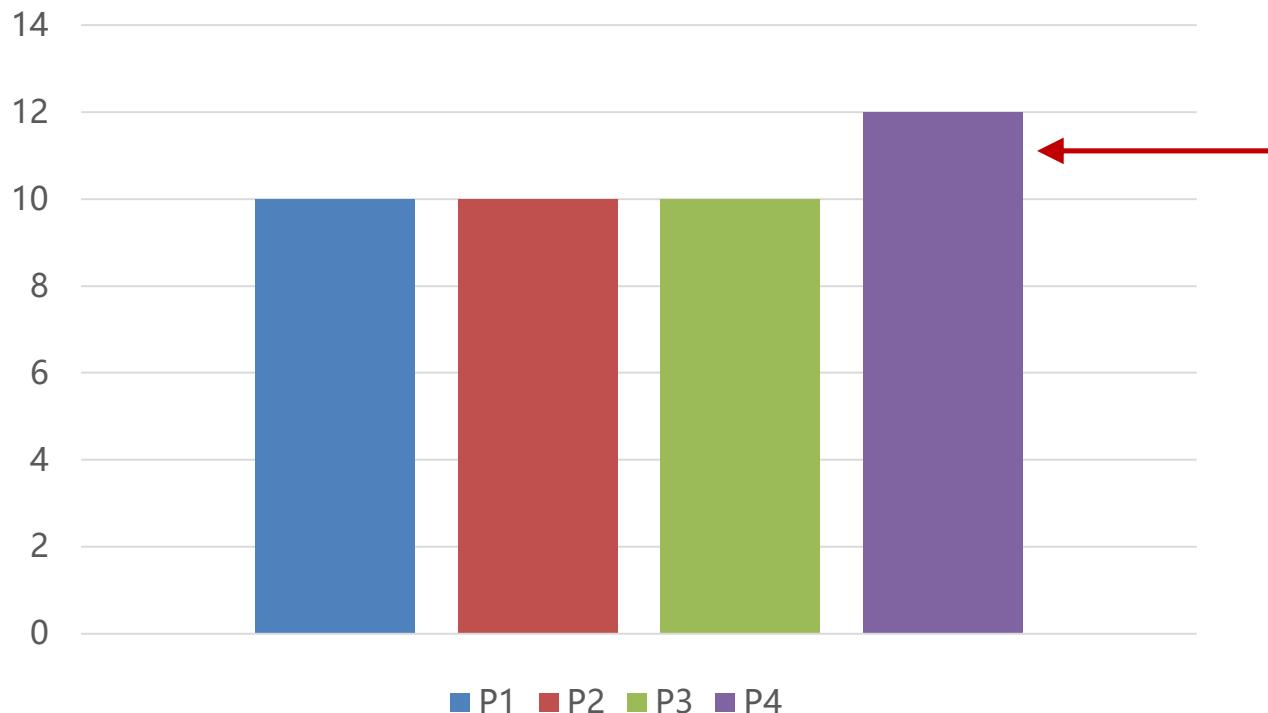


# Paralelní programování

## Balancování

- Ideálně chceme, aby všechna vlákna/jádra pracovaly a skončily současně

Čas výpočtu (s) pro jednotlivé vlákna/procesory



- Pokud 1 procesor pracuje o 20% déle, celý program pracuje o 20% déle
- Vzpomeňte si na Amdahlův zákon

# Rozdělení práce

## Statické rozdělení

- Fixní a statické rozdělení úkolů pro jednotlivá vlákna
  - Ne nutně v době komplikace
  - Jednou přidělíme vláknům úkoly a toto přidělení je neměnné
- Kdy nám statické rozdělení pomůže?
  - Všechny úkoly trvají (přibližně) stejně dlouho
  - Každý úkol může trvat různě dlouho, ale víme předem očekávanou dobu trvání
    - Můžeme vyřešit optimálně pomocí rozvrhování (Constraint Satisfaction Programming)

Čas výpočtu (s) pro jednotlivé  
vlákna/procesory



# Rozdělení práce

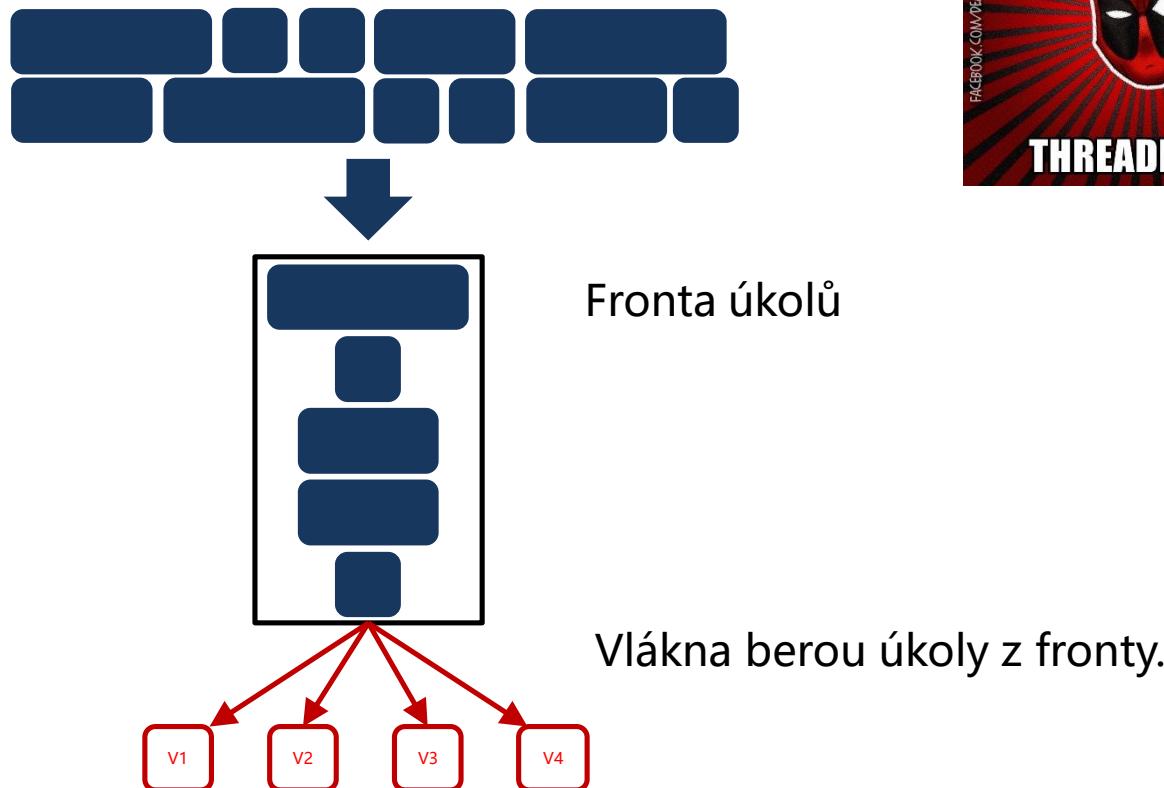
## Dynamické rozdělení

- Program přiděluje úkoly dynamicky na základě aktuálního vytížení jednotlivých vláken

# Rozdělení práce

## Dynamické rozdělení

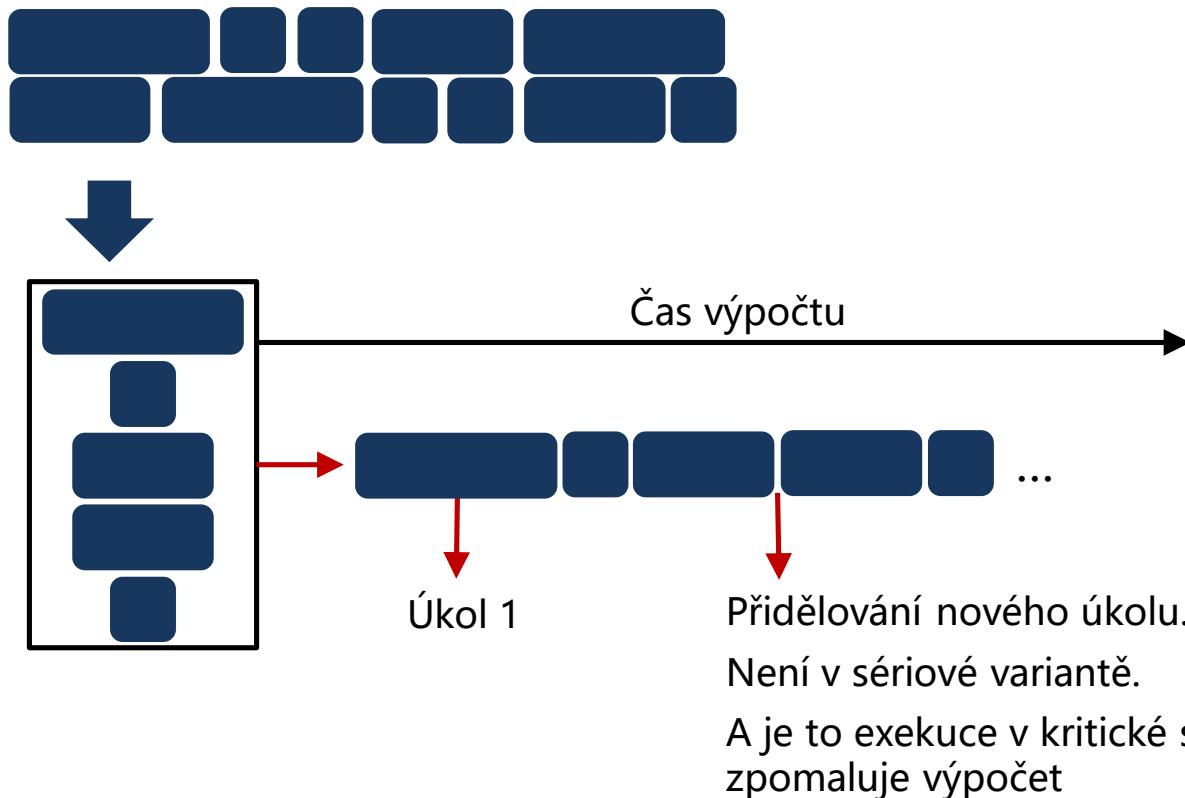
- Program přiděluje úkoly dynamicky na základě aktuálního vytížení jednotlivých vláken
  - Threadpool a fronta úkolů



# Rozdělení práce

## Dynamické rozdělení

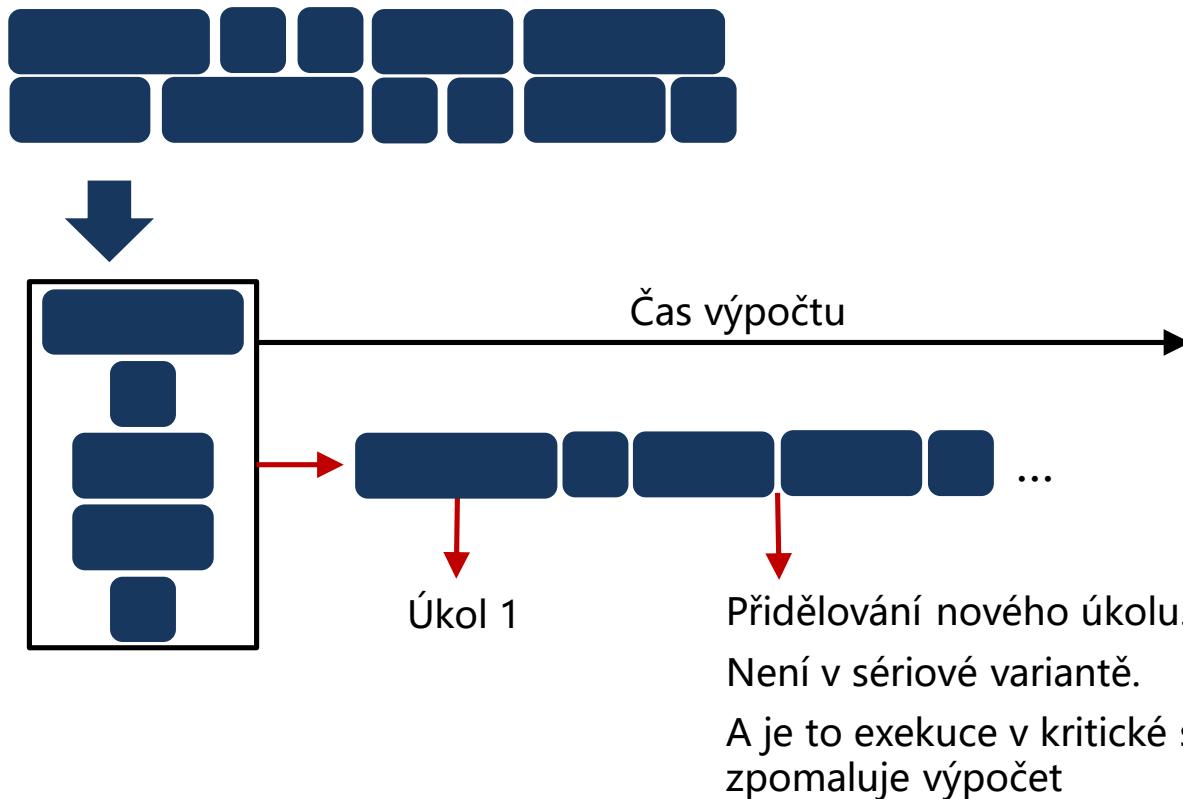
- Jak to bude vypadat z pohledu jednoho vlákna?
  - Threadpool a fronta úkolů



# Rozdělení práce

## Dynamické rozdělení

- Jak to bude vypadat z pohledu jednoho vlákna?
  - Threadpool a fronta úkolů



Více malých úkolů znamená dobré vybalancování mezi vlákna.

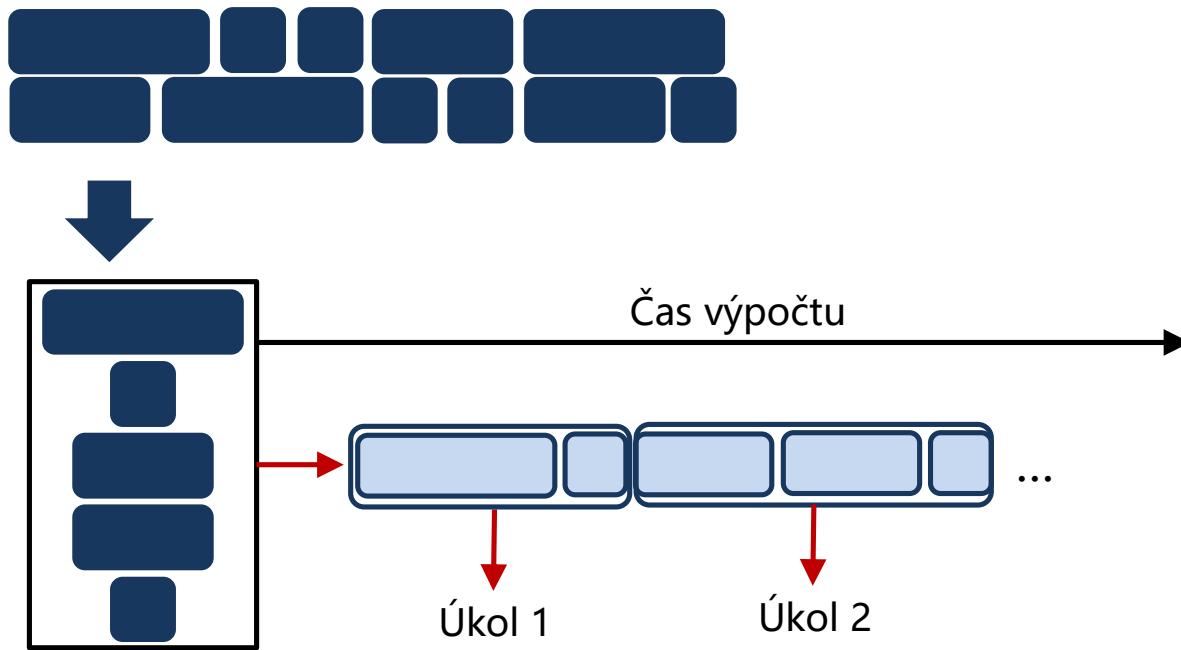


Více malých úkolů znamená více synchronizace a zpomalení.

# Rozdělení práce

## Dynamické rozdělení

- Můžeme měnit granularitu dekompozice



Zmenšení počtu úkolů sníží zpomalení kvůli synchronizaci



Ale můžeme mít problém s vybalancováním.

# Rozdělení práce

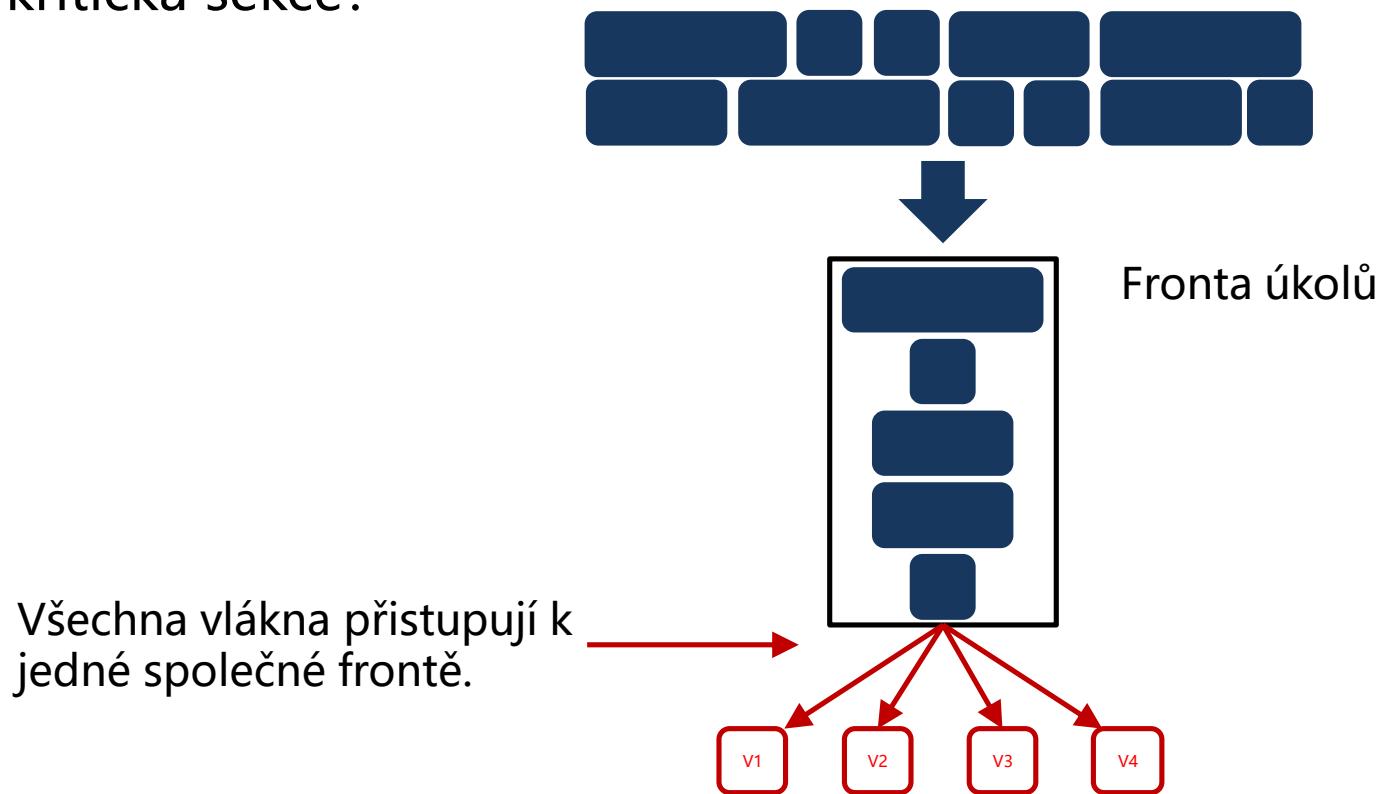
## Dynamické rozdělení

- Jak zvolit správnou velikost úkolu?
- Neexistuje univerzální odpověď – závisí na problému/HW (#CPU) atd.
- Pokud lze (máme odhad), můžeme přiřazovat dlouhé úkoly nejdřív a pak krátké úkoly

# Rozdělení práce

## Dynamické rozdělení – problémy

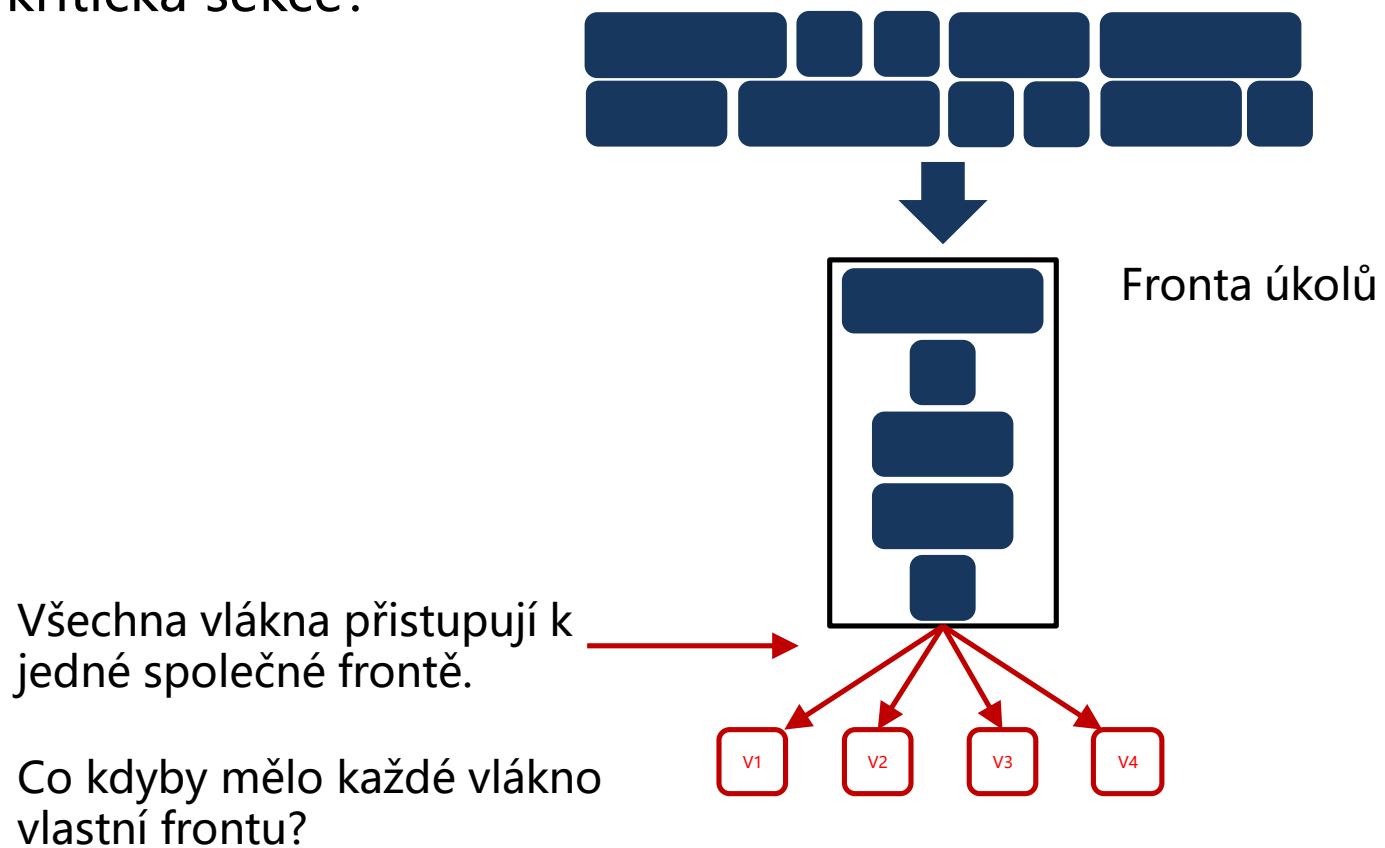
- Kde je kritická sekce?



