PROCESY, VLÁKNA, PARALELNÍ VÝPOČTY

Kurz: **Datové struktury a algoritmy**

Lektor: Doc. Ing. Radim Burget, Ph.D.

Autor: Doc. Ing. Radim Burget, Ph.D.

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

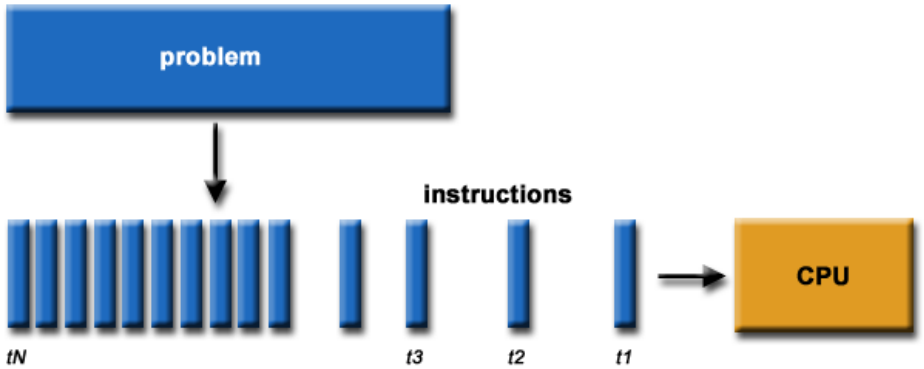
Vytvoření této videopřednášky bylo podpořeno projektem č. CZ.1.07/2.2.00/28.0098
Evropského sociálního fondu a státním rozpočtem České republiky.

Cíle přednášky

1. Druhy přístupu do paměti
2. Paralelní programovací modely
3. Paralelní programování:
   - Vlákna a procesy (ukázka synchronizace vláken)
   - Výpočetní gridy s využitím více počítačů (Hadoop)
   - Heterogenní výpočty (např. s použitím grafických procesorů)

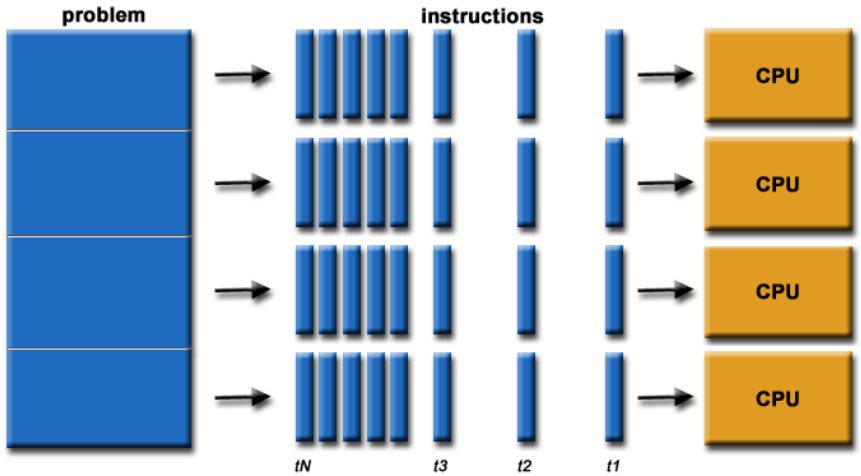
Sériové výpočty

- Běží na jediném CPU s použitím jediného CPU (Central Processing Unit)
- Problém je rozdělen na posloupnost instrukcí
- Každá instrukce je vykonávána jedna po druhé
- V daném čase může být spuštěna pouze jediná instrukce



Paralelní výpočty

- Běží na několika CPU
- Problém je rozdělen na části, ty jsou řešeny paralelně
- Každá část je rozdělena na posloupnosti instrukcí
- Instrukce z každé části jsou poté řešeny souběžně na několika CPU



Sekvenční vs. paralelní

- Veškeré paralelní výpočty lze provádět sekvenčně
- Ne všechny sekvenční algoritmy lze paralelizovat

Třída **NC** je množina rozhodovacích problémů rozhodnutelná v polylogaritmickém čase $O(\log^c n)$ s polynomiálním počtem procesorů $O(n^k)$

$$\mathrm{P} \subseteq \mathrm{NP}$$

Pozn.: $\mathrm{NC} = \mathrm{P}$ či $\mathrm{NC} \neq \mathrm{P}$? Jedná se o stále nevyřešený problém.