# Rozdělení práce

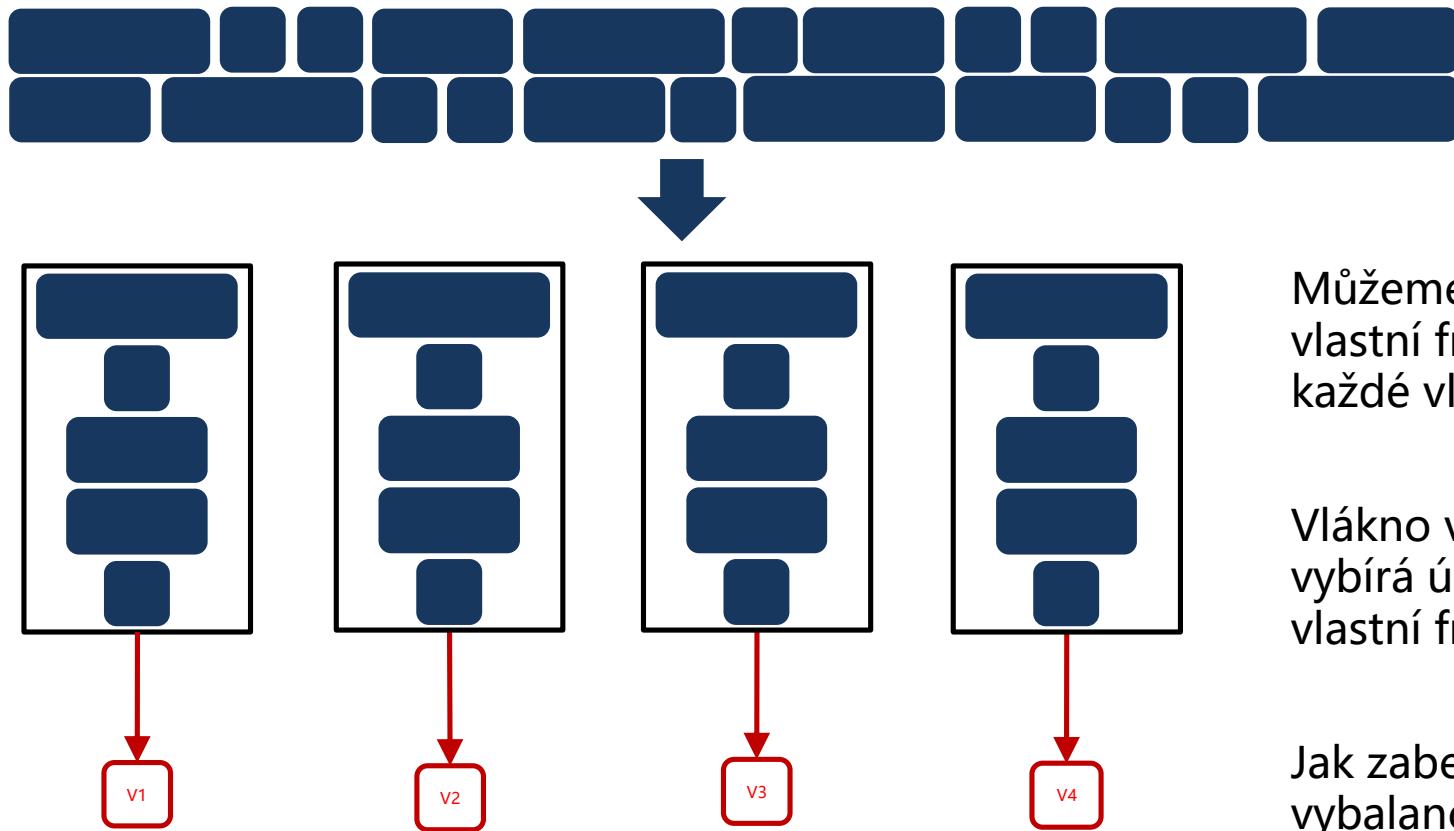
## Dynamické rozdělení – problémy

- Kde je kritická sekce?



# Rozdělení práce

Dynamické rozdělení – vlastní fronty



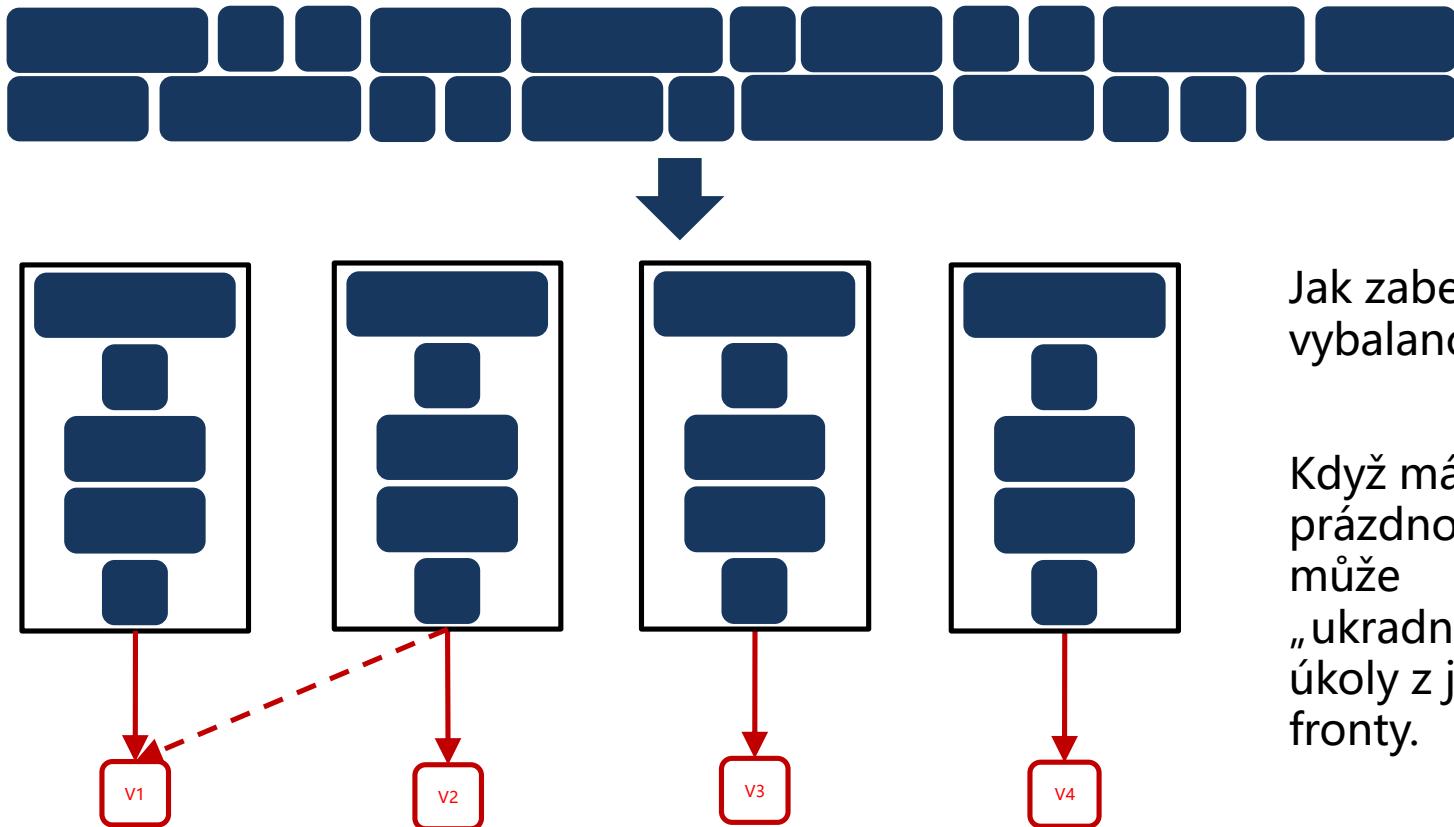
Můžeme vytvořit vlastní frontu pro každé vlákno

Vlákno vkládá a vybírá úkoly z vlastní fronty

Jak zabezpečíme vybalancování?

# Rozdělení práce

Dynamické rozdělení – vlastní fronty



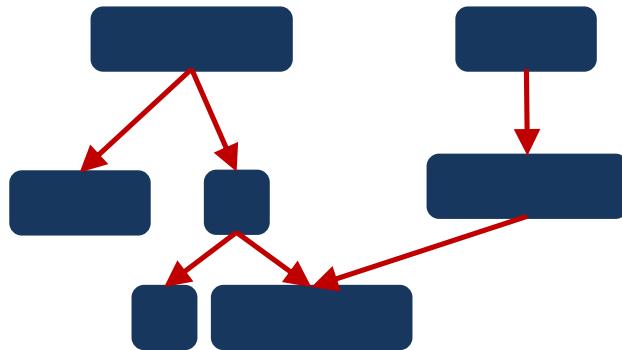
Jak zabezpečíme vybalancování?

Když má vlákno prázdnou frontu, může „ukradnout“ úkoly z jiné fronty.

# Rozdělení práce

## Dynamické rozdělení – závislosti

- Ne vždy je možné pustit libovolný úkol (např. pro spuštění úkolu X musíme znát aktuální hodnotu proměnné Y)
- Úkol bude zpracovaný vlákнем/procesorem pouze v případě, že všechny závislosti jsou splněny



- V OpenMP např. pomocí
  - `#pragma omp tasks depend([in/out/inout]:variables)`

# Rozdělení práce

## Dynamické rozdělení – závislosti v OpenMP

```
int main(int argc, char* argv[]) {

 int x = 0;

 #pragma omp parallel num_threads(thread_count) shared(x)
 {
 #pragma omp single
 {
 #pragma omp task depend(out:x)
 {
 std::this_thread::sleep_for(std::chrono::milliseconds(10));
 x++;
 std::cout << "1: x " << x << "\n";
 }
 #pragma omp task depend(in:x)
 {
 x *= 3;
 std::cout << "2: x " << x << "\n";
 }
 }
 std::cout << "final: x " << x << "\n";

 return 0;
 }
}
```

Když definujeme „in“ závislost, vytvoří se závislosti, vytvoří se závislost úkolu na již generovaných úkolech, které mají pro danou proměnnou nastavenou dependenci „out“ případně „inout“.

# Rozdělení práce

## Hybridní přístupy

- Rozdělení nemusí být pouze statické nebo dynamické
- V podstatě je možné zvolit libovolný mezistupeň mezi dvěma extrémy
  - Rozdělím úkoly
  - Sbírám statistiky o délce zpracování
  - Přerozdělím úkoly a opakuji

# Vzorce paralelizace

- Datový paralelismus
  - SIMD přístup
  - Rozdělím data a rovnou spustím zpracování pro jednotlivá vlákna
- Fork-join
  - Jedno vlákno zpracovává část úkolu
  - Identifikuje možné podúkoly a spustí nové vlákna/úkoly

# Rozděluj a panuj

## Quick Sort

- Základní třídící algoritmus
- Jak budeme paralelizovat?

```
void qs(std::vector<int>& vector_to_sort, int from, int to) {
 if (to - from <= base_size) {
 std::sort(vector_to_sort.begin() + from, vector_to_sort.begin() + to);
 return;
 }

 // rozdelení dle pivota (vector_to_sort[from])
 int part2_start = partition(vector_to_sort, from, to, vector_to_sort[from]);

 if (part2_start - from > 1) {
 qs(vector_to_sort, from, part2_start);
 }
 if (to - part2_start > 1) {
 qs(vector_to_sort, part2_start, to);
 }
}
```

# Rozděluj a panuj

## Quick Sort

- Můžeme asynchronně volat rekuzivní úkoly

```
void qs(std::vector<int>& vector_to_sort, int from, int to) {
 if (to - from <= base_size) {
 std::sort(vector_to_sort.begin() + from, vector_to_sort.begin() + to);
 return;
 }

 // rozdelení dle pivota (vector_to_sort[from])
 int part2_start = partition(vector_to_sort, from, to, vector_to_sort[from]);

 if (part2_start - from > 1) {
 #pragma omp task shared(vector_to_sort) firstprivate(from, part2_start)
 {
 qs(vector_to_sort, from, part2_start);
 }
 }
 if (to - part2_start > 1) {
 qs(vector_to_sort, part2_start, to);
 }
}
```

# Rozděluj a panuj

## Quick Sort

- Omezíme minimální velikost, aby nedocházelo k false-sharingu
- Můžeme asynchronně volat rekurzivní úkoly

```
void qs(std::vector<int>& vector_to_sort, int from, int to) {
 if (to - from <= base_size) {
 std::sort(vector_to_sort.begin() + from, vector_to_sort.begin() + to);
 return;
 }

 //rozdelení dle pivota (vector_to_sort[from])
 int part2_start = partition(vector_to_sort, from, to, vector_to_sort[from]);

 if (part2_start - from > 1) {
#pragma omp task shared(vector_to_sort) firstprivate(from, part2_start)
 {
 qs(vector_to_sort, from, part2_start);
 }
}
if (to - part2_start > 1) {
 qs(vector_to_sort, part2_start, to);
}
```

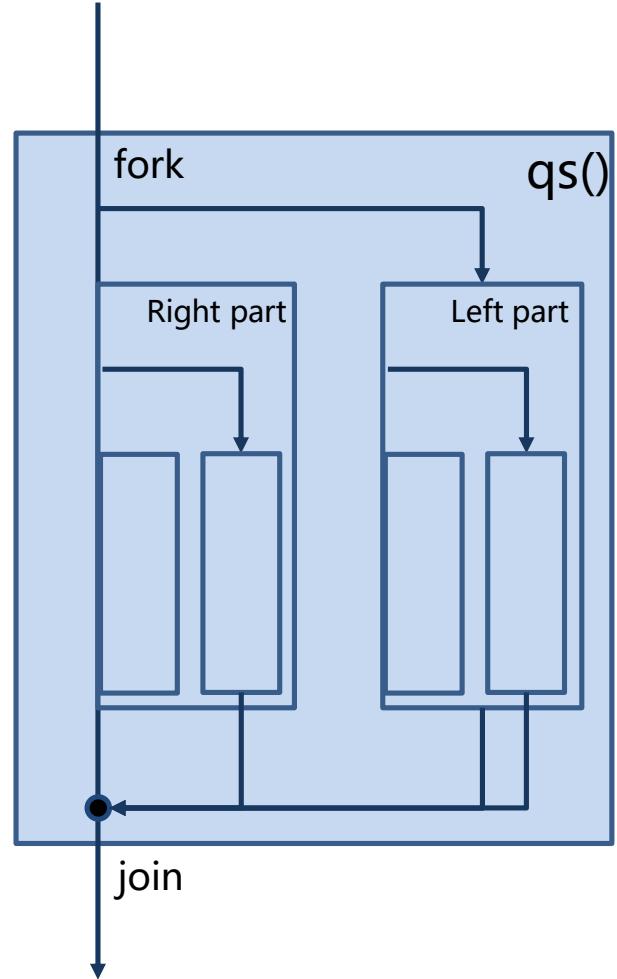
# Rozděluj a panuj

## Quick Sort

```
void qs(std::vector<int>& vector_to_sort, int from, int to) {
 if (to - from <= base_size) {
 std::sort(vector_to_sort.begin() + from, vector_to_sort.begin() + to);
 return;
 }

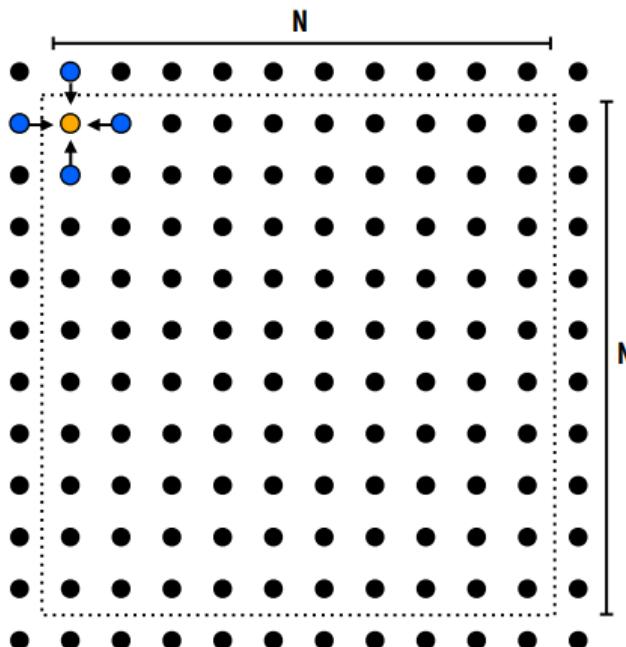
 //rozdelení dle pivota (vector_to_sort[from])
 int part2_start = partition(vector_to_sort,from,to,vector_to_sort[from]);

 if (part2_start - from > 1) {
#pragma omp task shared(vector_to_sort) firstprivate(from,part2_start)
 {
 qs(vector_to_sort, from, part2_start);
 }
 }
 if (to - part2_start > 1) {
 qs(vector_to_sort, part2_start, to);
 }
}
```



# Dekompozice se závislostmi

- paralelizace QuickSortu byla snadná vzhledem k žádné závislosti mezi úkoly
- Co když jsou úkoly závislé?

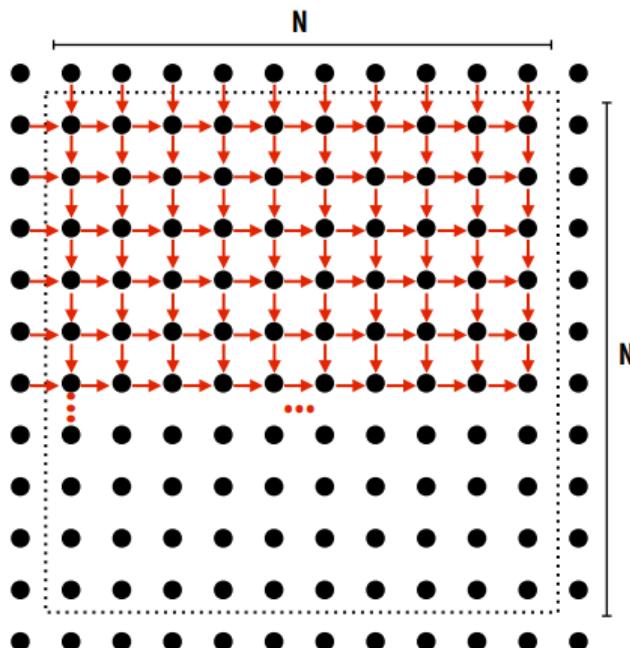


Problém:

- Chceme iterativně počítat průměr pro každé pole mřížky
  - $A[i, j] = 0.2 * (A[i - 1, j] + A[i, j - 1] + A[i, j] + A[i + 1, j] + A[i, j + 1])$
- Každou iteraci chceme projít celou matici z horního levého rohu
- Jaké jsou zde závislosti?

# Dekompozice se závislostmi

- paralelizace QuickSortu byla snadná vzhledem k žádné závislosti mezi úkoly
- Co když jsou úkoly závislé?

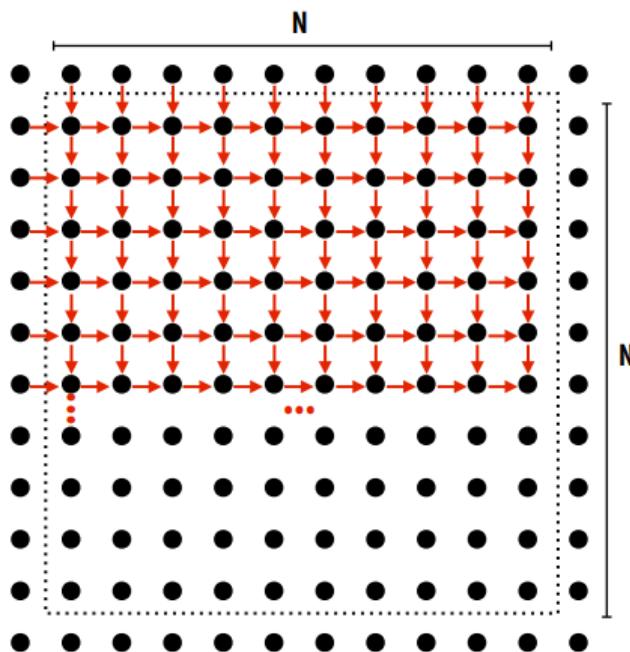


Problém:

- Chceme iterativně počítat průměr pro každé pole mřížky
  - $A[i, j] = 0.2 * (A[i - 1, j] + A[i, j - 1] + A[i, j] + A[i + 1, j] + A[i, j + 1])$
- Každou iteraci chceme projít celou matici z horního levého rohu
- Jaké jsou zde závislosti?