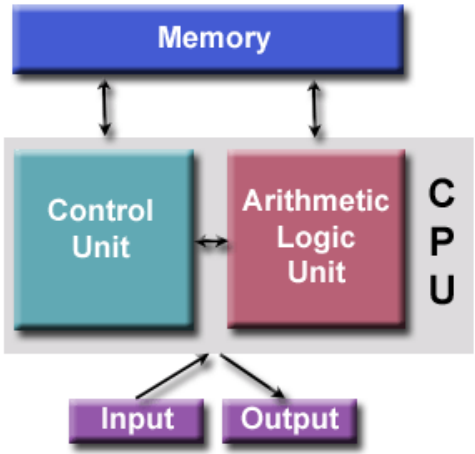
Sériové vs. paralelní modely

Flynnova klasifikace:

- **SISD**
  - Single Instruction, Single Data
- **SIMD**
  - Single Instruction, Multiple Data
- **MISD**
  - Multiple Instruction, Single Data
- **MIMD**
  - Multiple Instruction, Multiple Data

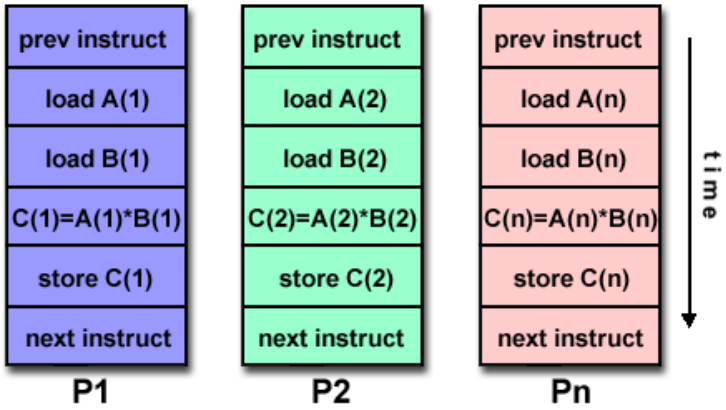
sériový

paralelní



SIMD: Single Instruction, Multiple Data

- V každém okamžiku veškeré procesory vykonávají stejnou instrukci
- Každá výpočetní jednotka může pracovat s libovolnými daty

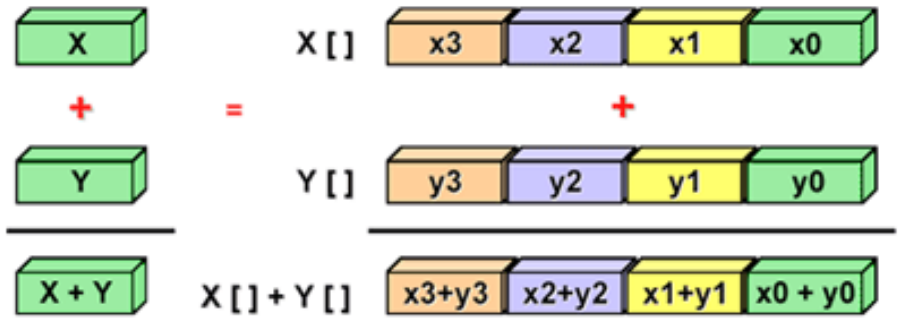


Kde je vhodná:

- Problémy charakteristické velkou mírou pravidelnosti, např. zpracování obrazu/videa, násobení matic, atp.
- Synchronní (lockstep) a deterministické spouštění
- Dvě varianty: Processorová pole a Vektorové pipeliny, NN