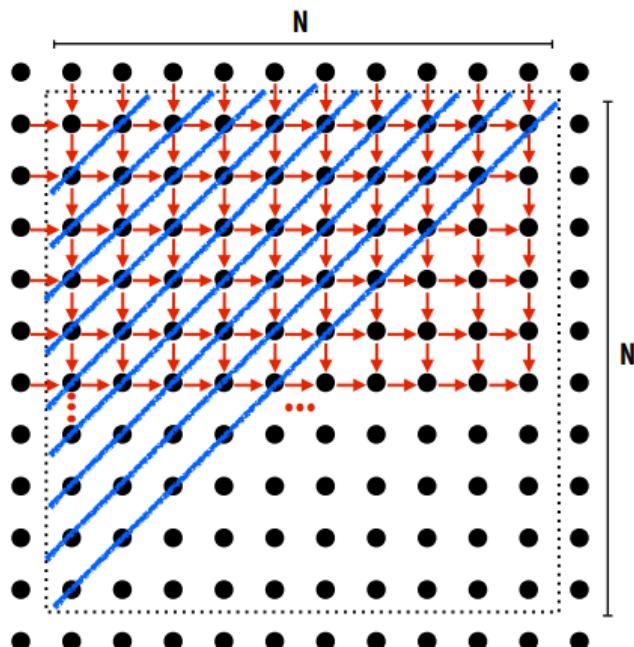
# Dekompozice se závislostmi

- Jakým způsobem můžeme tento problém paralelizovat?
- Zkusíme nalézt nezávislé úkoly
  - Které uzly lze aktualizovat paralelně?



# Dekompozice se závislostmi

- Jakým způsobem můžeme tento problém paralelizovat?
- Zkusíme nalézt nezávislé úkoly
  - Které uzly lze aktualizovat paralelně?



Uzly na diagonále jsou nezávislé (mohou přistupovat ke stejné proměnné, ale pouze pro čtení).

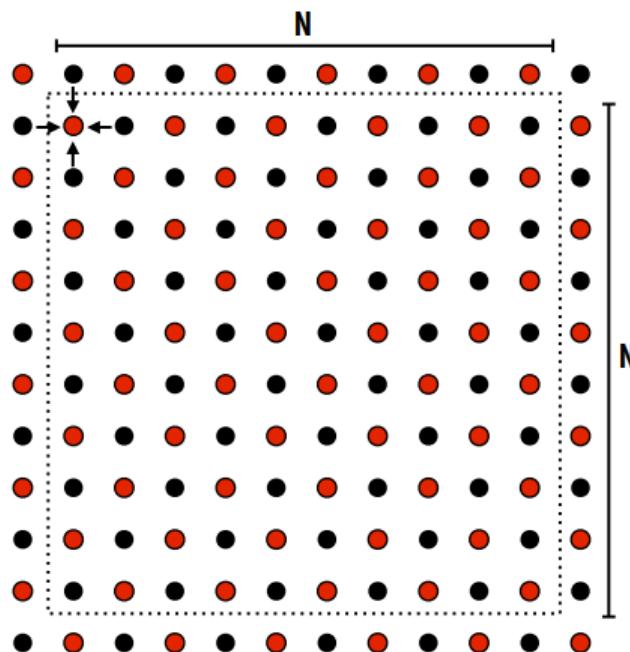


Problematické rozdělení na vlákna/procesory.

# Dekompozice se závislostmi

- Existuje lepší způsob?

 Pro konvergenci nemusíme nutně postupovat sekvenčně z jednoho rohu – uzly rozdělíme do dvou skupin a aktualizujeme nejdřív jednu, pak druhou



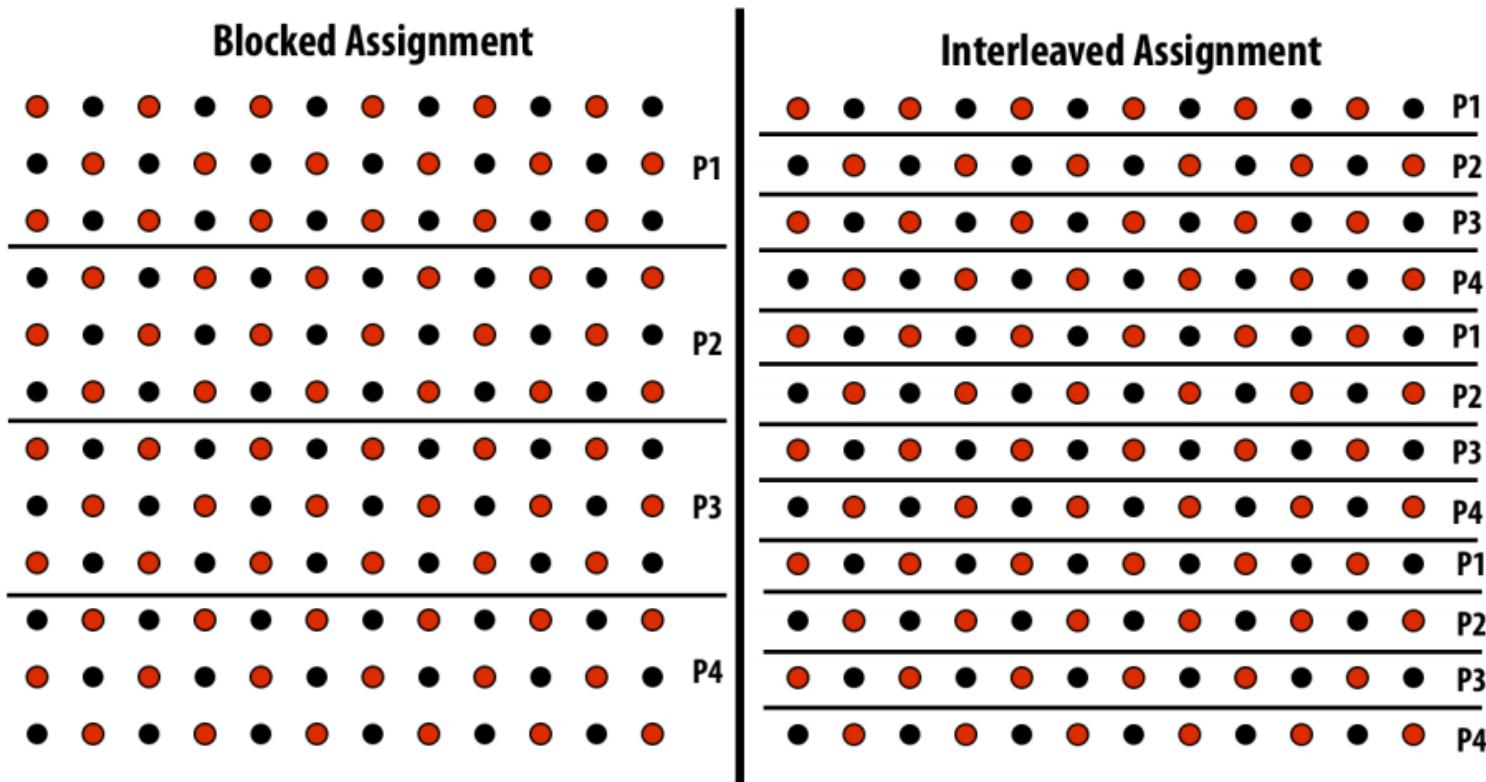
Jednoduchá paralelizace  
a rozdělení úkolu  
vláknům.



Musíme vědět, že si to  
můžeme dovolit (znalost  
problému/domény).

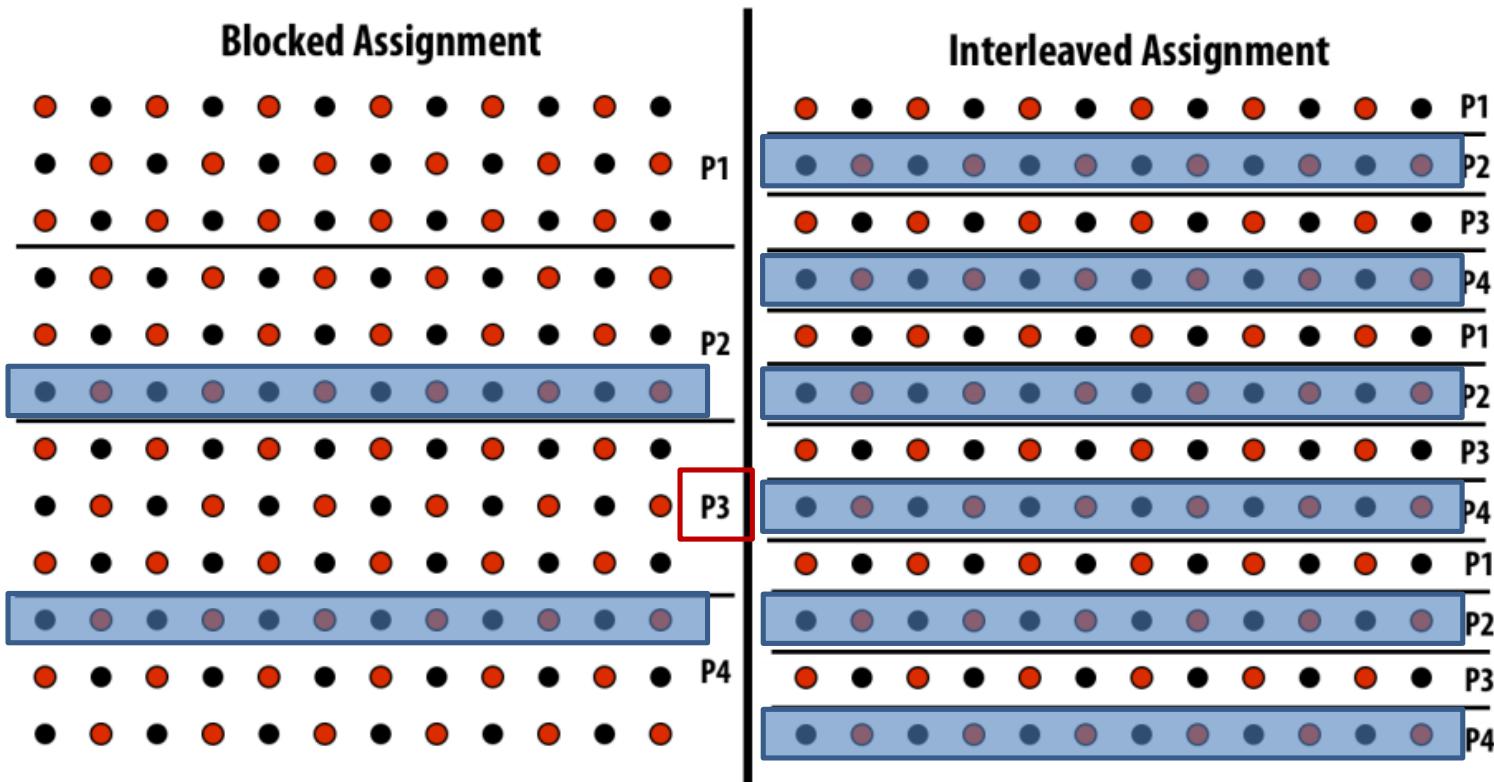
# Dekompozice se závislostmi

- Existuje lepší způsob?
- Jak můžeme rozdělit na úkoly pro vlákna/procesory?



# Dekompozice se závislostmi

- Který je lepší?
- Které části jsou privátní a které sdílené?



# Hledání prvočísel

## Eratostenovo síto

- Problém: Chceme zjistit počet prvočísel mezi 0 a X (např.  $10^9$ )
- Jaký je sériový algoritmus?

# Hledání prvočísel

## Eratostenovo síto

- Problém: Chceme zjistit počet prvočísel mezi 0 a X (např.  $10^9$ )
- Jaký je sériový algoritmus?

```
long result = 0;

for (int i = 2; i < MAXSQRRT; i++) {
 if (primes[i] == 1) {
 for (int j = i * i; j < MAXNUMBER; j += i) {
 primes[j] = 0;
 }
 }
}
for (int i = 0; i < MAXNUMBER; i++)
 result += primes[i];

return result;
```

Jak na to?

# Hledání prvočísel

## Eratostenovo síto

- Zkusíme paralelizovat hlavní for cyklus
- Můžeme paralelizovat druhý for cyklus pro součet

```
long result = 0;

#pragma omp parallel num_threads(thread_count)
{
#pragma omp for schedule(static)
 for (int i = 2; i < MAXSQRT; i++) {
 if (primes[i] == 1) {
 for (int j = i * i; j < MAXNUMBER; j += i) {
 primes[j] = 0;
 }
 }
 }
}

#pragma omp parallel for reduction(+:result)
for (int i = 0; i < MAXNUMBER; i++)
 result += primes[i];

return result;
```

Jak nám to bude fungovat?

# Hledání prvočísel

## Eratostenovo síto

```
long result = 0;

#pragma omp parallel num_threads(thread_count)
{
 #pragma omp for schedule(static)
 for (int i = 2; i < MAXSqrt; i++) {
 if (primes[i] == 1) {
 for (int j = i * i; j < MAXNUMBER; j += i) {
 primes[j] = 0;
 }
 }
 }

#pragma omp parallel for reduction(+:result)
 for (int i = 0; i < MAXNUMBER; i++)
 result += primes[i];

 return result;
}
```

Pro  $X=10^9$

| Sériová varianta | První paralelizace (4 vlákna) |
|------------------|-------------------------------|
| 11.7188 s        | 13.0681 s                     |

# Hledání prvočísel

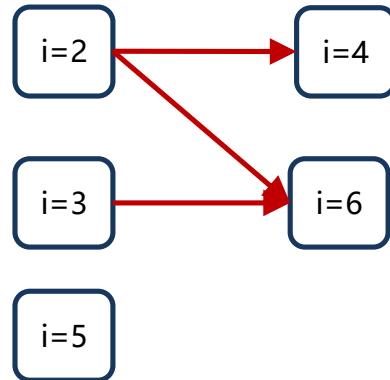
## Eratostenovo síto

- Co se stane když parallelizujeme hlavní cyklus?
  - Např. vlákno 0 bude zpracovávat iteraci  $i=2$ , vlákno 2 bude zpracovávat iteraci  $i=4$
  - Vlákno 2 dělá úplně zbytečnou práci – informace o tom, že číslo 4 není prvočíslo se k němu nemusí dostat včas

# Hledání prvočísel

## Eratostenovo síto

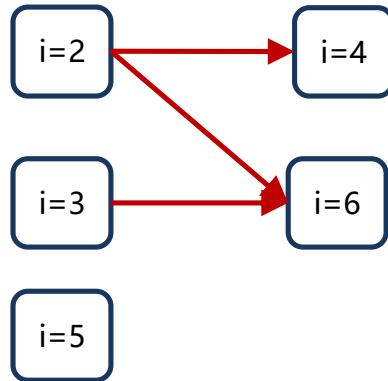
- Co se stane když parallelizujeme hlavní cyklus?
  - Např. vlákno 0 bude zpracovávat iteraci  $i=2$ , vlákno 2 bude zpracovávat iteraci  $i=4$
  - Vlákno 2 dělá úplně zbytečnou práci – informace o tom, že číslo 4 není prvočíslo se k němu nemusí dostat včas
- Jaká je závislost mezi úkoly?



# Hledání prvočísel

## Eratostenovo síto

- Co se stane když parallelizujeme hlavní cyklus?
  - Např. vlákno 0 bude zpracovávat iteraci  $i=2$ , vlákno 2 bude zpracovávat iteraci  $i=4$
  - Vlákno 2 dělá úplně zbytečnou práci – informace o tom, že číslo 4 není prvočíslo se k němu nemusí dostat včas
- Jaká je závislost mezi úkoly?

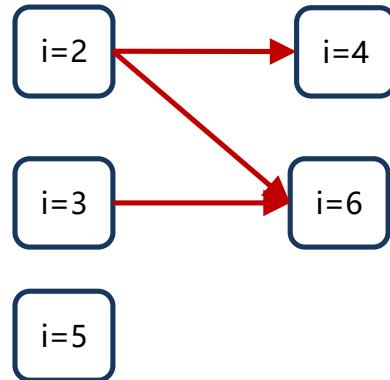


Na to abychom identifikovaly správné pořadí  
musíme vyřešit vlastní problém

# Hledání prvočísel

## Eratostenovo síto

- Co se stane když parallelizujeme hlavní cyklus?
  - Např. vlákno 0 bude zpracovávat iteraci  $i=2$ , vlákno 2 bude zpracovávat iteraci  $i=4$
  - Vlákno 2 dělá úplně zbytečnou práci – informace o tom, že číslo 4 není prvočíslo se k němu nemusí dostat včas
- Jaká je závislost mezi úkoly?



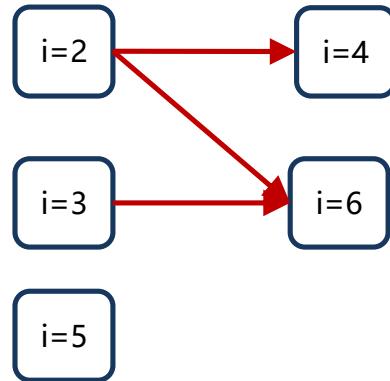
Na to abychom identifikovaly správné pořadí musíme vyřešit vlastní problém



# Hledání prvočísel

## Eratostenovo síto

- Co se stane když parallelizujeme hlavní cyklus?
  - Např. vlákno 0 bude zpracovávat iteraci  $i=2$ , vlákno 2 bude zpracovávat iteraci  $i=4$
  - Vlákno 2 dělá úplně zbytečnou práci – informace o tom, že číslo 4 není prvočíslo se k němu nemusí dostat včas
- Jaká je závislost mezi úkoly?



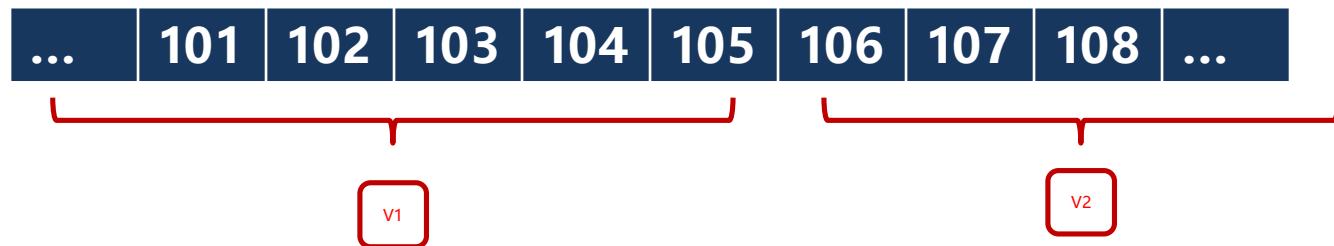
Na to abychom identifikovaly správné pořadí musíme vyřešit vlastní problém



# Hledání prvočísel

## Eratostenovo síto

- Jak můžeme snížit závislost?
  - Každé vlákno může kontrolovat pouze podinterval čísel

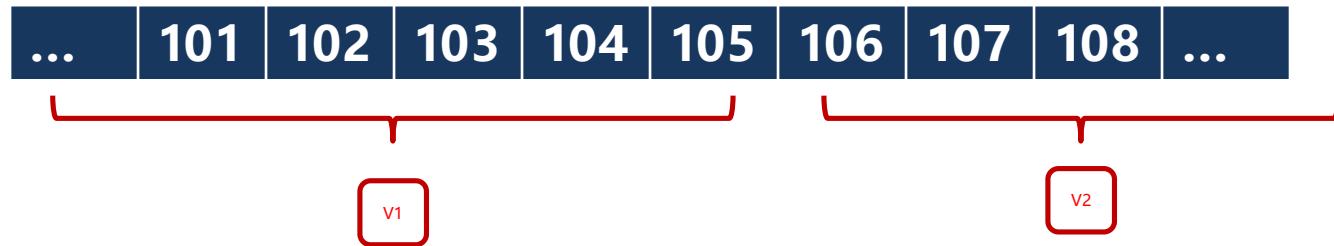


Úkol pro vlákno: označit čísla, které nejsou  
prvočísky v daném podintervalu

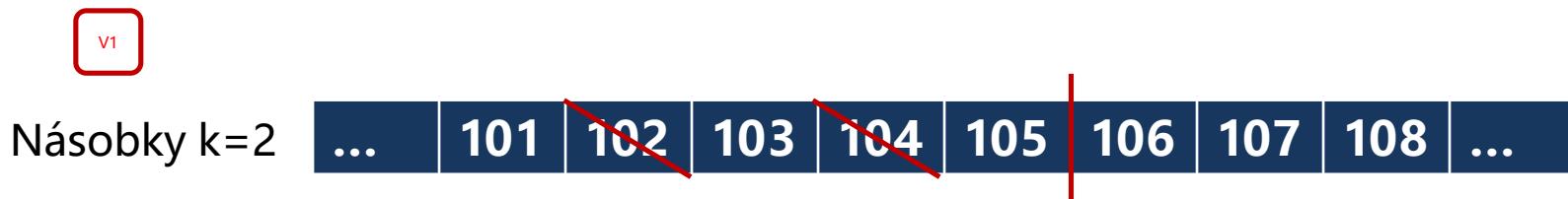
# Hledání prvočísel

## Eratostenovo síto

- Jak můžeme snížit závislost?
  - Každé vlákno může kontrolovat pouze podinterval čísel



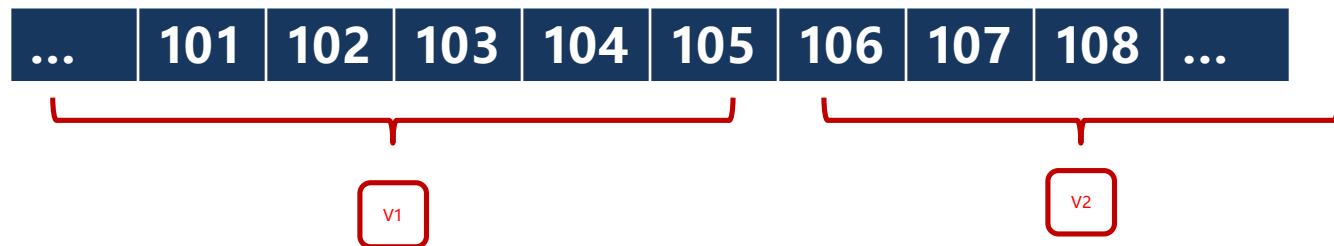
Úkol pro vlákno: označit čísla, které nejsou  
prvočísky v daném podintervalu



# Hledání prvočísel

## Eratostenovo síto

- Jak můžeme snížit závislost?
  - Každé vlákno může kontrolovat pouze podinterval čísel



Úkol pro vlákno: označit čísla, které nejsou  
prvočísky v daném podintervalu

$v_1$

Násobky  $k=2$



Násobky  $k=3$



...

# Hledání prvočísel

## Eratostenovo síto

```
long sieve() {
 int step = MAXNUMBER/thread_count/500;
 long result = 0;

#pragma omp parallel num_threads(thread_count) reduction(+:result)
{
#pragma omp for schedule(static)
 for (int i = 2; i < MAXNUMBER; i += step) {
 int from = i;
 int to = (i + step < MAXNUMBER) ? i + step : MAXNUMBER;

 for (int k = 2; k < MAXSQR; k++) {
 if (primes[k] == 1) {
 int start = std::max((from % k == 0) ? from : ((from/k)*k)+k, k*k);
 for (int j = start; j < to; j += k) {
 primes[j] = 0;
 }
 }
 for (int k = from; k < to; k++)
 result += primes[k];
 }
 }
 return result;
}
```

# Hledání prvočísel

## Eratostenovo síto

```
long sieve() {
 int step = MAXNUMBER/thread_count/500;
 long result = 0;

#pragma omp parallel num_threads(thread_count) reduction(+:result)
{
#pragma omp for schedule(static)
 for (int i = 2; i < MAXNUMBER; i += step) {
 int from = i;
 int to = (i + step < MAXNUMBER) ? i + step : MAXNUMBER;

 for (int k = 2; k < MAXSQR; k++) {
 if (primes[k] == 1) {
 int start = std::max((from % k == 0) ? from : ((from/k)*k)+k, k*k);
 for (int j = start; j < to; j += k) {
 primes[j] = 0;
 }
 }
 for (int k = from; k < to; k++)
 result += primes[k];
 }
 }
 return result;
}
```

první násobek **k** v intervalu [from,to]

# Hledání prvočísel

## Eratostenovo síto

```
long sieve() {
 int step = MAXNUMBER/thread_count/500;
 long result = 0;

#pragma omp parallel num_threads(thread_count) reduction(+:result)
{
#pragma omp for schedule(static)
 for (int i = 2; i < MAXNUMBER; i += step) {
 int from = i;
 int to = (i + step < MAXNUMBER) ? i + step : MAXNUMBER;

 for (int k = 2; k < MAXSQR; k++) {
 if (primes[k] == 1) {
 int start = std::max((from % k == 0) ? from : ((from/k)*k)+k, k*k);
 for (int j = start; j < to; j += k) {
 primes[j] = 0;
 }
 }
 for (int k = from; k < to; k++)
 result += primes[k];
 }
 }
 return result;
}
```

nulujeme od druhé mocniny  $k$

# Hledání prvočísel

## Eratostenovo síto

```
long sieve() {
 int step = MAXNUMBER/thread_count/500;
 long result = 0;

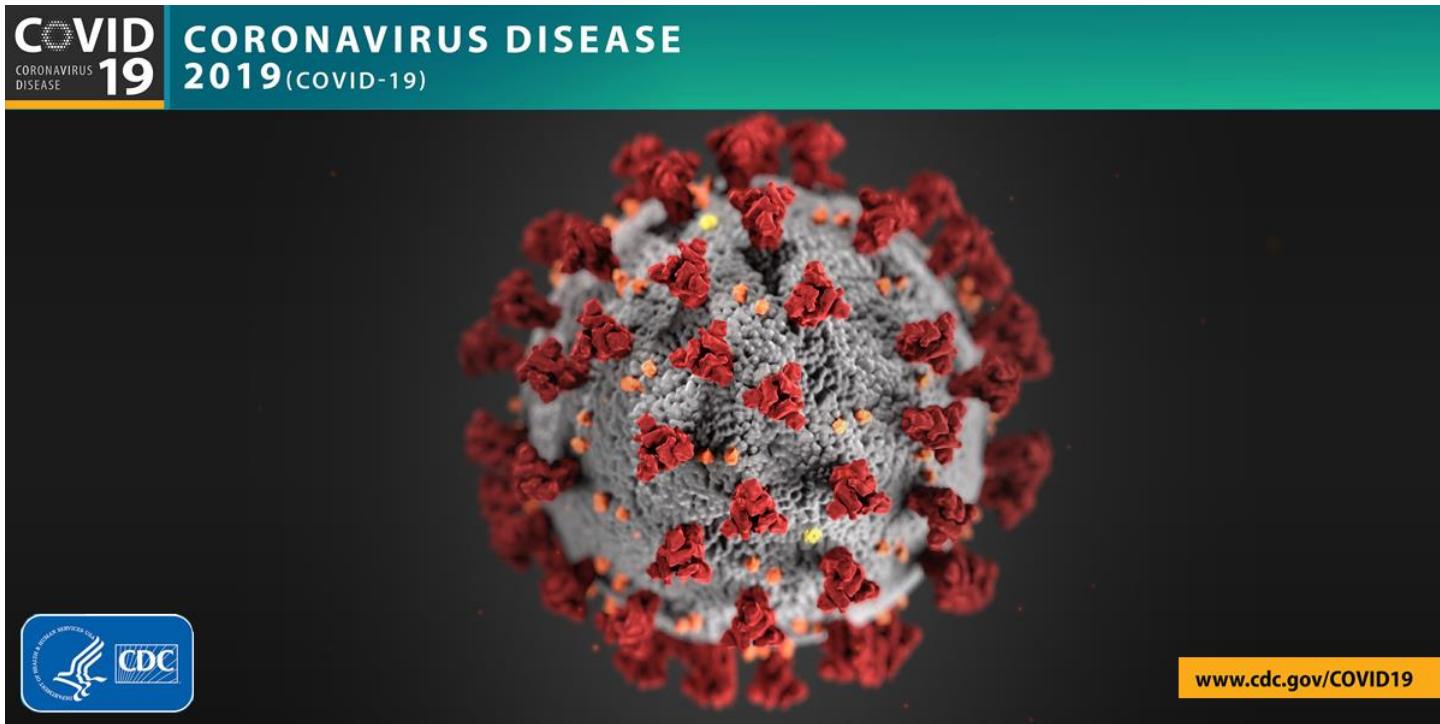
#pragma omp parallel num_threads(thread_count) reduction(+:result)
{
#pragma omp for schedule(static)
 for (int i = 2; i < MAXNUMBER; i += step) {
 int from = i;
 int to = (i + step < MAXNUMBER) ? i + step : MAXNUMBER;

 for (int k = 2; k < MAXSQR; k++) {
 if (primes[k] == 1) {
 int start = std::max((from % k == 0) ? from : ((from/k)*k)+k, k*k);
 for (int j = start; j < to; j += k) {
 primes[j] = 0;
 }
 }
 for (int k = from; k < to; k++)
 result += primes[k];
 }
 }
 return result;
}
```

Pro  $X=10^9$

| 1 větka | parallelizace (4 větky) |
|---------|-------------------------|
| 3.99 s  | 1.74 s                  |

# Paralelní a distribuované výpočty (B4B36PDV)



- Přihlašujte se pod Google účtem
- Pokud je to možné, používejte sluchátka
- Pokud nemluvíte, vypněte si mikrofón



# Paralelní a distribuované výpočty (B4B36PDV)

**Branislav Bošanský, Michal Jakob**

[bosansky@fel.cvut.cz](mailto:bosansky@fel.cvut.cz)

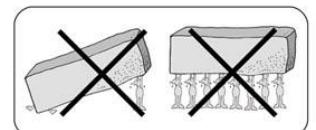
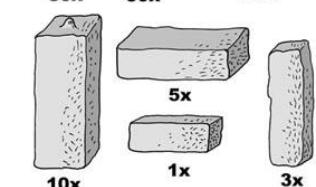
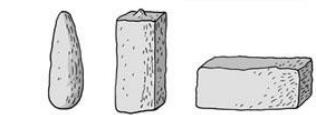
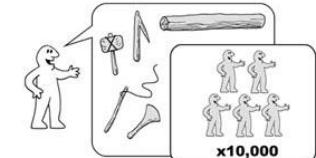
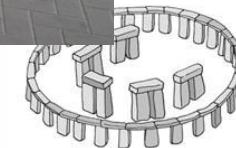
Artificial Intelligence Center  
Department of Computer Science  
Faculty of Electrical Engineering  
Czech Technical University in Prague

# Dnešní přednáška

## Motivace



HËNJ



# Dnešní přednáška

## Techniky paralelizace 2

Chci paralelizovat řadící algoritmus



Jak na to?

# Paralelní řazení

Techniky rozděluj a panuj

- Potřebujeme seřadit pole (čísel) dané velikosti a využít k tomu techniky paralelizace
- Podobně jako při standardních řadících algoritmech – pro ilustraci myšlenek paralelizace se věnujeme i těm méně efektivním
- Připomenutí
  - Quick Sort

```
void qs(std::vector<int>& vector_to_sort, int from, int to) {
 if (to - from <= base_size) {
 std::sort(vector_to_sort.begin() + from, vector_to_sort.begin() + to);
 return;
 }

 //rozdelení dle pivota (vector_to_sort[from])
 int part2_start = partition(vector_to_sort,from,to,vector_to_sort[from]);

 if (part2_start - from > 1) {
#pragma omp task shared(vector_to_sort) firstprivate(from,part2_start)
 {
 qs(vector_to_sort, from, part2_start);
 }
 }
 if (to - part2_start > 1) {
 qs(vector_to_sort, part2_start, to);
 }
}
```

# Paralelní řazení

- Který je nejjednodušší řadící algoritmus, který jste se naučili jako první?

# Paralelní řazení

- Který je nejjednodušší řadící algoritmus, který jste se naučili jako první?
- Bubble Sort
  - porovnává dva za sebou následující prvky
  - pokud jsou v nesprávném pořadí, vymění je

# Paralelní řazení

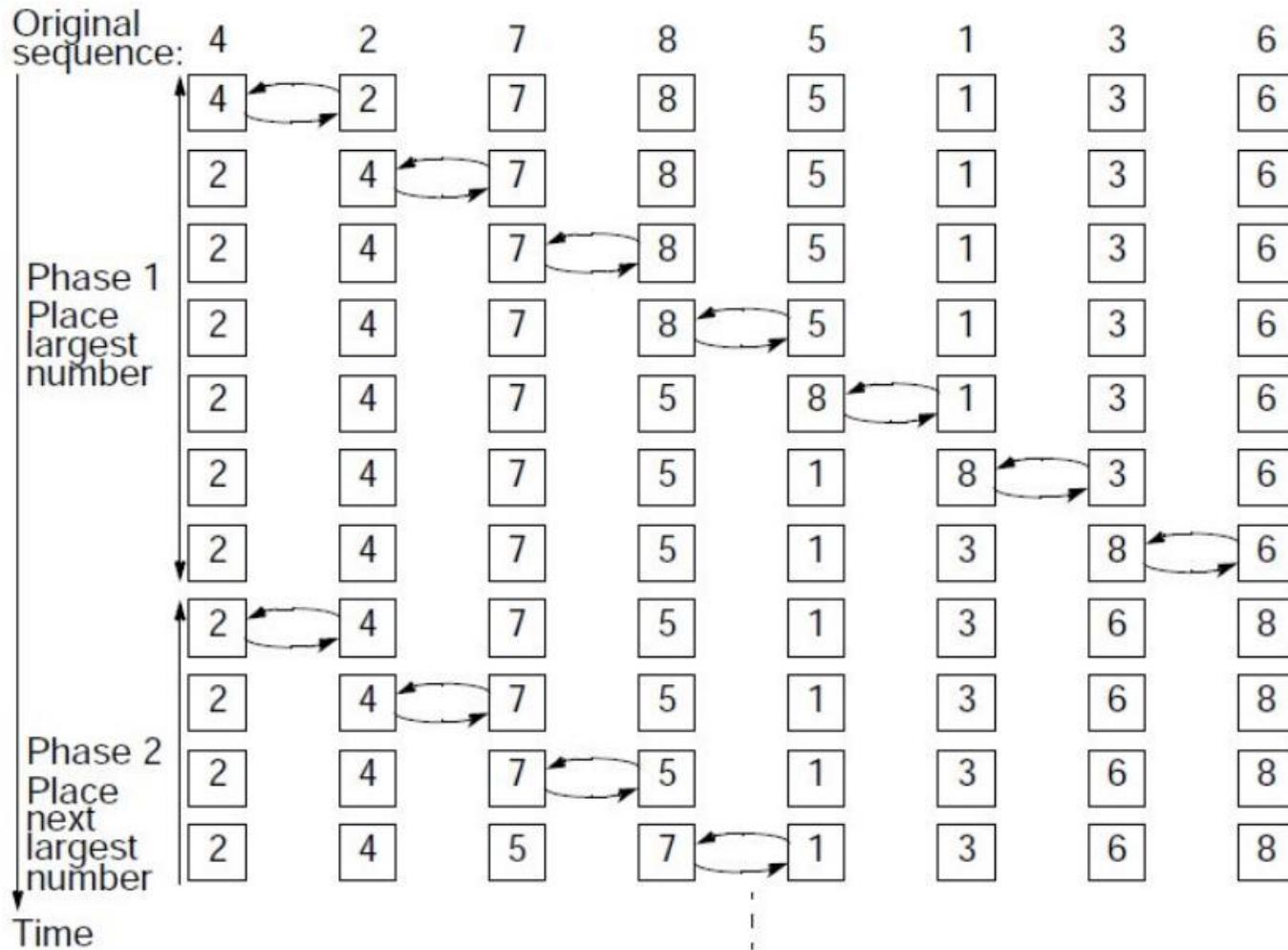
- Který je nejjednodušší řadící algoritmus, který jste se naučili jako první?
- Bubble Sort
  - porovnává dva za sebou následující prvky
  - pokud jsou v nesprávném pořadí, vymění je

```
bool compare_swap(std::vector<int>& vector_to_sort, const int& val1, const int& val2) {
 if (vector_to_sort[val1] > vector_to_sort[val2]) {
 std::iter_swap(vector_to_sort.begin() + val1,
 vector_to_sort.begin() + val2);
 return true;
 }
 return false;
}

void bubble(std::vector<int>& vector_to_sort, int from, int to) {
 bool change = true;
 while (change) {
 change = false;
 for (int i = from + 1; i < to; i++) {
 change |= compare_swap(vector_to_sort, i - 1, i);
 }
 }
}
```

# Paralelní řazení

## Bubble Sort



# Paralelní řazení

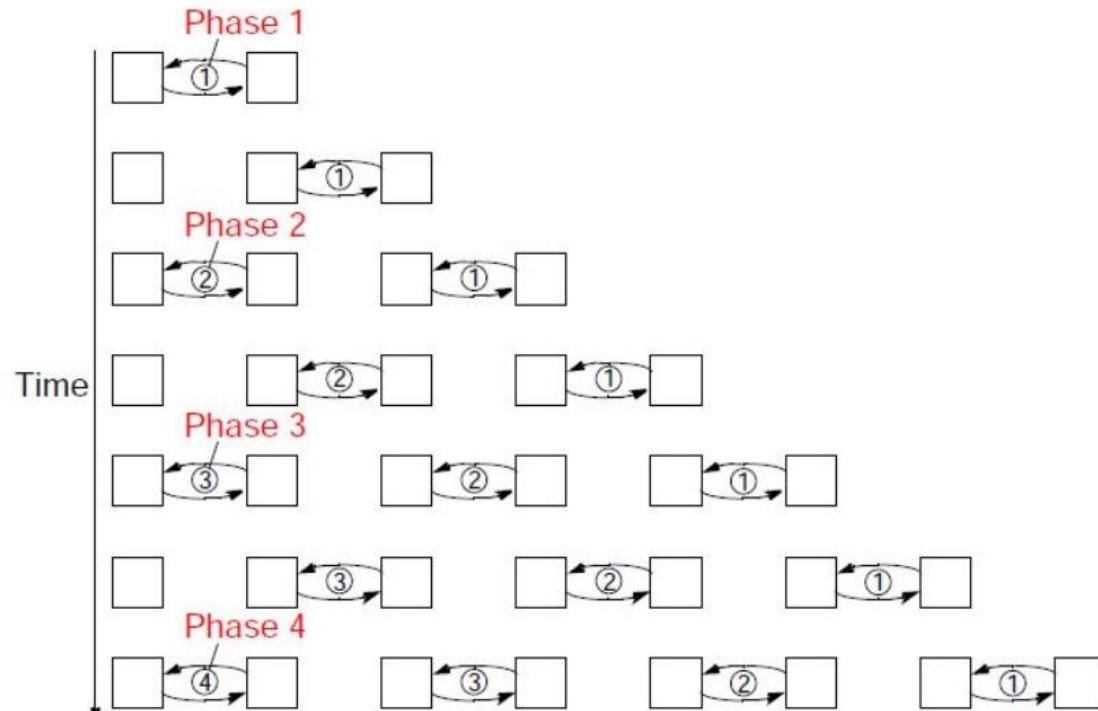
## Bubble Sort

- Jak lze bubble sort paralelizovat?

# Paralelní řazení

## Bubble Sort

- Jak lze bubble sort paralelizovat?
- Varianta 1
  - Vzpomeňte si na paralelizaci úloh na CPU – pipelineing



# Paralelní řazení

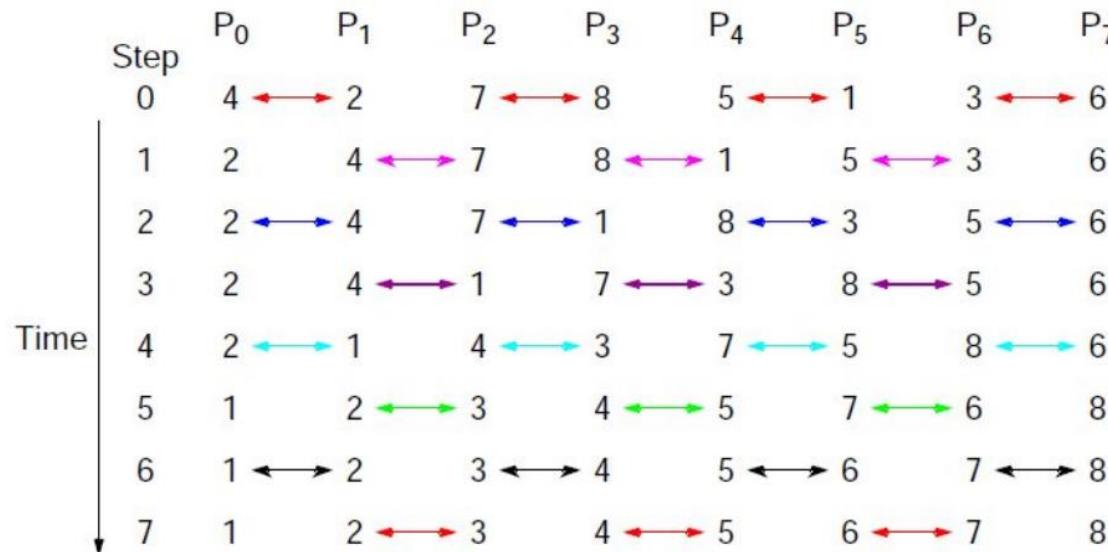
## Bubble Sort

- Lze bubble sort paralelizovat ještě jinak?
- Které porovnání lze dělat paralelně bez konfliktu?