SIMD: Příklad

- Sčítání dvou polí (vektorů) A a B.
- Pole jsou rozdělena do bloků
- Každý procesor pracuje s blokem
- Položky pole jsou vybírány na základě ID procesu /vlákna



SIMD: příklad (sečtení dvou polí)

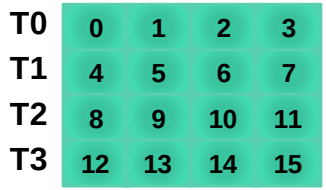
= iterace cyklu

Jedno vláknové (CPU)

```
// mějme N prvků
for(i = 0; i < N; i++)
  C[i] = A[i] + B[i]
```

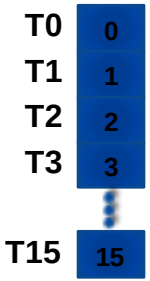
Více vláknové (CPU)

```
// tid je id vlákna
// P je počet jader
for(i = 0; i < tid*N/P; i++)
  C[i] = A[i] + B[i]
```



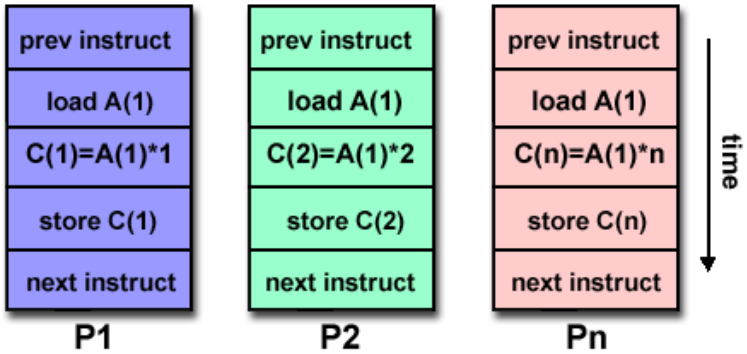
Masivně multivláknové (GPU)

```
// tid je id vlákna
C[tid] = A[tid] + B[tid]
```



MISD: Multiple Instruction, Single Data

- Jeden datový proud je napojen na několik **procesorů**
- Každý procesor pracuje na datech nezávislou streamem instrukcí



Vhodné:

- Vícenásobné frekvenční filtry pracující na jednom signálovém proudu
- Vícenásobné kryptografické algoritmy pokoušející se prolomit jednu zakódovanou zprávu

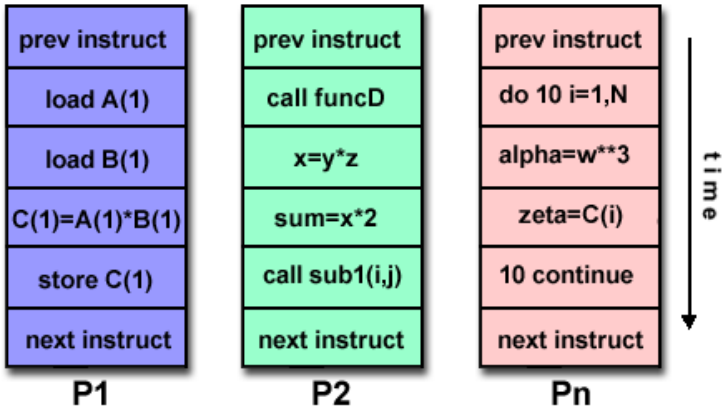
MISD: Příklad

- Vícenásobné vyhledávací algoritmy mohou pracovat se stejnými daty
- Algoritmy mohou používat různou strategii či hledat jiné vzory

Cray Y-MP

MIMD: Multiple Instruction, Multiple Data

- **Multiple Instruction**
  - Procesory zpracovávají různé posloupnosti instrukcí
- **Multiple Data**
  - Procesory pracují s různými datovými proudy



- Spuštění může být synchronizované, asynchronizované, deterministické či ne-deterministické

Vhodné pro:

- Obecné intensivně paralelní výpočty

MIMD: Příklady

- Většina dnešních superpočítačů
- Výpočetní clustery propojené datovou sítí a „gridy“
- Multi-processory SMP počítače
- Vícejádrové PC

Cray XT3

Terminologie: paralelní výpočty

- **Úkol (Task)**
  - Logicky diskrétní sekce výpočetní práce
  - Úkol je typicky program nebo množina instrukcí podobná program, která je vykonávána procesorem
- **Paralelní úkoly (parallel task)**
  - Úkol, který může být bezpečně spuštěn na více procesorech (tj. získává správný výsledek)
- **Sériové spouštění**
  - Spouštění programu sekvenčně, jeden příkaz po druhém
- **Paralelní spouštění (Parallel Execution)**
  - Spouštění programu jako více než jeden úkol, každý úkol může spouštět stejnou či odlišné instrukce v daném čase
- **Řetězení (Pipelining)**
  - Rozdělení úkolů do kroků vykonávaných různými procesory, se vstupy streamovanými skrze procesory

Terminologie: paralelní výpočty

- **Symetrické Multi-Processory (SMP)**
  - Hardwarová architektura, kde více procesorů sdílí jeden paměťový prostor a přistupuje ke všem zdrojům; výpočty se sdílenou pamětí
- **Komunikace**
  - Paralelní úkoly obvyklě vyžadují vyměňovat data.
  - Skrze sdílenou paměťovou sběrnici či po síti
- **Synchronizace**
  - Souhra paralních úkolů v reálném čase, velmi často asociovaných s komunikací.
  - Synchronizace obvykle představuje čekání na alespoň jednu úlohu, a může proto způsobit prodloužení doby vykonání programu.

Paralelní režie

- Množství času potřebného ke koordinaci paralelních úkolů, mimo užitečných prací
- Doba spuštění (inicializace) úlohy
- Datová komunikace
- Softwarové režijní náklady způsobené paralelními kompilátory, knihovnami, nástroji, operačním systémem, atd.
- Čas ukončení úkolu

Granularita

Je kvalitativním měřítkem poměru výpočtu ke komunikaci

- **Hrubé**: mezi komunikačními událostmi se provádí relativně velké množství výpočetní práce
  - Rychlejší interakce s dalšími procesy / uživatelem
- **Jemné**: mezi komunikačními událostmi se provádí relativně malé množství výpočetní práce
  - Vyšší režie
  - Tváří se interaktivně

Škálovatelnost

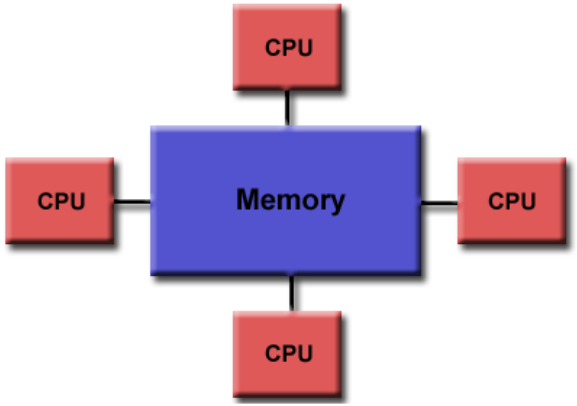
- Vztahuje se na paralelní systém (hardware nebo software) schopnost prokázat přiměřené zvýšení paralelního zrychlení s přidáním více procesorů.
- Hardware - zejména propustnost komunikace paměť-CPU a síťová komunikace
- Aplikační algoritmus
- Paralelní související režie
- Charakteristika vaší konkrétní aplikace a kódu

Architektury paralelní počítačové paměti

- Sdílená paměť
- Distribuovaná paměť
- Hybridní distribuovaná paměť

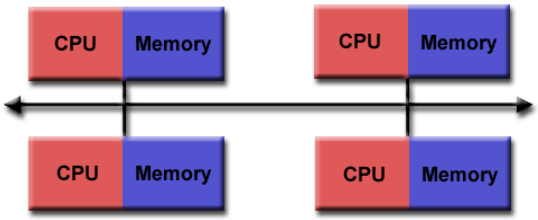
Sdílená paměť

- Sdílená paměť umožňuje všem procesorům přístup ke všem pamětím jako globální adresový prostor.
- Více procesorů může pracovat nezávisle, ale sdílejí stejné paměťové prostředky.
- Změny v paměťovém místě provedené jedním procesorem jsou viditelné pro všechny ostatní procesory.