# Paralelní řazení

## Bubble Sort

- Lze bubble sort paralelizovat ještě jinak?
- Které porovnání lze dělat paralelně bez konfliktu?
  - Vzpomeňte si na dekompozici při výpočtu průměrů v matici
  - Porovnání dle lichých/sudých čísel



# Paralelní řazení

## Bubble Sort

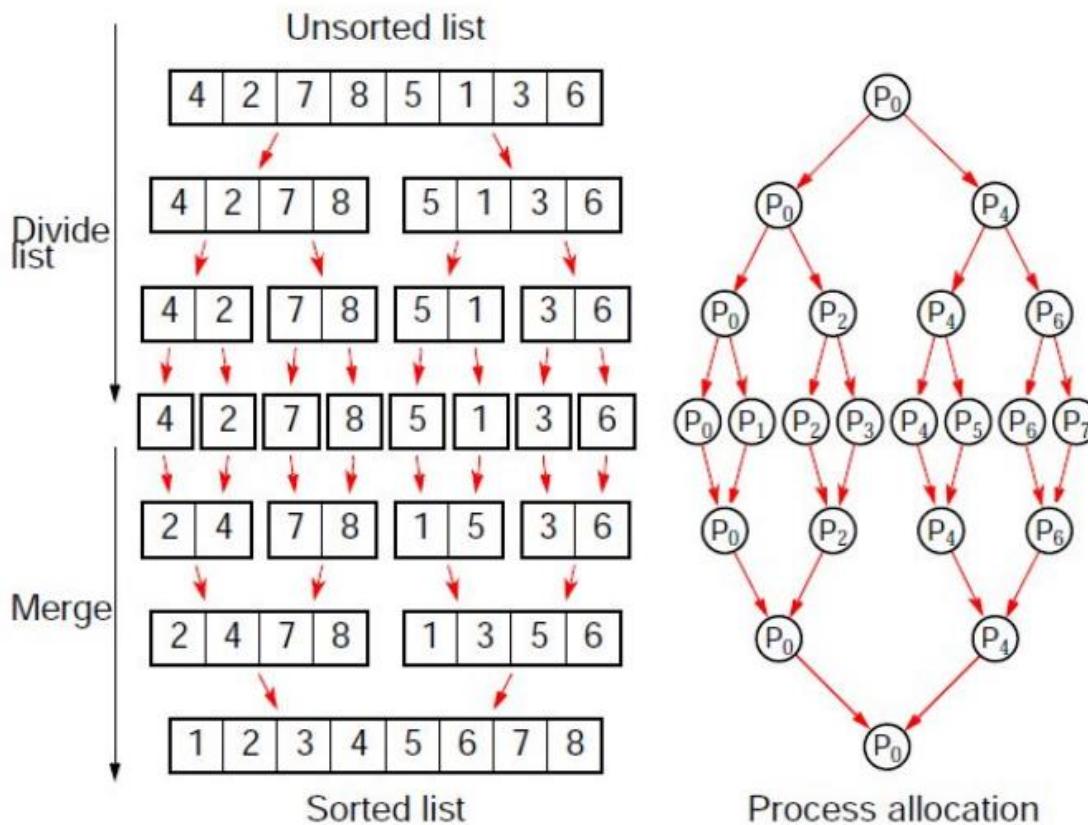
- Pro zvýšení paralelizace opět rozdělíme po blocích
- A můžeme paralelizovat

```
void parallel_bubble (std::vector<int>& vector_to_sort, unsigned int from,unsigned int to) {
 while (change) {
 change = false;
#pragma omp parallel for num_threads(thread_count) schedule(static) shared(vector_to_sort) reduction(|:change)
 for (int i = from + 1; i < to; i += 2) {
 change |= compare_swap(vector_to_sort, i - 1, i);
 }
#pragma omp parallel for num_threads(thread_count) schedule(static) shared(vector_to_sort) reduction(|:change)
 for (int i = from + 2; i < to; i += 2) {
 change |= compare_swap(vector_to_sort, i - 1, i);
 }
 }
}
```

# Paralelní řazení

## Merge Sort

- Jak paralelizujeme MergeSort?



# Paralelní řazení

## Merge Sort

- Jak paralelizujeme MergeSort?

```
void ms_serial(std::vector<int>& vector_to_sort, int from, int to) {
 if (to - from <= 1) {
 return;
 }
 int middle = (to - from)/2 + from;

 ms_serial(vector_to_sort, from, middle);
 ms_serial(vector_to_sort, middle, to);
 std::inplace_merge(vector_to_sort.begin() + from, vector_to_sort.begin() + middle, vector_to_sort.begin() + to);
}

void ms(std::vector<int>& vector_to_sort, int from, int to) {
 if (to - from <= base_size) {
 ms_serial(vector_to_sort, from, to);
 return;
 }
 int middle = (to - from)/2 + from;

 ms(vector_to_sort, from, middle);
 ms(vector_to_sort, middle, to);

 std::inplace_merge(vector_to_sort.begin() + from, vector_to_sort.begin() + middle, vector_to_sort.begin() + to);
}
```

# Paralelní řazení

## Merge Sort

- Jak paralelizujeme MergeSort?

```
void ms_serial(std::vector<int>& vector_to_sort, int from, int to) {
 if (to - from <= 1) {
 return;
 }
 int middle = (to - from)/2 + from;

 ms_serial(vector_to_sort, from, middle);
 ms_serial(vector_to_sort, middle, to);
 std::inplace_merge(vector_to_sort.begin() + from, vector_to_sort.begin() + middle, vector_to_sort.begin() + to);
}

void ms(std::vector<int>& vector_to_sort, int from, int to) {
 if (to - from <= base_size) {
 ms_serial(vector_to_sort, from, to);
 return;
 }
 int middle = (to - from)/2 + from;

 ms(vector_to_sort, from, middle);
 ms(vector_to_sort, middle, to);

 std::inplace_merge(vector_to_sort.begin() + from, vector_to_sort.begin() + middle, vector_to_sort.begin() + to);
}
```

# Paralelní řazení

## Merge Sort

- Která varianta je správná?

A

```
#pragma omp task shared(vector_to_sort) firstprivate(from,middle)
 ms(vector_to_sort, from, middle);
 ms(vector_to_sort, middle, to);

#pragma omp taskwait
 std::inplace_merge(vector_to_sort.begin() + from, vector_to_sort.begin() + middle, vector_to_sort.begin() + to);
```

B

```
ms(vector_to_sort, from, middle);
#pragma omp task shared(vector_to_sort) firstprivate(from,middle)
 ms(vector_to_sort, middle, to);

#pragma omp taskwait
 std::inplace_merge(vector_to_sort.begin() + from, vector_to_sort.begin() + middle, vector_to_sort.begin() + to);
```

C

```
#pragma omp task shared(vector_to_sort) firstprivate(from,middle)
 ms(vector_to_sort, from, middle);
 ms(vector_to_sort, middle, to);

 std::inplace_merge(vector_to_sort.begin() + from, vector_to_sort.begin() + middle, vector_to_sort.begin() + to);
```



# Paralelní řazení

## Merge Sort

- Jak paralelizujeme MergeSort?

```
void ms_serial(std::vector<int>& vector_to_sort, int from, int to) {
 if (to - from <= 1) {
 return;
 }
 int middle = (to - from)/2 + from;

 ms_serial(vector_to_sort, from, middle);
 ms_serial(vector_to_sort, middle, to);
 std::inplace_merge(vector_to_sort.begin() + from, vector_to_sort.begin() + middle, vector_to_sort.begin() + to);
}

void ms(std::vector<int>& vector_to_sort, int from, int to) {
 if (to - from <= base_size) {
 ms_serial(vector_to_sort, from, to);
 return;
 }
 int middle = (to - from)/2 + from;

#pragma omp task shared(vector_to_sort) firstprivate(from,middle)
 ms(vector_to_sort, from, middle);

 ms(vector_to_sort, middle, to);

#pragma omp taskwait
 std::inplace_merge(vector_to_sort.begin() + from, vector_to_sort.begin() + middle, vector_to_sort.begin() + to);
}
```

# Paralelní řazení

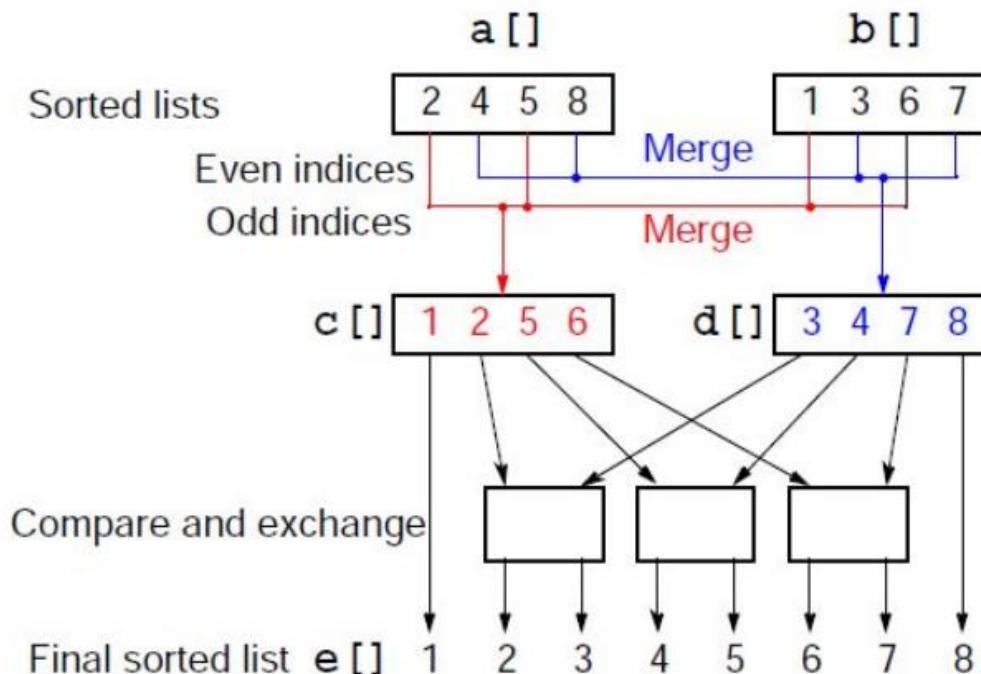
## Merge Sort

- Lze merge sort paralelizovat lépe?

# Paralelní řazení

## Merge Sort

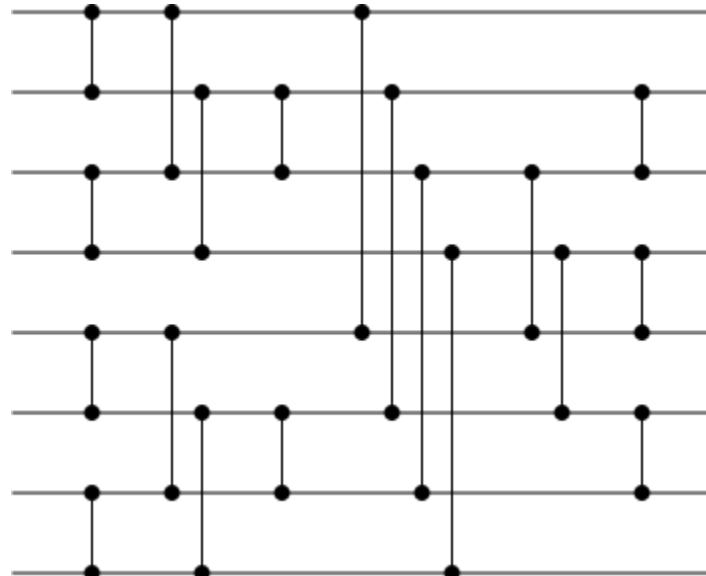
- Lze merge sort paralelizovat lépe?
- Také zde lze využít liché/sudé porovnání



# Paralelní řazení

## Odd-Even Merge Sort

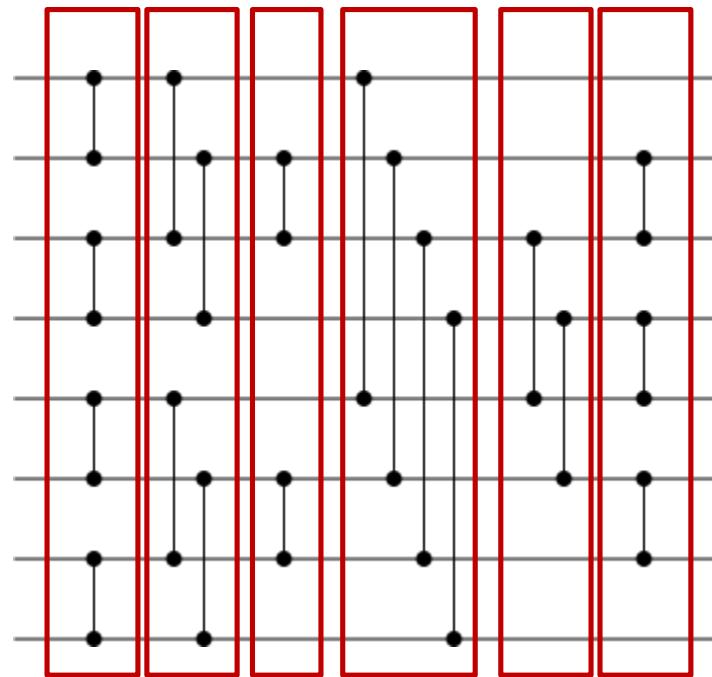
- Využíváme podobnou myšlenku jak v bubble sortu
  - Identifikujeme dvojice čísel, porovnání kterých lze dělat paralelně
- Jak to funguje?
  - Pro 8 prvků



# Paralelní řazení

## Odd-Even Merge Sort

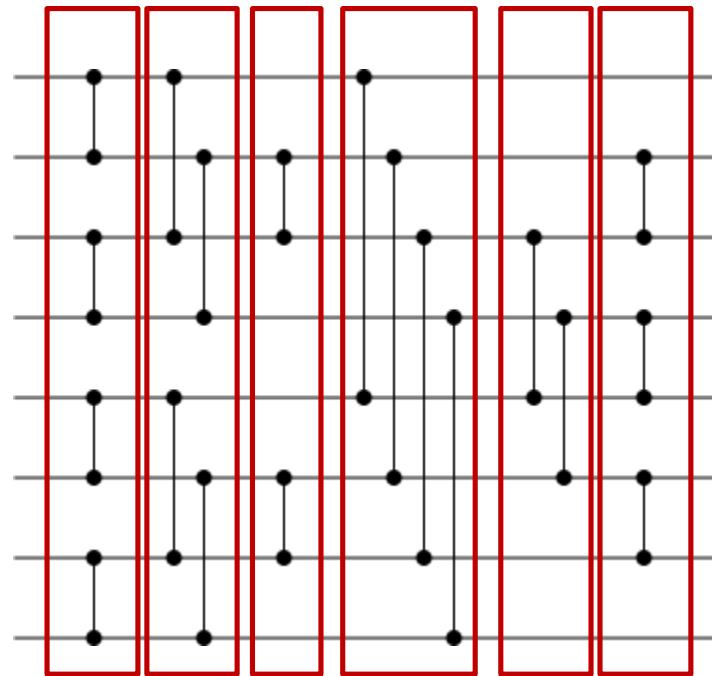
- Využíváme podobnou myšlenku jak v bubble sortu
  - Identifikujeme dvojice čísel, porovnání kterých lze dělat paralelně
- Jak to funguje?
  - Pro 8 prvků



# Paralelní řazení

## Odd-Even Merge Sort

- Využíváme podobnou myšlenku jak v bubble sortu
  - Identifikujeme dvojice čísel, porovnání kterých lze dělat paralelně
- Jak to funguje?
  - Pro 8 prvků
- Obecně?



# Paralelní řazení

## Odd-Even Merge Sort

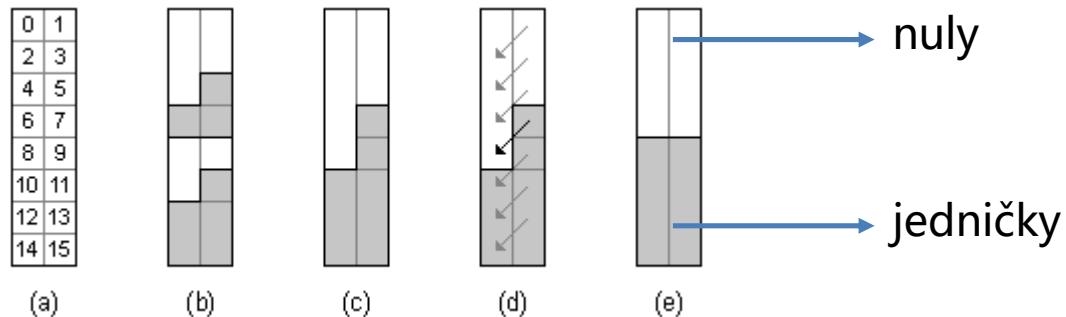
- Využíváme podobnou myšlenku jak v bubble sortu
  - Identifikujeme dvojice čísel, porovnání kterých lze dělat paralelně
- Jak to funguje?
  - Pro 8 prvků
- Obecně?

```
void odd-even-merge (std::vector<int>& vector_to_sort, int from, int to, int step) {
 auto new_step = step * 2;
 if (new_step < to - from) {
 odd-even-merge(vector_to_sort,from,to,new_step);
 odd-even-merge(vector_to_sort,from+step,to,new_step);
 for (int i=from+step; i<to-step; i += new_step) {
 compare_and_swap(vector_to_sort,i,i+step);
 }
 } else {
 compare_and_swap(vector_to_sort,from,from+step);
 }
}
```

# Paralelní řazení

## Odd-Even Merge Sort

- Proč to funguje?
  - Lze dokázat pomocí indukce a tzv. 0-1 principu
    - (pokud třídící síť dokáže setřídit libovolnou posloupnost nul a jedniček, dokáže setřídit libovolnou sekvenci libovolných celých čísel)
  - Předpokládejme (Indukční krok), že algoritmus funguje pro  $n < k$

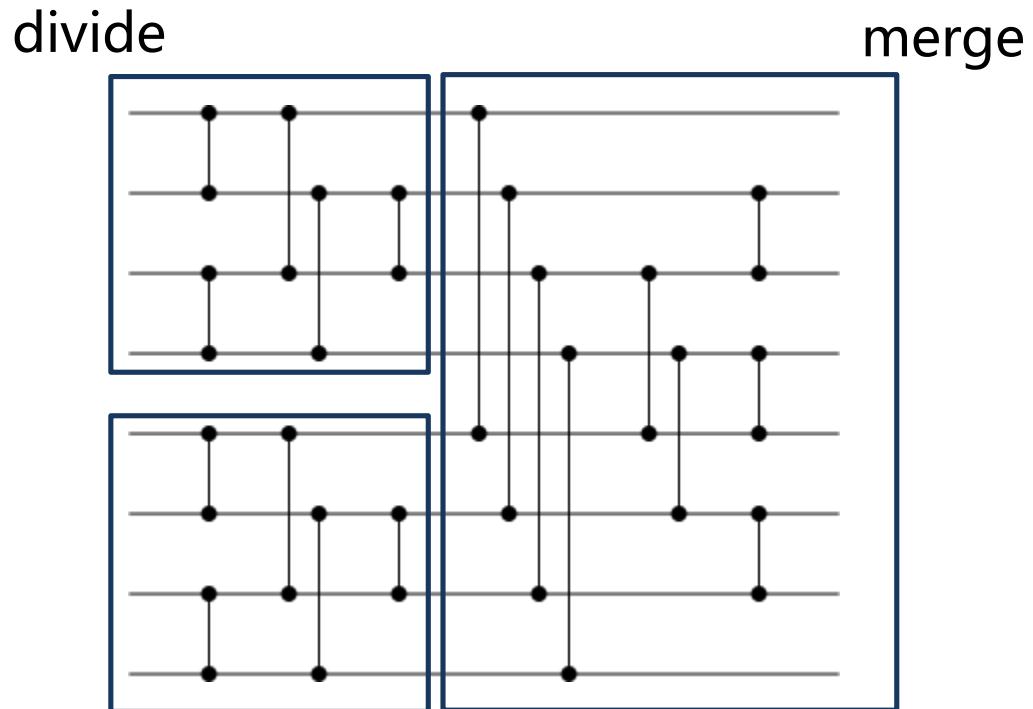


- Ideální pro HW/GPU implementaci
- $O(\log^2(n))$  paralelní výpočetní čas

# Paralelní řazení

## Bitonic Sort

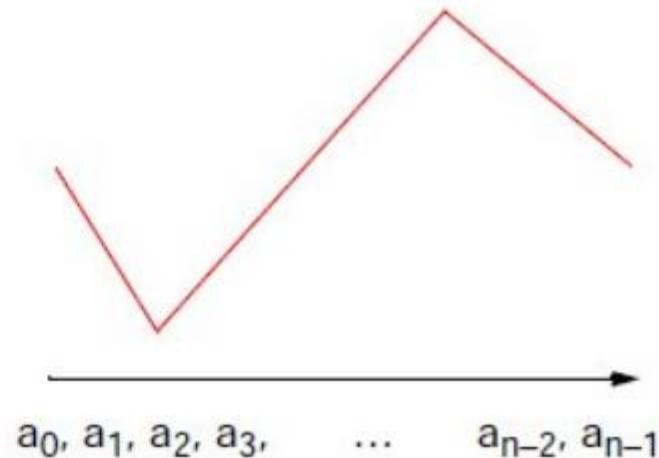
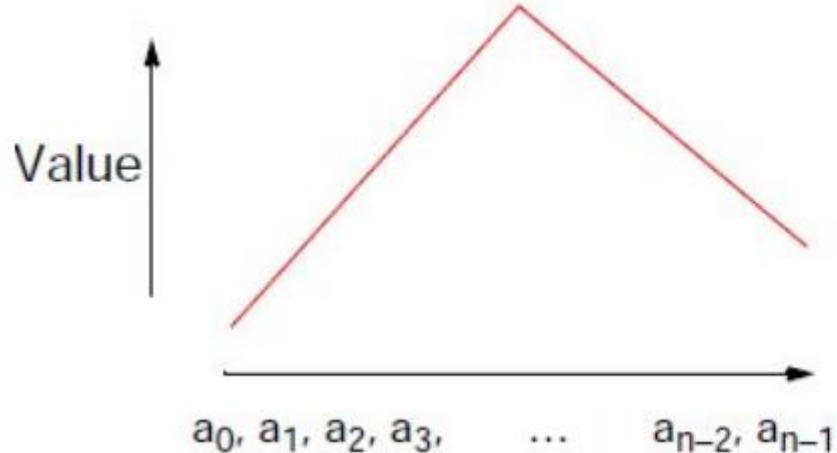
- Bitonic Sort
- Vylepšená varianta Odd-Even Merge Sortu
- Pro paralelní slučování nepotřebujeme mít plně setříděné dílčí sekvence



# Paralelní řazení

## Bitonic Sort

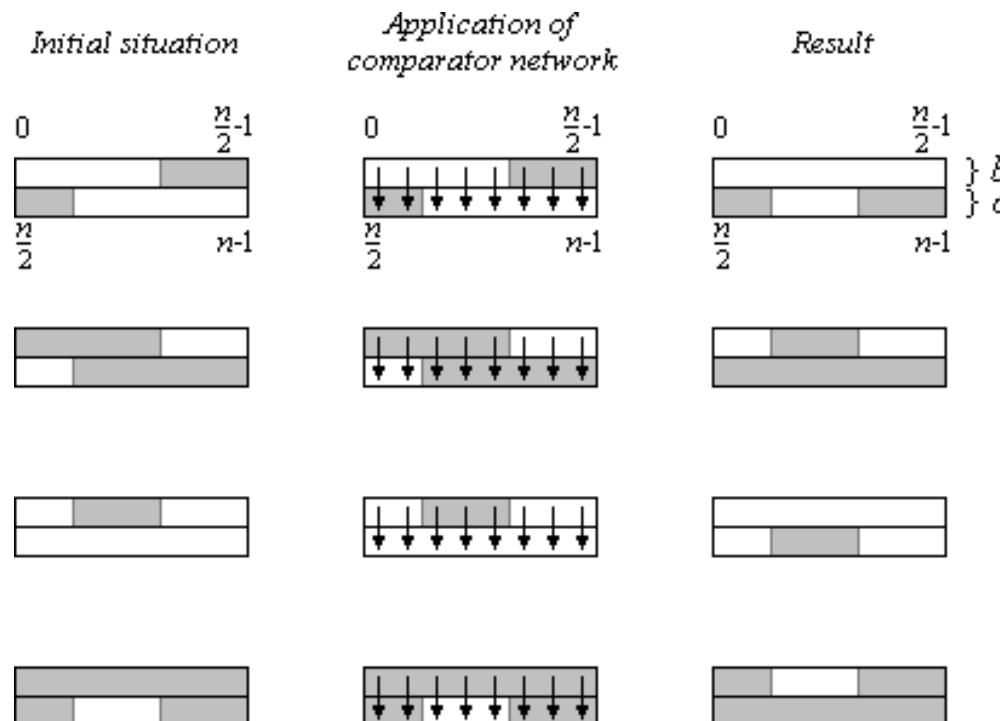
- Sekvence čísel je **bitonická**, pokud
  - obsahuje 2 podsekvence – jednu rostoucí a jednu klesající
    - tedy pro nějaké ( $0 \leq i \leq n$ ) platí
$$a_1 < a_2 < \dots < a_{i-1} < a_i > a_{i+1} > a_{i+2} > \dots > a_n$$
  - nebo lze dosáhnout této vlastnosti pomocí rotací prvků pole



# Paralelní řazení

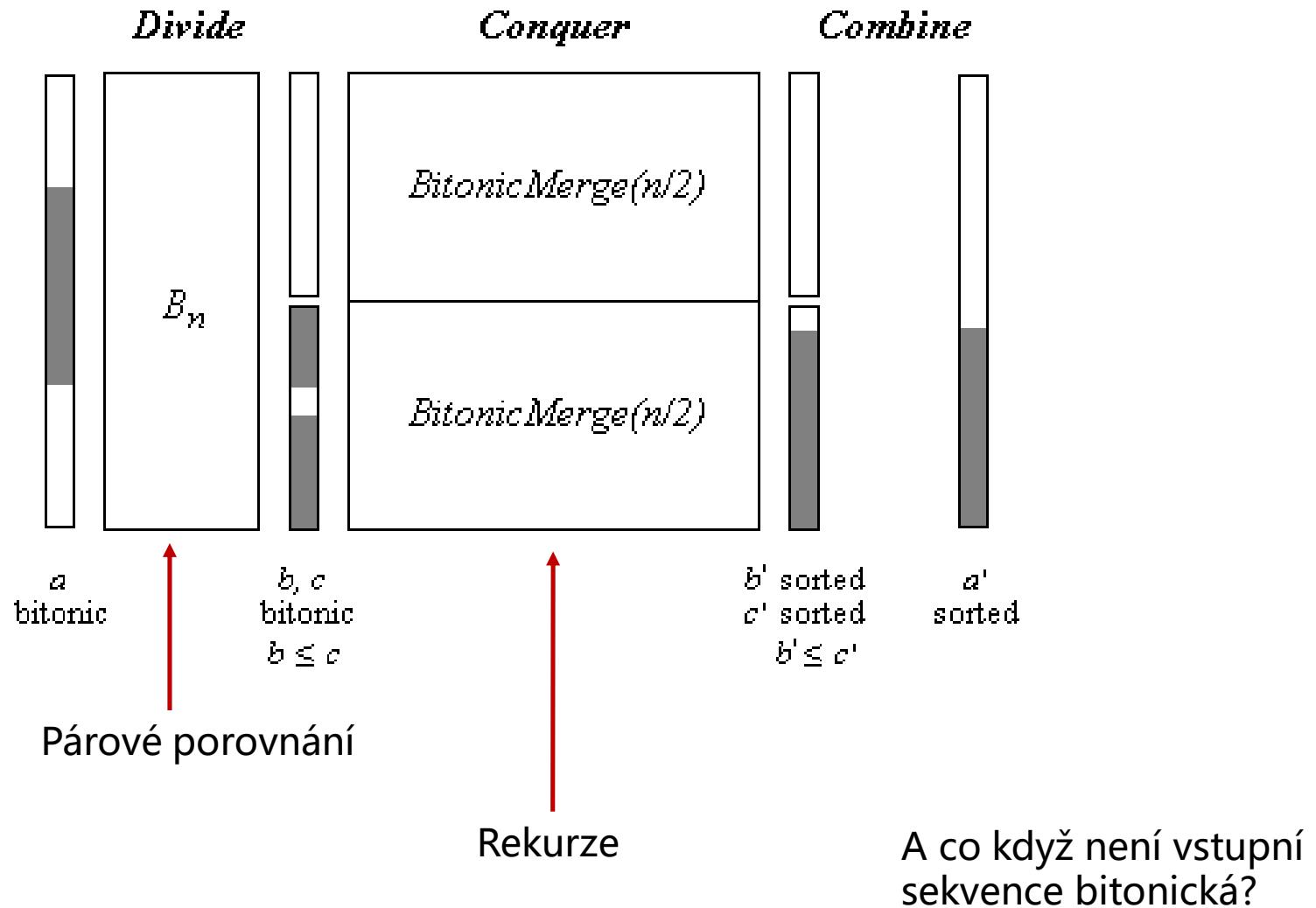
## Bitonic Sort

- Párovým porovnáním prvků dvou částí bitonické sekvence dostaneme 2 bitonické sekvence



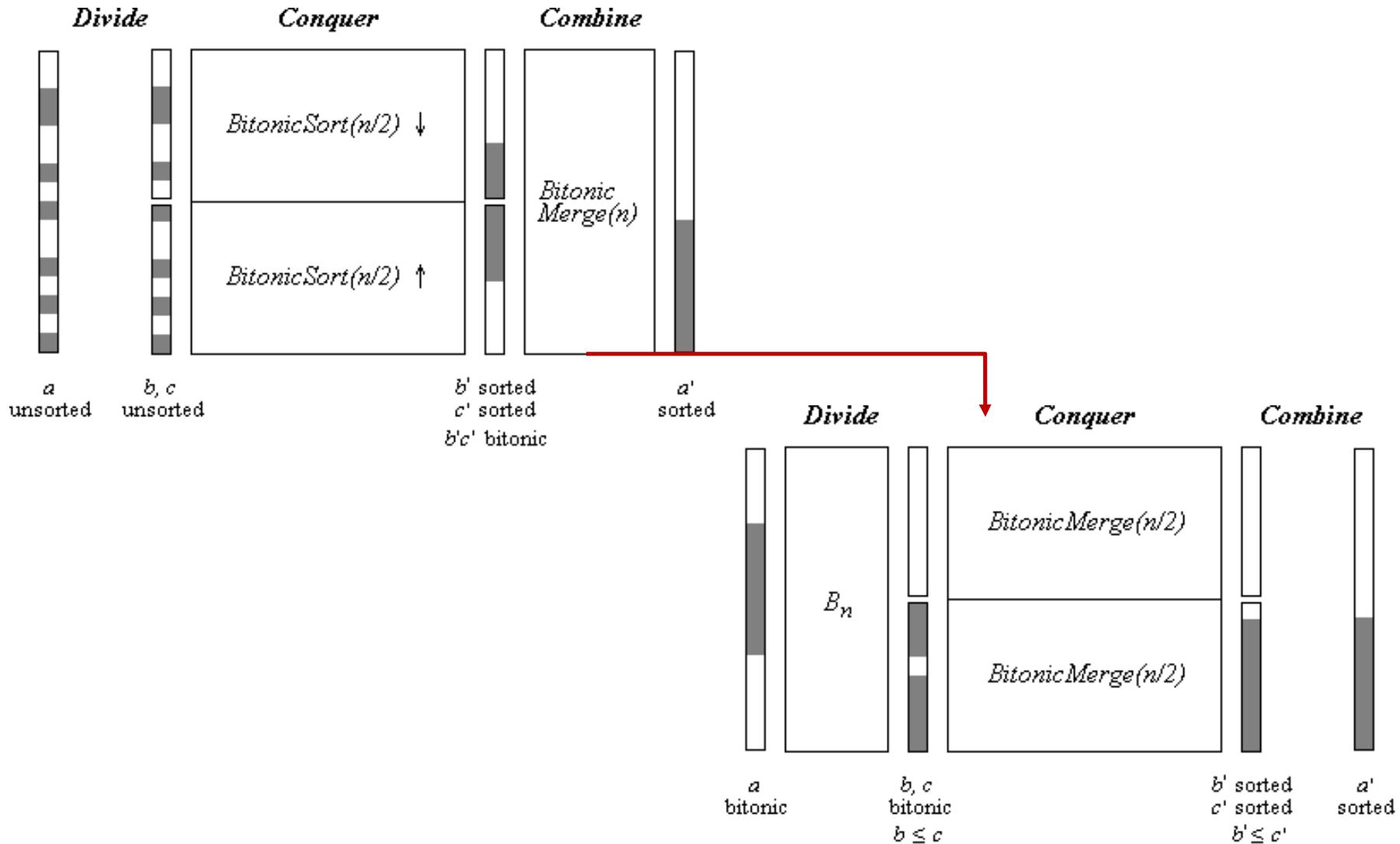
# Paralelní řazení

## Bitonic Sort



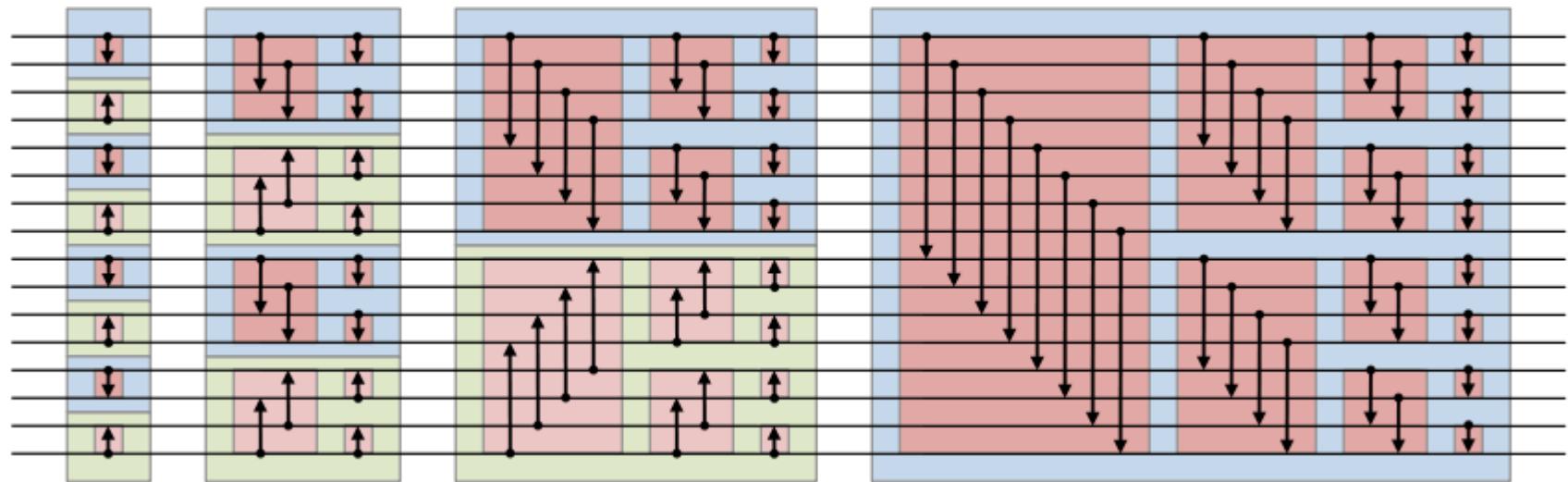
# Paralelní řazení

## Bitonic Sort



# Paralelní řazení

## Bitonic Sort



# Paralelní řazení

## Bitonic Sort

- Jak efektivně implementovat?
- Chceme provést vybranou skupinu porovnání zároveň
  - SIMD typ kroku – chci porovnání a případnou výměnu prvků na vícero datech současně
  - Ideální pro implementaci na GPUs
- Co když GPU nemáme?
- Můžeme využít vektorizaci na dnešních CPUs

# Krátký úvod do vektorových instrukcí

- Myšlenka – jednotlivá čísla polí budeme reprezentovat pomocí vektoru čísel
- Použitím přístupných datových struktur a metod řekneme procesoru, které operace se mohou vykonat paralelně (SIMD)

SSE

| Datové typy         |                                              |
|---------------------|----------------------------------------------|
| <code>_m128</code>  | 128 bitový vektor, obsahuje 4x <b>float</b>  |
| <code>_m128d</code> | 128 bitový vektor, obsahuje 2x <b>double</b> |
| <code>_m128i</code> | 128 bitový vektor, obsahuje celá čísla       |
| <code>_m256</code>  | 256 bitový vektor, obsahuje 8x <b>float</b>  |
| <code>_m256d</code> | 256 bitový vektor, obsahuje 4x <b>double</b> |
| <code>_m256i</code> | 256 bitový vektor, obsahuje celá čísla       |
| ...                 |                                              |

AVX

překládejte s přepínači **-march=native -mavx**

# Krátký úvod do vektorových instrukcí

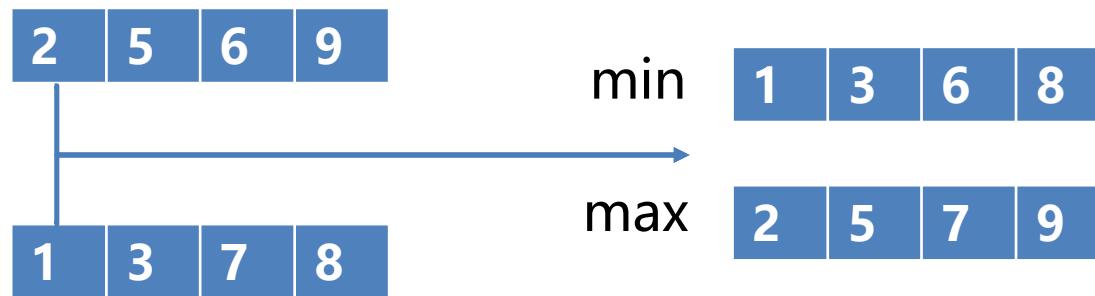
- Počet celých čísel záleží na typu
- Ve `_m256i` může být:
  - 32 char
  - 16 short
  - 8 int
  - 4 long

| Datové typy         |                                              |
|---------------------|----------------------------------------------|
| <code>_m128</code>  | 128 bitový vektor, obsahuje 4x <b>float</b>  |
| <code>_m128d</code> | 128 bitový vektor, obsahuje 2x <b>double</b> |
| <code>_m128i</code> | 128 bitový vektor, obsahuje celá čísla       |
| <code>_m256</code>  | 256 bitový vektor, obsahuje 8x <b>float</b>  |
| <code>_m256d</code> | 256 bitový vektor, obsahuje 4x <b>double</b> |
| <code>_m256i</code> | 256 bitový vektor, obsahuje celá čísla       |
| ...                 |                                              |

- Pole je reprezentované v obráceném pořadí
- `float[4] {0f,1f,2f,3f}` → 

# Krátký úvod do vektorových instrukcí

- Klíčová operace v bitonic sortu
  - Párové porovnání (a případná výměna) prvků v dvou polích
- Jak na to?



- porovnání 4 (8) čísel se provede zároveň

# Krátký úvod do vektorových instrukcí

## Párové porovnání 2 vektorů

```
#include ...
#include <immintrin.h>

int main() {

 std::vector<int> vec1 = std::vector<int>(SIZE);
 std::vector<int> vec2 = std::vector<int>(SIZE);

 for (int i=0; i<SIZE; i++) {
 vec1[i] = rand() % 10000;
 vec2[i] = rand() % 10000;
 }

 auto t_start = std::chrono::high_resolution_clock::now();

 __m256i v1;
 __m256i v2;
 __m256i r1,r2;

 for (int i=0; i<SIZE; i += 8) {
 v1 = _mm256_loadu_si256((__m256i *) &vec1[i]);
 v2 = _mm256_loadu_si256((__m256i *) &vec2[i]);
 r1 = _mm256_min_epi32(v1, v2);
 r2 = _mm256_max_epi32(v1, v2);
 _mm256_storeu_si256((__m256i *) &vec1[i], r1);
 _mm256_storeu_si256((__m256i *) &vec2[i], r2);
 }

 auto t_end = std::chrono::high_resolution_clock::now();
 double elapsed = std::chrono::duration<double, std::milli>(t_end-t_start).count()/1000.0;

 std::cout << "compared in " << elapsed << " s" << std::endl;
 return 0;
}
```

# Krátký úvod do vektorových instrukcí

## Párové porovnání 2 vektorů

```
#include ...
#include <immintrin.h>

int main() {

 std::vector<int> vec1 = std::vector<int>(SIZE);
 std::vector<int> vec2 = std::vector<int>(SIZE);

 for (int i=0; i<SIZE; i++) {
 vec1[i] = rand() % 10000;
 vec2[i] = rand() % 10000;
 }

 auto t_start = std::chrono::high_resolution_clock::now();

 __m256i v1;
 __m256i v2;
 __m256i r1,r2;

 for (int i=0; i<SIZE; i += 8) {
 v1 = _mm256_loadu_si256((__m256i *) &vec1[i]);
 v2 = _mm256_loadu_si256((__m256i *) &vec2[i]);
 r1 = _mm256_min_epi32(v1, v2);
 r2 = _mm256_max_epi32(v1, v2);
 _mm256_storeu_si256((__m256i *) &vec1[i], r1);
 _mm256_storeu_si256((__m256i *) &vec2[i], r2);
 }

 auto t_end = std::chrono::high_resolution_clock::now();
 double elapsed = std::chrono::duration<double, std::milli>(t_end-t_start).count()/1000.0;

 std::cout << "compared in " << elapsed << " s" << std::endl;
 return 0;
}
```

Načtení dat do  
vektorové  
reprezentace

# Krátký úvod do vektorových instrukcí

## Párové porovnání 2 vektorů

```
#include ...
#include <immintrin.h>

int main() {

 std::vector<int> vec1 = std::vector<int>(SIZE);
 std::vector<int> vec2 = std::vector<int>(SIZE);

 for (int i=0; i<SIZE; i++) {
 vec1[i] = rand() % 10000;
 vec2[i] = rand() % 10000;
 }

 auto t_start = std::chrono::high_resolution_clock::now();

 __m256i v1;
 __m256i v2;
 __m256i r1,r2;

 for (int i=0; i<SIZE; i += 8) {
 v1 = _mm256_loadu_si256((__m256i *) &vec1[i]);
 v2 = _mm256_loadu_si256((__m256i *) &vec2[i]);
 r1 = _mm256_min_epi32(v1, v2);
 r2 = _mm256_max_epi32(v1, v2);

 _mm256_storeu_si256((__m256i *) &vec1[i], r1);
 _mm256_storeu_si256((__m256i *) &vec2[i], r2);
 }

 auto t_end = std::chrono::high_resolution_clock::now();
 double elapsed = std::chrono::duration<double, std::milli>(t_end-t_start).count()/1000.0;

 std::cout << "compared in " << elapsed << " s" << std::endl;
 return 0;
}
```

2 operace  
porovnání  
(lze i pomocí  
jednoho  
porovnání a 1 xor)

# Krátký úvod do vektorových instrukcí

## Párové porovnání 2 vektorů

```
#include ...
#include <immintrin.h>

int main() {

 std::vector<int> vec1 = std::vector<int>(SIZE);
 std::vector<int> vec2 = std::vector<int>(SIZE);

 for (int i=0; i<SIZE; i++) {
 vec1[i] = rand() % 10000;
 vec2[i] = rand() % 10000;
 }

 auto t_start = std::chrono::high_resolution_clock::now();

 __m256i v1;
 __m256i v2;
 __m256i r1,r2;

 for (int i=0; i<SIZE; i += 8) {
 v1 = _mm256_loadu_si256((__m256i *) &vec1[i]);
 v2 = _mm256_loadu_si256((__m256i *) &vec2[i]);
 r1 = _mm256_min_epi32(v1, v2);
 r2 = _mm256_max_epi32(v1, v2);
 _mm256_storeu_si256((__m256i *) &vec1[i], r1);
 _mm256_storeu_si256((__m256i *) &vec2[i], r2);
 }

 auto t_end = std::chrono::high_resolution_clock::now();
 double elapsed = std::chrono::duration<double, std::milli>(t_end-t_start).count()/1000.0;

 std::cout << "compared in " << elapsed << " s" << std::endl;
 return 0;
}
```



Uložení výsledků

# Krátký úvod do vektorových instrukcí

- Párové porovnání (a případná výměna) prvků v poli (např. sousedních)
  - $x_0 ? x_1$  (a případně vyměnit tak, aby  $x_0$  byla menší)
  - $x_2 ? x_3$  (a případně vyměnit tak, aby  $x_2$  byla menší)

| <b>x3</b> | <b>x2</b> | <b>x1</b> | <b>x0</b> |
|-----------|-----------|-----------|-----------|
| 2         | 5         | 6         | 9         |

- Jak na to?