Distribuovaná paměť 1/2

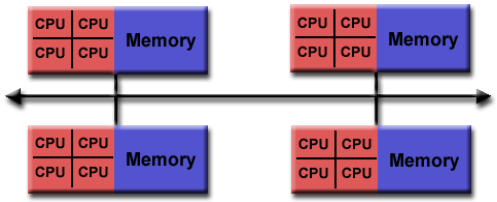
- Distribuované paměťové systémy vyžadují pro propojení meziprocesorové paměti komunikační síť
- Procesory mají vlastní lokální paměť. Paměťové adresy v jednom procesoru nejsou mapovány do jiného procesoru, takže neexistuje žádný koncept globálního adresního prostoru ve všech procesorech.
- Procesory pracují nezávisle
- Programátor obvykle explicitně definuje, jak a kdy jsou data komunikována mezi dvěma procesory.
- Synchronizace mezi úkoly je rovněž odpovědností programátora.
- Síťová struktura (model) používaná pro přenos dat se velmi liší, i když může být stejně jednoduchá jako Ethernet.



Distribuovaná paměť 2/2

- Výhody:
  - Paměť je škálovatelná s počtem procesorů.
  - Každý procesor může rychle přistupovat k vlastní paměti bez rušení a bez režijních nákladů spojených se snahou udržovat koherenci cache.
  - Nákladová efektivita: může využívat komodity, off-the-shelf procesory a sítě.
- Nevýhody:
  - Programátor je zodpovědný za mnoho detailů spojených s datovou komunikací mezi procesory.
  - Může být obtížné namapovat existující datové struktury založené na globální paměti na tuto organizaci paměti.
  - Nekompatibilní přístup do paměti (NUMA)

Hybridní distribuovaná-sdílení paměť



- Komponenta sdílené paměti je obvykle vyrovnávací paměť SMP.
- Procesory na daném SMP mohou adresovat paměť tohoto stroje jako globální
- Komponenta distribuované paměti je zasíťovíání více SMP
- SMP znají pouze svou vlastní paměť - nikoli paměť jiného SMP
- Pro přesouvání dat z jednoho SMP do druhého je vyžadována síťová komunikace.

Jednotný a nejednotný přístup do paměti

Doba přístupu k paměti klasifikuje paralelní práci do:

- Jednotný přístup do paměti
  - Identické procesory
  - Stejný přístup a časy přístupu k paměti
  - Koherence mezipaměti je často prováděna na úrovni hardwaru.
- Non-Uniform Memory Access
  - Často je vytvořen fyzickým propojením dvou nebo více symetrických víceprocesorů (SMP)
  - Jeden SMP může přímo přistupovat k paměti jiného SMP
  - Ne všechny procesory mají stejný přístupový čas ke všem pamětem
  - Přístup přes paměť je pomalejší
  - Pokud je zachována koherence mezipaměti, může být také nazývána **Cache Coherent NUMA**

Jednotný a nejednotný přístup do paměti

- Výhody
  - Globální adresový prostor poskytuje uživatelsky příjemný programovací pohled do paměti
  - Sdílení dat mezi úkoly je rychlé a jednotné díky blízkosti paměti CPU
- Nevýhody:
  - Nedostatek škálovatelnosti mezi pamětí a CPU.
  - Odpovědnost programátora za synchronizační konstrukty, které zajišťují "správný" přístup globální paměti.
- Navrhování a výroba sdílených paměťových počítačů se stále rostoucím počtem procesorů je stále obtížnější a dražší

Paralelní programovací modely

Paralelní programovací modely

- Nejčastěji používané:
  - Sdílená paměť
  - Vlákna
  - Předávání zpráv
  - Data Paralelní
  - Hybridní
- Paralelní programovací modely jsou abstrakce nad architekturou hardwaru a paměti.

Model sdílené paměti

- Úkoly sdílí společný adresní prostor, který čtou a zapisují asynchronně
- Výhodou tohoto modelu z pohledu programátora je, že chybí pojem "vlastnictví", takže není třeba výslovně specifikovat komunikaci dat mezi úkoly.
- Vývoj programu lze často zjednodušit.

Nevýhody

- Je obtížné porozumět a spravovat datovou lokalitu.
  - Uchovávání dat lokálně vůči procesoru, který s nimi pracuje, šetří přístup k paměti, obnovu mezipaměti a provoz sběrnice, ke kterému přistupuje, i když více procesorů používá stejná data.
  - Řídící datová lokalita je bohužel těžko srozumitelná a mimo kontrolu průměrného uživatele

Řízení přístupu ke sdílené paměti

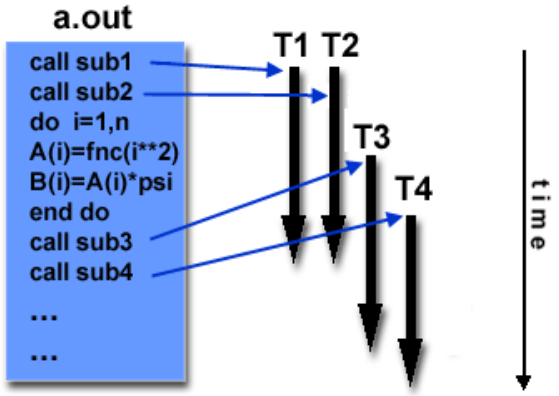
- Zámky
- Semafory

Implementace

- Nativní kompilátory převádějí proměnné uživatelského programu do aktuálních adres paměti, které jsou globální.
- Implementace běžné distribuované paměťové platformy neexistuje.
- Pohled na data sdílené paměti, i když jsou distribuována na fyzické paměti stroje, vypadají jako virtuální sdílená paměť

Vláknový model

- Jeden proces může mít více souběžných cest provádění
- Hlavní program načte a získá všechny potřebné systémové a uživatelské zdroje
- Provádí nějakou sériovou práci a poté vytvoří řadu úloh (podprocesů), které běží souběžně

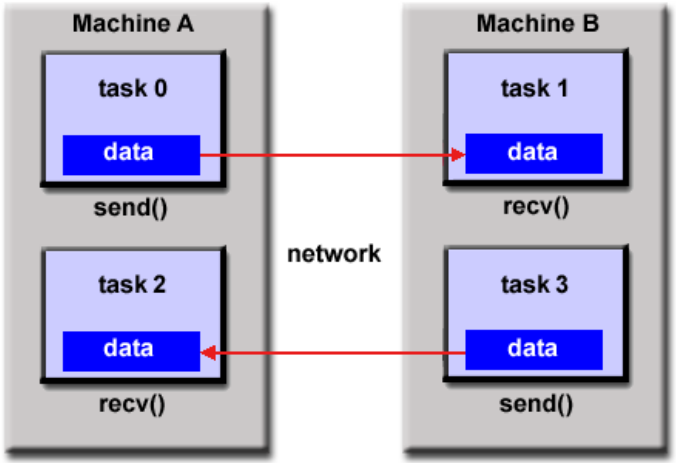


Vláknový model

- Slovo „vláno“ lze popsat jako podprogram v rámci hlavního programu
- Všechny podproces sdílí celý paměťový prostor
- Každé vlákno má lokální data
- Šetří režii - netřeba replikace zdrojů programu
- Vlákna komunikují mezi sebou prostřednictvím globální paměti
- Vlákna vyžadují synchronizaci, aby bylo zajištěno, že více než jeden podproces neaktualizuje stejnou globální adresu
- Vlákna mohou vznikat a skončit, ale hlavní vlákno zůstane přítomno, aby poskytlo potřebné sdílené prostředky, dokud nebude aplikace dokončena.