# Krátký úvod do vektorových instrukcí

- Párové porovnání (a případná výměna) prvků v poli (např. sousedních)
  - $x_0 ? x_1$  (a případně vyměnit tak, aby  $x_0$  byla menší)
  - $x_2 ? x_3$  (a případně vyměnit tak, aby  $x_2$  byla menší)

| x3 | x2 | x1 | x0 |
|----|----|----|----|
| 2  | 5  | 6  | 9  |

- Jak na to?
- Vytvoříme posunutou kopii vektoru a porovnáme

# Krátký úvod do vektorových instrukcí

- Vytvoříme posunutou kopii vektoru a porovnáme
- Metoda alignr – spojí 2 vektory, umožní posun a ořízne

doplníme

|   |   |   |   | x3 | x2 | x1 | x0 |
|---|---|---|---|----|----|----|----|
| 0 | 0 | 0 | 0 | 2  | 5  | 6  | 9  |

# Krátký úvod do vektorových instrukcí

- Vytvoříme posunutou kopii vektoru a porovnáme
- Metoda alignr – spojí 2 vektory, umožní posun a ořízne

doplníme

|   |   |   |   | x3 | x2 | x1 | x0 |
|---|---|---|---|----|----|----|----|
| 0 | 0 | 0 | 0 | 2  | 5  | 6  | 9  |

Shift 1 do prava

|   |   |   |   |   | x3 | x2 | x1 |
|---|---|---|---|---|----|----|----|
| 0 | 0 | 0 | 0 | 0 | 2  | 5  | 6  |

# Krátký úvod do vektorových instrukcí

- Vytvoříme posunutou kopii vektoru a porovnáme
- Metoda alignr – spojí 2 vektory, umožní posun a ořízne

doplníme

|   |   |   |   | x3 | x2 | x1 | x0 |
|---|---|---|---|----|----|----|----|
| 0 | 0 | 0 | 0 | 2  | 5  | 6  | 9  |

Shift 1 do prava

|   |   |   |   |   | x3 | x2 | x1 |
|---|---|---|---|---|----|----|----|
| 0 | 0 | 0 | 0 | 0 | 2  | 5  | 6  |

ořízneme

x3 x2 x1

0 2 5 6

x3 x2 x1 x0

2 5 6 9

Porovnáme s  
původním  
vektorem

# Krátký úvod do vektorových instrukcí

- Vytvoříme posunutou kopii vektoru a porovnáme
- Metoda alignr – spojí 2 vektory, umožní posun a ořízne

doplníme

|   |   |   |   |    |    |    |    |
|---|---|---|---|----|----|----|----|
|   |   |   |   | x3 | x2 | x1 | x0 |
| 0 | 0 | 0 | 0 | 2  | 5  | 6  | 9  |

Shift 1 do prava

|   |   |   |   |    |    |    |   |
|---|---|---|---|----|----|----|---|
|   |   |   |   | x3 | x2 | x1 |   |
| 0 | 0 | 0 | 0 | 0  | 2  | 5  | 6 |

ořízneme

|    |    |    |    |
|----|----|----|----|
|    | x3 | x2 | x1 |
| 0  | 2  | 5  | 6  |
| x3 | x2 | x1 | x0 |
| 2  | 5  | 6  | 9  |

Porovnáme s  
původním  
vektorem



Zajímají nás  
minima

|    |    |    |    |
|----|----|----|----|
| x3 | x2 | x1 | x0 |
| 0  | 2  | 5  | 6  |

# Krátký úvod do vektorových instrukcí

- Vytvoříme posunutou kopii vektoru a porovnáme

Zajímají nás  
minima

|   |   |   |   |
|---|---|---|---|
| 3 | 2 | 1 | 0 |
| 0 | 2 | 5 | 6 |

# Krátký úvod do vektorových instrukcí

- Vytvoříme posunutou kopii vektoru a porovnáme

Zajímají nás minima

|   |   |   |   |
|---|---|---|---|
| 3 | 2 | 1 | 0 |
| 0 | 2 | 5 | 6 |



Ale pouze sudé pozice

- $\min(x_0, x_1)$  je na pozici 0
- $\min(x_2, x_3)$  je na pozici 2
- ...

Vynulujeme pomocí masky

|   |   |   |   |
|---|---|---|---|
| 3 | 2 | 1 | 0 |
| 0 | 2 | 0 | 6 |

# Krátký úvod do vektorových instrukcí

- Vytvoříme posunutou kopii vektoru a porovnáme

Zajímají nás minima

|   |   |   |   |
|---|---|---|---|
| 3 | 2 | 1 | 0 |
| 0 | 2 | 5 | 6 |



Ale pouze sudé pozice

- $\min(x_0, x_1)$  je na pozici 0
- $\min(x_2, x_3)$  je na pozici 2
- ...

Vynulujeme pomocí masky

|   |   |   |   |
|---|---|---|---|
| 3 | 2 | 1 | 0 |
| 0 | 2 | 0 | 6 |

Podobně získáme maxima z porovnání a uložíme je na liché pozice



Výsledek je OR těchto vektorů

|   |   |   |   |
|---|---|---|---|
| 3 | 2 | 1 | 0 |
| 5 | 0 | 9 | 0 |

# Krátký úvod do vektorových instrukcí

```
int SIZE = 8;
std::vector<int> vec1 = std::vector<int>(SIZE);

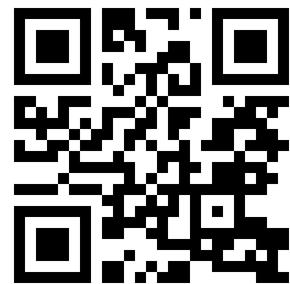
for (int i=0; i<SIZE; i++) {
 vec1[i] = rand() % 10000;
 std::cout << vec1[i] << " ";
}

__m128i mask_llhllhh = _mm_set_epi32(0xffffffff,0,0xffffffff,0);
__m128i mask_hhllhhll = _mm_set_epi32(0,0xffffffff,0,0xffffffff);

__m128i v1;
__m128i v2;
__m128i r1,r2;

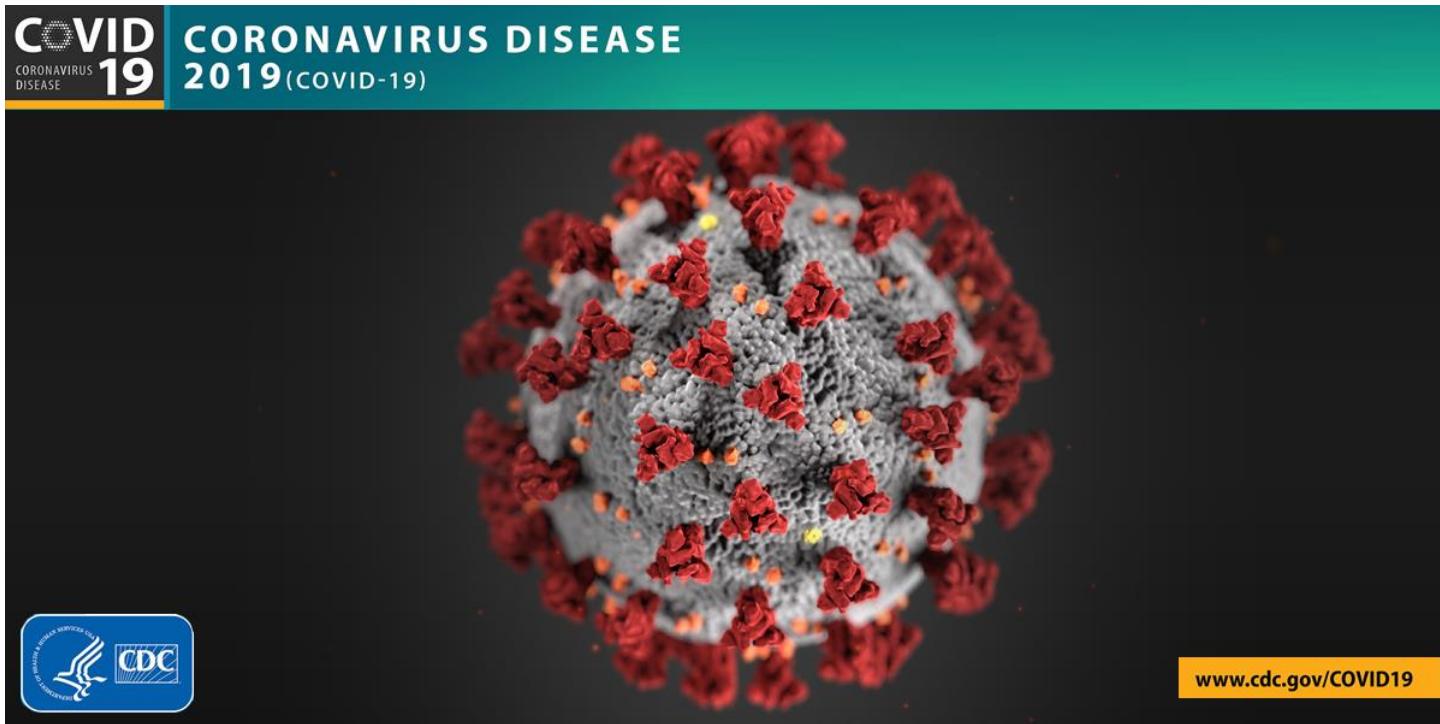
for (int i=0; i<SIZE; i += 4) {
 v1 = _mm_loadu_si128((__m128i *) &vec1[i]);
 v2 = _mm_alignr_epi8(_mm_setzero_si128(), v1 ,1*4);
 r1 = _mm_min_epi32(v1, v2);
 r1 = _mm_and_si128(r1,mask_hhllhhll);
 v2 = _mm_alignr_epi8(v1, _mm_setzero_si128(),3*4);
 r2 = _mm_max_epi32(v1, v2);
 r2 = _mm_and_si128(r2,mask_llhllhh);
 r1 = _mm_or_si128(r1,r2);
 _mm_storeu_si128((__m128i *) &vec1[i], r1);
}
```

# Hlasování



<https://goo.gl/a6BEMb>

# Paralelní a distribuované výpočty (B4B36PDV)



- Přihlašujte se pod Google účtem
- Pokud je to možné, používejte sluchátka
- Pokud nemluvíte, vypněte si mikrofón



# Paralelní a distribuované výpočty (B4B36PDV)

**Branislav Bošanský, Michal Jakob**

[bosansky@fel.cvut.cz](mailto:bosansky@fel.cvut.cz)

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

# Dnešní přednáška

## Motivace



# Dnešní přednáška

## Techniky paralelizace 3

Chci paralelizovat maticový algoritmus



Jak na to?

# Maticové operace

## Násobení matice vektorem

$$\begin{array}{|c|c|c|c|c|} \hline a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ \hline a_{21} & \dots & & & \\ \hline \dots & & & & \\ \hline & & & & a_{55} \\ \hline \end{array} \times \begin{array}{|c|} \hline x_1 \\ \hline x_2 \\ \hline x_3 \\ \hline x_4 \\ \hline x_5 \\ \hline \end{array} = \begin{array}{|c|} \hline y_1 \\ \hline y_2 \\ \hline y_3 \\ \hline y_4 \\ \hline y_5 \\ \hline \end{array}$$

$$y_1 = \sum_{\{i=1,\dots,5\}} a_{1i} \cdot x_i$$

# Maticové operace

## Násobení matice vektorem

- Jak paralelizovat?

$$\begin{array}{|c|c|c|c|c|} \hline a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ \hline a_{21} & \dots & & & \\ \hline \dots & & & & \\ \hline & & & & a_{55} \\ \hline \end{array} \times \begin{array}{|c|} \hline x_1 \\ \hline x_2 \\ \hline x_3 \\ \hline x_4 \\ \hline x_5 \\ \hline \end{array} = \begin{array}{|c|} \hline y_1 \\ \hline y_2 \\ \hline y_3 \\ \hline y_4 \\ \hline y_5 \\ \hline \end{array}$$

$$y_1 = \sum_{\{i=1,\dots,5\}} a_{1i} \cdot x_i$$

# Maticové operace

## Násobení matice vektorem

- Zřejmá paralelizace – výpočet každé složky  $y$  je nezávislý a všechny mohou být spočteny paralelně



$$y_1 = \sum_{\{i=1,\dots,5\}} a_{1i} \cdot x_i$$

# Maticové operace

## Násobení matice vektorem

- Zřejmá paralelizace – výpočet každé složky  $y$  je nezávislý a všechny mohou být spočteny paralelně

```
void multiply(std::vector<int>& A, std::vector<int>& x, std::vector<int>& y) {
 #pragma omp declare reduction(vec_int_plus : std::vector<int> : \
 std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>()) \
 initializer(omp_priv = omp_orig)

 #pragma omp parallel for num_threads(thread_count) reduction(vec_int_plus : y)
 for (int i=0; i<ROWS; i++) {
 for (int j=0; j<COLS; j++) {
 y[i] += A[i * COLS + j]*x[j];
 }
 }
}
```

# Maticové operace

## Násobení matice vektorem

- Zřejmá paralelizace – výpočet každé složky  $y$  je nezávislý a všechny mohou být spočteny paralelně

```
void multiply(std::vector<int>& A, std::vector<int>& x, std::vector<int>& y) {
 #pragma omp declare reduction(vec_int_plus : std::vector<int> : \
 std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>()) \
 initializer(omp_priv = omp_orig)

 #pragma omp parallel for num_threads(thread_count) reduction(vec_int_plus : y)
 for (int i=0; i<ROWS; i++) {
 for (int j=0; j<COLS; j++) {
 y[i] += A[i * COLS + j]*x[j];
 }
 }
}
```

- Co můžeme zlepšit?

# Maticové operace

## Násobení matice vektorem

- Zřejmá paralelizace – výpočet každé složky  $y$  je nezávislý a všechny mohou být spočteny paralelně

```
void multiply(std::vector<int>& A, std::vector<int>& x, std::vector<int>& y) {
 #pragma omp declare reduction(vec_int_plus : std::vector<int> : \
 std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>()) \
 initializer(omp_priv = omp_orig)

 #pragma omp parallel for num_threads(thread_count) reduction(vec_int_plus : y)
 for (int i=0; i<ROWS; i++) {
 for (int j=0; j<COLS; j++) {
 y[i] += A[i * COLS + j]*x[j];
 }
 }
}
```

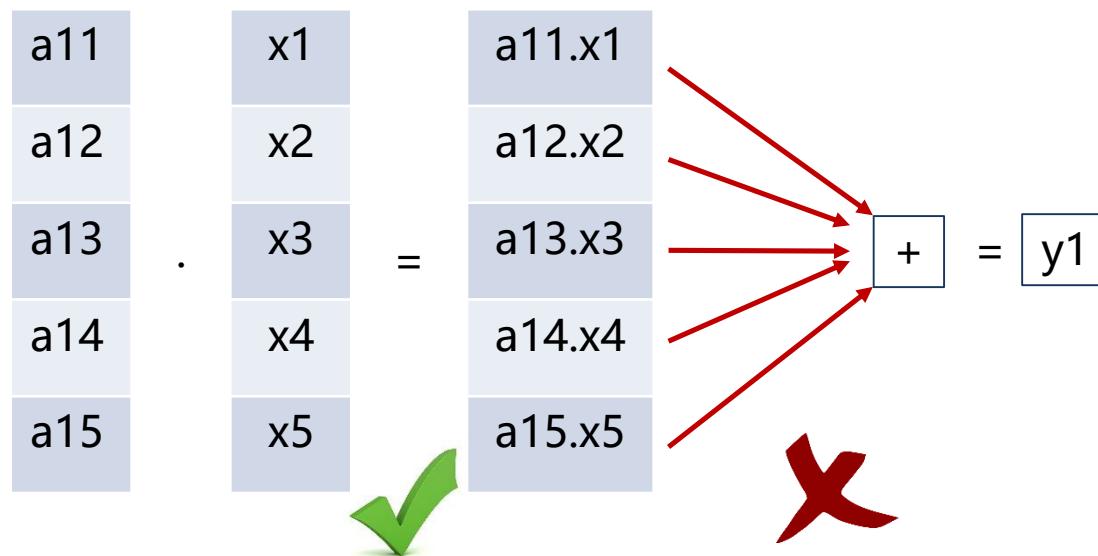
- Co můžeme zlepšit?
- Lze využít vektorizaci?

# Maticové operace

## Násobení matice vektorem

```
void multiply(std::vector<int>& A, std::vector<int>& x, std::vector<int>& y) {
#pragma omp declare reduction(vec_int_plus : std::vector<int> : \
 std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>()) \
 initializer(omp_priv = omp_orig)

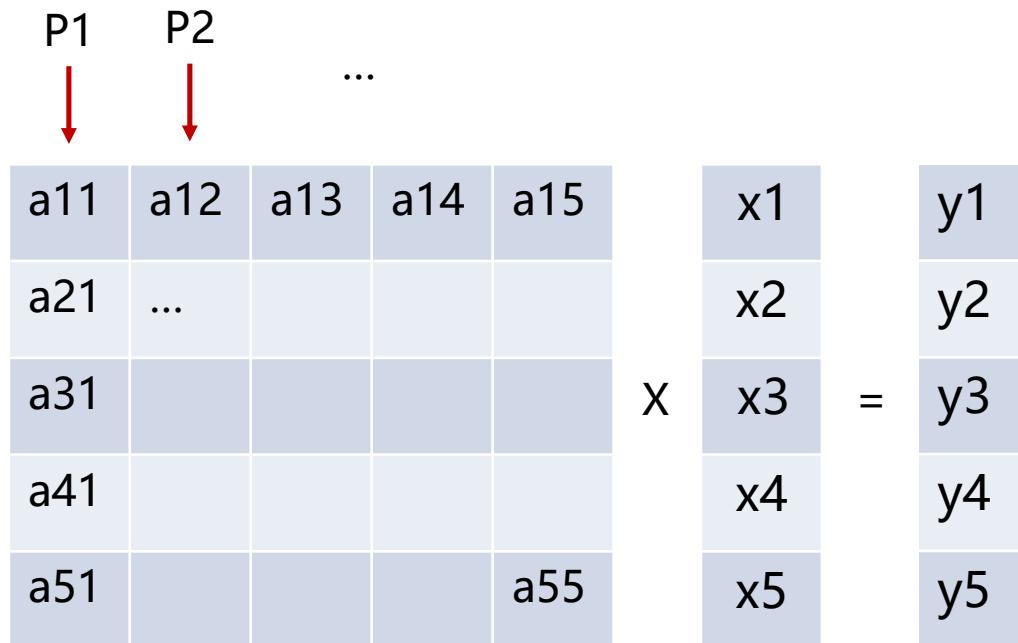
#pragma omp parallel for num_threads(thread_count) reduction(vec_int_plus : y)
 for (int i=0; i<ROWS; i++) {
 for (int j=0; j<COLS; j++) {
 y[i] += A[i * COLS + j]*x[j];
 }
 }
}
```



# Maticové operace

## Násobení matice vektorem

- Co když budeme násobit po sloupcích?



$$z_{ij} = a_{ij} \cdot x_j$$

$$y_i = \sum_{\{j=1, \dots, 5\}} z_{ij}$$

# Maticové operace

## Násobení matice vektorem

|     |    |        |
|-----|----|--------|
| a11 | x1 | a11.x1 |
| a21 | x1 | a21.x1 |
| a31 | x1 | a31.x1 |
| a41 | x1 | a41.x1 |
| a51 | x1 | a51.x1 |
| a12 | x2 | a12.x2 |
| a22 | x2 | a22.x2 |
| a32 | x2 | a32.x2 |
| a42 | x2 | a42.x2 |
| a52 | x2 | a52.x2 |

z1

z2

|        |        |    |
|--------|--------|----|
| a11.x1 | a12.x2 | y1 |
| a21.x1 | a22.x2 | y2 |
| a31.x1 | a32.x2 | y3 |
| a41.x1 | a42.x2 | y4 |
| a51.x1 | a52.x2 | y5 |



# Maticové operace

Násobení matice vektorem

- Bude to fungovat?