Model předávání zpráv

- Používá se model předávání zpráv
  - Soubor úloh, které během výpočtu používají vlastní lokální paměť.
  - Více úkolů může být umístěno na stejném fyzickém stroji i přes libovolný počet strojů.

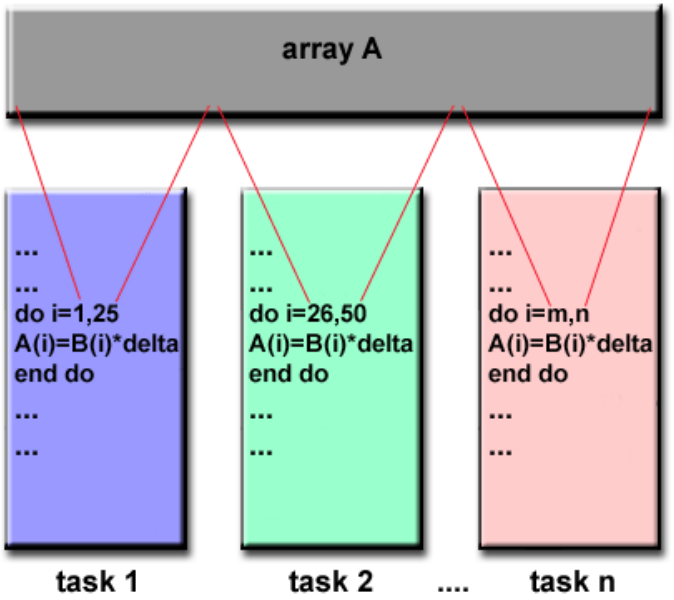


Model předávání zpráv

- Úkoly si prostřednictvím komunikace odesílají a přijímají zprávy.
- Přenos dat obvykle vyžaduje, aby každý proces prováděl předávání data a spolupracovaly.
- Komunikační procesy mohou existovat na stejném stroji (u různých strojů)

Datově paralelní model

- Většina paralelní práce se zaměřuje na provádění operací na datové množině
- Datová množina je obvykle organizována do společné struktury
- Množina úloh pracuje společně na stejné datové struktuře, každý úkol pracuje na jiném oddílu stejné datové struktury



Datově paralelní model

- Úlohy provádějí stejnou operaci při dělení práce.
- Na architekturách sdílené paměti mohou mít všechny úkoly přístup ke struktuře dat prostřednictvím globální paměti. Na architekturách s distribuovanou pamětí je struktura dat rozdělena a je uložena jako "bloky" v místní paměti každého úkolu.

Návrh paralelních algoritmů

- Programátor je obvykle zodpovědný za identifikaci a implementaci paralelismu.
- Manuální vývoj paralelních kódů je časově náročný, složitý, náchylný k chybám a iterativní proces.

- V současné době je nejběžnějším typem nástroje, který se používá pro automatickou paralelizaci sériového programu, paralelní kompilátor nebo preprocesor.

Paralelizující překladač

- Plně automatický
  - Kompilátor analyzuje zdrojový kód a identifikuje příležitosti pro paralelismus
  - Analýza zahrnuje identifikaci inhibitorů paralelismu a možná i nákladovou váhu, zda by paralelismus skutečně zlepšila výkonnost
  - Smyčky (do, for) jsou nejčastějším cílem pro automatickou paralelizaci
- Řízen programátorem
  - Pomocí "kompilátoru direktivy" nebo případně kompilátoru příznaky, programátor výslovně řekne kompilátoru, jak paralelelizovat kód
  - Může být možné použít i ve spojení s určitým stupněm automatické paralelizace