# Maticové operace

## Násobení matice vektorem

- Bude to fungovat?
  - Operace sice mohou být vektorizovány, nicméně přístup není vhodný pro cache (mnoho dotazů)
  - Můžeme data v matici uspořádat po sloupcích

|     |     |     |     |     |
|-----|-----|-----|-----|-----|
| a11 | a12 | a13 | a14 | a15 |
| a21 | ... |     |     |     |
| a31 |     |     |     |     |
| a41 |     |     |     |     |
| a51 |     |     |     | a55 |



|     |     |     |     |     |     |  |  |
|-----|-----|-----|-----|-----|-----|--|--|
| a11 | a21 | a31 | a41 | a51 | ... |  |  |
|-----|-----|-----|-----|-----|-----|--|--|

# Maticové operace

## Násobení matice vektorem

- Bude to teď fungovat?

```
...
// data has to be ordered by columns in memory
for (int i = 0; i < COLS; i++) {
 x[i] = rand() % 1000;
 for (int j = 0; j < ROWS; j++) {
 A[i * ROWS + j] = rand() % 1000;
 }
}
...
void multiply_column(std::vector<int> &A, std::vector<int> &x, std::vector<int> &y) {
#pragma omp declare reduction(vec_int_plus : std::vector<int> : \
 std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>()) \
 initializer(omp_priv = omp_orig)

#pragma omp parallel for num_threads(thread_count) reduction(vec_int_plus : y)
 for (int i = 0; i < COLS ; i ++) {
 for (int j = 0; j < ROWS; j++) {
 y[j] += A[i * ROWS + j]*x[i];
 }
 }
}
```

# Maticové operace

## Násobení matice vektorem

- Lze dále zefektivnit původní přístup?
  - Vzpomeňte si na falsesharing ...

```
void multiply(std::vector<int>& A, std::vector<int>& x, std::vector<int>& y) {
#pragma omp declare reduction(vec_int_plus : std::vector<int> : \
 std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>()) \
 initializer(omp_priv = omp_orig)

#pragma omp parallel for num_threads(thread_count) reduction(vec_int_plus : y)
 for (int i=0; i<ROWS; i++) {
 for (int j=0; j<COLS; j++) {
 y[i] += A[i * COLS + j]*x[j];
 }
 }
}
```

# Maticové operace

## Násobení matice vektorem

- Lze dále zefektivnit původní přístup?
  - Vzpomeňte si na falsesharing ...

```
void multiply(std::vector<int>& A, std::vector<int>& x, std::vector<int>& y) {
 #pragma omp declare reduction(vec_int_plus : std::vector<int> : \
 std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>()) \
 initializer(omp_priv = omp_orig))

 #pragma omp parallel for num_threads(thread_count) reduction(vec_int_plus : y)
 for (int i=0; i<ROWS; i++) {
 for (int j=0; j<COLS; j++) {
 y[i] += A[i * COLS + j]*x[j];
 }
 }
}
```

- Nahradíme pole lokální proměnnou

# Maticové operace

## Násobení matice vektorem

- Lokální proměnná

```
void multiply(std::vector<int> &A, std::vector<int> &x, std::vector<int> &y) {
#pragma omp declare reduction(vec_int_plus : std::vector<int> : \
 std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>()) \
 initializer(omp_priv = omp_orig))

 int tmp;
#pragma omp parallel for num_threads(thread_count) reduction(vec_int_plus : y)
 for (int i=0; i<ROWS; i++) {
 tmp = 0;
 for (int j=0; j<COLS; j++) {
 tmp += A[i * COLS + j]*x[j];
 }
 y[i] += tmp;
 }
}
```

# Maticové operace

## Násobení dvou matic

$$\begin{array}{|c|c|c|} \hline a_{11} & a_{12} & a_{13} \\ \hline a_{21} & \dots & \\ \hline \dots & & \\ \hline \end{array} \times \begin{array}{|c|c|c|} \hline b_{11} & b_{12} & b_{13} \\ \hline b_{21} & \dots & \\ \hline \dots & & \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline c_{11} & c_{12} & c_{13} \\ \hline c_{21} & \dots & \\ \hline \dots & & \\ \hline \end{array}$$

$$c_{ij} = \sum_{\{k=1, \dots, n\}} a_{ik} \cdot b_{kj}$$

# Maticové operace

Násobení dvou matic

$$\begin{array}{|c|c|c|} \hline a_{11} & a_{12} & a_{13} \\ \hline a_{21} & \dots & \\ \hline \dots & & \\ \hline \end{array} \times \begin{array}{|c|c|c|} \hline b_{11} & b_{12} & b_{13} \\ \hline b_{21} & \dots & \\ \hline \dots & & \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline c_{11} & c_{12} & c_{13} \\ \hline c_{21} & \dots & \\ \hline \dots & & \\ \hline \end{array}$$

Výpočet prvků c je opět nezávislý a lze paralelizovat

$$c_{ij} = \sum_{\{k=1, \dots, n\}} a_{ik} \cdot b_{kj}$$

Nevýhody?

Velké množství úloh, malé úlohy

# Maticové operace

## Násobení dvou matic

$$\begin{array}{|c|c|c|} \hline a_{11} & a_{12} & a_{13} \\ \hline a_{21} & \dots & \\ \hline \dots & & \\ \hline \end{array} \times \begin{array}{|c|c|c|} \hline b_{11} & b_{12} & b_{13} \\ \hline b_{21} & \dots & \\ \hline \dots & & \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline c_{11} & c_{12} & c_{13} \\ \hline c_{21} & \dots & \\ \hline \dots & & \\ \hline \end{array}$$

Můžeme zvětšit úkoly spojením několika řádků

```
void multiply(std::vector<int>& A, std::vector<int>& B, std::vector<int>& C) {
 #pragma omp declare reduction(vec_int_plus : std::vector<int> : \
 std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>()) \
 initializer(omp_priv = omp_orig)
 int tmp;
 #pragma omp parallel for collapse(2) num_threads(thread_count) reduction(vec_int_plus : C)
 for (int i=0; i<ROWS; i++) {
 for (int j=0; j<COLS; j++) {
 tmp = 0;
 for (int k=0; k<ROWS; k++) {
 tmp += A[i * COLS + k] * B[k * COLS + j];
 }
 C[i * COLS + j] += tmp;
 }
 }
}
```

# Maticové operace

## Násobení dvou matic

- Rozdělení na bloky
- 1 úkol odpovídá částečnému výsledku submatice

# Maticové operace

## Násobení dvou matic

- Rozdělení na bloky
- 1 úkol odpovídá částečnému výsledku submatice ( např.  $c_{11}, c_{12}, c_{21}, c_{22}$ )

|     |     |     |     |
|-----|-----|-----|-----|
| b11 | b12 | b13 | b14 |
| b21 | ... |     |     |
| ... |     |     |     |
|     |     |     |     |

|     |     |     |     |
|-----|-----|-----|-----|
| a11 | a12 | a13 | a14 |
| a21 | ... |     |     |
| ... |     |     |     |
|     |     |     |     |

|     |     |     |     |
|-----|-----|-----|-----|
| c11 | c12 | c13 | c14 |
| c21 | ... |     |     |
| ... |     |     |     |
|     |     |     |     |

$$c_{11} += a_{11} \cdot b_{11} + a_{12} \cdot b_{21}$$

$$c_{12} += a_{11} \cdot b_{12} + a_{12} \cdot b_{22}$$

...

# Maticové operace

## Násobení dvou matic

```
void multiply_blocks(std::vector<int>& A, std::vector<int>& B, std::vector<int>& C) {
 #pragma omp declare reduction(vec_int_plus : std::vector<int> : \
 std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>())) \
 initializer(omp_priv = omp_orig)

 const int ROWS_IN_BLOCK = 10;
 const int BLOCKS_IN_ROW = ROWS/ROWS_IN_BLOCK;
 int tmp;

#pragma omp parallel for collapse(2) num_threads(thread_count) reduction(vec_int_plus : C) private(tmp)
 for (int br1=0; br1<BLOCKS_IN_ROW; br1++) {
 for (int bb=0; bb<BLOCKS_IN_ROW; bb++) {
 for (int bc2 = 0; bc2 < BLOCKS_IN_ROW; bc2++) {
 for (int r = br1 * ROWS_IN_BLOCK; r < (br1 + 1) * ROWS_IN_BLOCK; r++) {
 for (int c = bc2 * ROWS_IN_BLOCK; c < (bc2 + 1) * ROWS_IN_BLOCK; c++) {
 tmp = 0;
 for (int k = 0; k < ROWS_IN_BLOCK; k++) {
 tmp += A[r * COLS + (k + bb*ROWS_IN_BLOCK)] * B[(bb*ROWS_IN_BLOCK + k) * COLS + c];
 }
 C[r * COLS + c] += tmp;
 }
 }
 }
 }
 }
}
```

# Maticové operace

## Násobení dvou matic

$$\begin{array}{|c|c|c|} \hline a_{11} & a_{12} & a_{13} \\ \hline a_{21} & \dots & \\ \hline \dots & & \\ \hline \end{array} \times \begin{array}{|c|c|c|} \hline b_{11} & b_{12} & b_{13} \\ \hline b_{21} & \dots & \\ \hline \dots & & \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline c_{11} & c_{12} & c_{13} \\ \hline c_{21} & \dots & \\ \hline \dots & & \\ \hline \end{array}$$

- A co dál? Lze využít vektorizaci?
- Můžeme najednou spočítat vektor částečných hodnot?

# Maticové operace

## Násobení dvou matic

$$\begin{array}{|c|c|c|} \hline a_{11} & a_{12} & a_{13} \\ \hline a_{21} & \dots & \\ \hline \dots & & \\ \hline \end{array} \times \begin{array}{|c|c|c|} \hline b_{11} & b_{12} & b_{13} \\ \hline b_{21} & \dots & \\ \hline \dots & & \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline c_{11} & c_{12} & c_{13} \\ \hline c_{21} & \dots & \\ \hline \dots & & \\ \hline \end{array}$$

- A co dál? Lze využít vektorizaci?
- Můžeme najednou spočítat vektor částečných hodnot?
- Co když budeme násobit řádek  $i$  matice A a řádek  $j$  matice B?

$$\begin{array}{|c|c|c|} \hline a_{11} & a_{12} & a_{13} \\ \hline a_{21} & \dots & \\ \hline \dots & & \\ \hline \end{array} \times \begin{array}{|c|c|c|} \hline b_{11} & b_{12} & b_{13} \\ \hline b_{21} & \dots & \\ \hline \dots & & \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline z_{21} & \dots & \\ \hline \dots & & \\ \hline \end{array}$$

# Maticové operace

## Násobení dvou matic



- Opět máme vektor dílčích výsledků  $z$
- Násobení je vektorové, součet různých vektorů z lze také vektorizovat

```
void multiply(std::vector<int>& A, std::vector<int>& B, std::vector<int>& C) {
 #pragma omp declare reduction(vec_int_plus : std::vector<int> : \
 std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<int>()) \
 initializer(omp_priv = omp_orig)

 #pragma omp parallel for collapse(2) num_threads(thread_count) reduction(vec_int_plus : C)
 for (int r1=0; r1<ROWS; r1++) {
 for (int r2=0; r2<ROWS; r2++) {
 for (int k=0; k<ROWS; k++) {
 C[r1 * COLS + k] += A[r1 * COLS + r2] * B[r2 * COLS + k];
 }
 }
 }
}
```

# Maticové operace

## Gaussova eliminace

- Dalším typickým úkolem je řešení soustavy lineárních rovnic
- Lze využít Gaussovou eliminaci

The diagram illustrates the process of Gaussian elimination on a system of linear equations represented by a matrix. On the left, a 3x4 matrix is shown with columns labeled a11, a12, a13 and rows labeled b1, a21, ..., b2, ... . A large blue arrow points to the right, indicating the transformation. On the right, the matrix is transformed into an upper triangular form, where the first row remains a11, a12, a13, b1, and the second row becomes 0, a'22, a'23, b'2. The third row is transformed into 0, 0, a'33, b'3.

|     |     |     |    |
|-----|-----|-----|----|
| a11 | a12 | a13 | b1 |
| a21 | ... |     | b2 |
| ... |     |     | b3 |

→

|     |      |      |     |
|-----|------|------|-----|
| a11 | a12  | a13  | b1  |
| 0   | a'22 | a'23 | b'2 |
| 0   | 0    | a'33 | b'3 |

- Jak můžeme paralelizovat?

# Maticové operace

## Gaussova eliminace

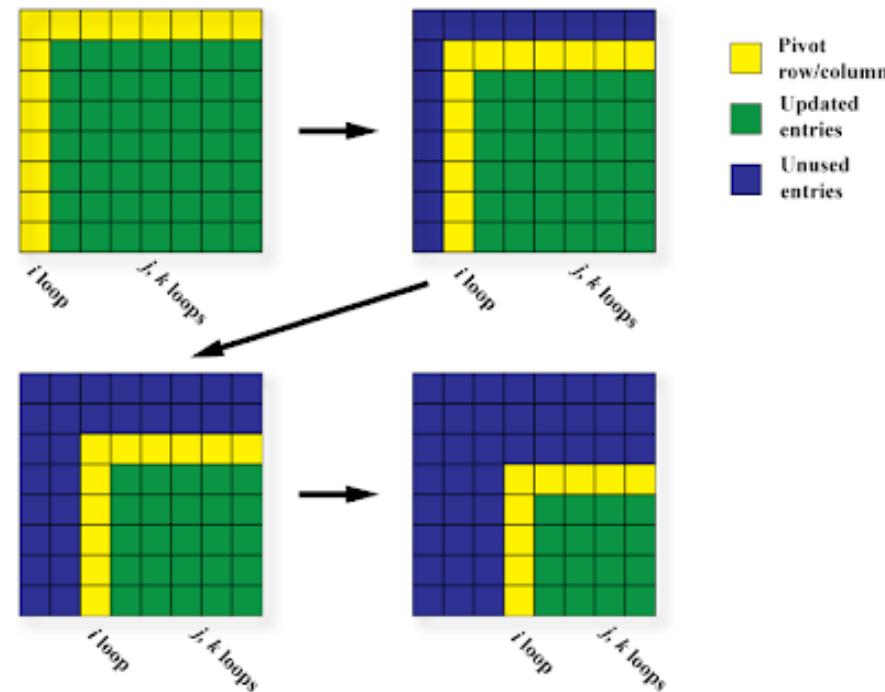
- Jaké jsou závislosti mezi hodnotami?
  - Po výběru pivota se změní všechny hodnoty pro řádky a sloupce větší než pozice pivota

# Maticové operace

## Gaussova eliminace

- Jaké jsou závislosti mezi hodnotami?
  - Po výběru pivota se změní všechny hodnoty pro řádky a sloupce větší než pozice pivota
- Kterou část můžeme parallelizovat?

```
void gauss(std::vector<double>& A) {
 for (int i=0; i<ROWS; i++) {
 // Make all rows below this one 0 in current column
 for (int k=i+1; k<ROWS; k++) {
 double c = -A[k * COLS + i]/A[i*COLS + i];
 for (int j=i; j<ROWS; j++) {
 if (j==j) {
 A[k * COLS + j] = 0;
 } else {
 A[k * COLS + j] += c * A[i * COLS + j];
 }
 }
 }
 }
}
```



# Maticové operace

## Gaussova eliminace

- Jaké jsou závislosti mezi hodnotami?
  - Po výběru pivota se změní všechny hodnoty pro řádky a sloupce větší než pozice pivota

```
void gauss_par(std::vector<double>& A) {
 #pragma omp declare reduction(vec_int_plus : std::vector<double> : \
 std::transform(omp_out.begin(), omp_out.end(), omp_in.begin(), omp_out.begin(), std::plus<double>())) \
 initializer(omp_priv = omp_orig)

 for (int i=0; i<ROWS; i++) {
 // Make all rows below this one 0 in current column
 #pragma omp parallel for num_threads(thread_count)
 for (int k=i+1; k<ROWS; k++) {
 double c = -A[k * COLS + i]/A[i*COLS + i];
 for (int j=i; j<ROWS; j++) {
 if (i==j) {
 A[k * COLS + j] = 0;
 } else {
 A[k * COLS + j] += c * A[i * COLS + j];
 }
 }
 }
 }
}
```

# Maticové operace

## Gaussova eliminace

- Co lze dále zefektivnit?

# Maticové operace

## Gaussova eliminace

- Co lze dál zefektivnit?

|       |       |       |       |       |
|-------|-------|-------|-------|-------|
| (0,0) | (0,1) | (0,2) | (0,3) | (0,4) |
| (1,0) | (1,1) | (1,2) | (1,3) | (1,4) |
| (2,0) | (2,1) | (2,2) | (2,3) | (2,4) |
| (3,0) | (3,1) | (3,2) | (3,3) | (3,4) |
| (4,0) | (4,1) | (4,2) | (4,3) | (4,4) |

(a) Iteration k = 0 starts

|       |              |              |              |              |
|-------|--------------|--------------|--------------|--------------|
| 1     | (0,1)        | (0,2)        | (0,3)        | (0,4)        |
| (1,0) | <u>(1,1)</u> | <u>(1,2)</u> | <u>(1,3)</u> | <u>(1,4)</u> |
| (2,0) | (2,1)        | (2,2)        | (2,3)        | (2,4)        |
| (3,0) | (3,1)        | (3,2)        | (3,3)        | (3,4)        |
| (4,0) | (4,1)        | (4,2)        | (4,3)        | (4,4)        |

(b)

|       |              |              |              |              |
|-------|--------------|--------------|--------------|--------------|
| 1     | (0,1)        | (0,2)        | (0,3)        | (0,4)        |
| (1,0) | <u>(1,1)</u> | <u>(1,2)</u> | <u>(1,3)</u> | <u>(1,4)</u> |
| (2,0) | <u>(2,1)</u> | <u>(2,2)</u> | <u>(2,3)</u> | <u>(2,4)</u> |
| (3,0) | (3,1)        | (3,2)        | (3,3)        | (3,4)        |
| (4,0) | (4,1)        | (4,2)        | (4,3)        | (4,4)        |

(c)

|       |              |              |              |              |
|-------|--------------|--------------|--------------|--------------|
| 1     | (0,1)        | (0,2)        | (0,3)        | (0,4)        |
| (1,0) | <u>(1,1)</u> | <u>(1,2)</u> | <u>(1,3)</u> | <u>(1,4)</u> |
| (2,0) | <u>(2,1)</u> | <u>(2,2)</u> | <u>(2,3)</u> | <u>(2,4)</u> |
| (3,0) | <u>(3,1)</u> | <u>(3,2)</u> | <u>(3,3)</u> | <u>(3,4)</u> |
| (4,0) | (4,1)        | (4,2)        | (4,3)        | (4,4)        |

(d)

|       |              |              |              |              |
|-------|--------------|--------------|--------------|--------------|
| 1     | (0,1)        | (0,2)        | (0,3)        | (0,4)        |
| 0     | <u>(1,1)</u> | <u>(1,2)</u> | <u>(1,3)</u> | <u>(1,4)</u> |
| (2,0) | (2,1)        | (2,2)        | (2,3)        | (2,4)        |
| (3,0) | (3,1)        | (3,2)        | (3,3)        | (3,4)        |
| (4,0) | <u>(4,1)</u> | <u>(4,2)</u> | <u>(4,3)</u> | <u>(4,4)</u> |

(e) Iteration k = 1 starts

|       |              |              |              |              |
|-------|--------------|--------------|--------------|--------------|
| 1     | (0,1)        | (0,2)        | (0,3)        | (0,4)        |
| 0     | <u>(1,1)</u> | <u>(1,2)</u> | <u>(1,3)</u> | <u>(1,4)</u> |
| (2,0) | <u>(2,1)</u> | <u>(2,2)</u> | <u>(2,3)</u> | <u>(2,4)</u> |
| (3,0) | (3,1)        | (3,2)        | (3,3)        | (3,4)        |
| (4,0) | <u>(4,1)</u> | <u>(4,2)</u> | <u>(4,3)</u> | <u>(4,4)</u> |

(f)

|       |              |              |              |              |
|-------|--------------|--------------|--------------|--------------|
| 1     | (0,1)        | (0,2)        | (0,3)        | (0,4)        |
| 0     | <u>(1,1)</u> | <u>(1,2)</u> | <u>(1,3)</u> | <u>(1,4)</u> |
| 0     | <u>(2,1)</u> | <u>(2,2)</u> | <u>(2,3)</u> | <u>(2,4)</u> |
| 0     | <u>(3,1)</u> | <u>(3,2)</u> | <u>(3,3)</u> | <u>(3,4)</u> |
| (4,0) | (4,1)        | (4,2)        | (4,3)        | (4,4)        |

(g) Iteration k = 0 ends

|   |              |              |              |              |
|---|--------------|--------------|--------------|--------------|
| 1 | (0,1)        | (0,2)        | (0,3)        | (0,4)        |
| 0 | <u>(1,1)</u> | <u>(1,2)</u> | <u>(1,3)</u> | <u>(1,4)</u> |
| 0 | <u>(2,1)</u> | <u>(2,2)</u> | <u>(2,3)</u> | <u>(2,4)</u> |
| 0 | <u>(3,1)</u> | <u>(3,2)</u> | <u>(3,3)</u> | <u>(3,4)</u> |
| 0 | <u>(4,1)</u> | <u>(4,2)</u> | <u>(4,3)</u> | <u>(4,4)</u> |

(h)

# Shrnutí paralelní části

- Základní nástroje pro psaní paralelního programu
  - vlákna a práce s nimi
  - Atomické proměnné
  - OpenMP
  - (Vektorizace, SIMD paralelizace)

# Shrnutí paralelní části

- Základní nástroje pro psaní paralelního programu
  - vlákna a práce s nimi
  - Atomické proměnné
  - OpenMP
  - (Vektorizace, SIMD paralelizace)
- Základní techniky paralelizace
  - Rozděluj a panuj
  - Threadpool
  - Dekompozice, nalezení co možná nejvíce paralelně bezkonfliktně vykonatelných operací

# Shrnutí paralelní části

- Základní nástroje pro psaní paralelního programu
  - vlákna a práce s nimi
  - Atomické proměnné
  - OpenMP
  - (Vektorizace, SIMD paralelizace)
- Základní techniky paralelizace
  - Rozděluj a panuj
  - Threadpool
  - Dekompozice, nalezení co možná nejvíce paralelně bezkonfliktně vykonatelných operací
- Základní algoritmy
  - Řazení
  - Maticové operace

# Shrnutí paralelní části

- Paralelizujete s cílem zefektivnit běh programu/algoritmu
- Musíte (alespoň částečně) rozumět vykonávání programu na HW
  - Falsesharing
  - Cache optimization

# Shrnutí paralelní části

- Paralelizujete s cílem zefektivnit běh programu/algoritmu
- Musíte (alespoň částečně) rozumět vykonávání programu na HW
  - Falsesharing
  - Cache optimization
- Vynechali jsme spoustu věcí
  - Úvodní kurz, aby jste získali základní znalosti a zkušenosti
- Pokud Vás paralelní programování zaujalo
  - Paralelní algoritmy (B4M35PAG)
  - Obecné výpočty na grafických procesorech (B4M39GPU)

# Shrnutí paralelní části

- Pro implementační zkoušku – programujte, programujte, programujte!

# Shrnutí paralelní části

- Pro implementační zkoušku – programujte, programujte, programujte!
  - Dostanete problém + sériový algoritmus
  - Cílem bude **zrychlit** algoritmus **díky parallelizaci**
    - **Parallelizace musí být korektní!** (musíte přemýšlet, ne všechny chyby se projeví, paralelní programování je nedeterministické)

# Shrnutí paralelní části

- Pro implementační zkoušku – programujte, programujte, programujte!
  - Dostanete problém + sériový algoritmus
  - Cílem bude **zrychlit** algoritmus **díky parallelizaci**
    - **Parallelizace musí být korektní!** (musíte přemýšlet, ne všechny chyby se projeví, paralelní programování je nedeterministické)
- V dalším studiu/práci – parallelizujte, pokud je to potřeba!
  - Pracujte iterativně – nejdřív je potřeba mít korektní sériovou variantu
  - Pokud je pomalá – zrychlujeme, parallelizujeme, atd.