Omezení automatické paralelizace

- Mohou být vytvořeny nesprávné výsledky
- Výkon může veskutečnosti degradovat
- Mnohem méně flexibilní než ruční paralelizace
- Omezeno na podmnožinu (většinou cykly) kódu
- Nemusí ve skutečnosti paralelizovat kód, pokud analýza naznačuje, že existují inhibitory nebo je kód příliš složitý

Problém & Program

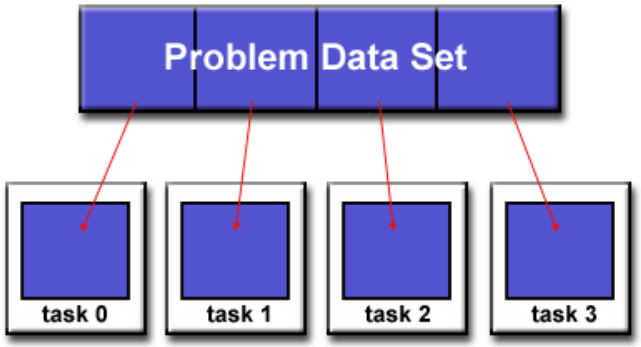
- Zjistěte, zda je problém ten, který může být ve skutečnosti paralelizován
- Identifikujte **hotspoty** programu:
  - Vím, kde se dělá většina skutečné práce
  - Pomoci zde mohou profilery a nástroje pro analýzu výkonnosti
  - Zaměřte se na paralelizaci hotspotů a ignorujte ty části programu, které zodpovídají za malé využití procesoru
- Identifikujte **úzká hrdla** v programu
  - Určete oblasti, kde je program pomalý nebo omezený
  - Může být možné restrukturalizovat program nebo použít jiný algoritmus ke snížení nebo odstranění zbytečných pomalých oblastí
  - Identifikujte inhibitory paralelismu. Jedna běžná třída inhibitorů je závislost na údajích, jak je prokázáno Fibonacciho sekvencí
- Pokud je to možné, zvažte i další algoritmy. To může být jedinou nejdůležitější úvahou při navrhování paralelní aplikace

Rozdělení

- Rozdělte problém do diskrétních "kusů" práce, které lze distribuovat do více úkolů.
  - Doménová dekompozice
  - Funkční dekompozice

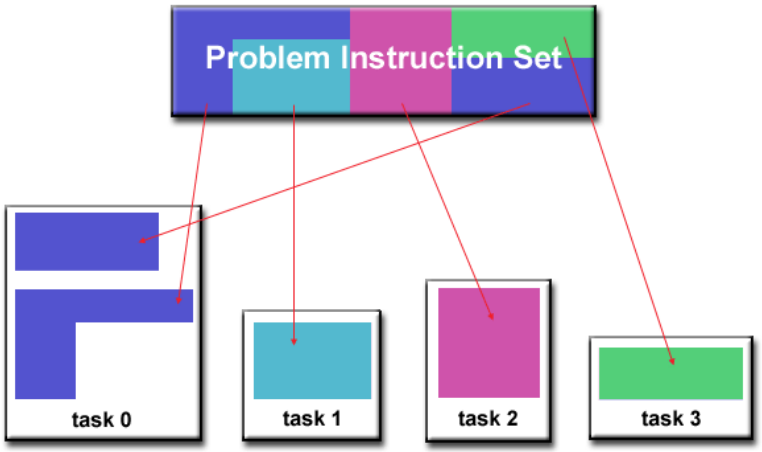
Doménová dekompozice

- Data spojená s problémem se rozdělí
- Každá paralelní úloha pak pracuje na části dat
- Tento oddíl lze provést různými způsoby
  - Řádky, sloupce, bloky, cyklické atd.



Funkční dekompozice

- Problém je rozdělen podle práce, která musí být provedena. Každá úloha pak provede část celkové práce



Komunikace

- Náklady na komunikaci
- Latence vs. šířka pásma
- Viditelnost komunikace
- Synchronní vs. asynchronní komunikace
- Rozsah komunikace
  - Bod-bod
  - Kolektivní
- Účinnost komunikace
- Režie a složitost

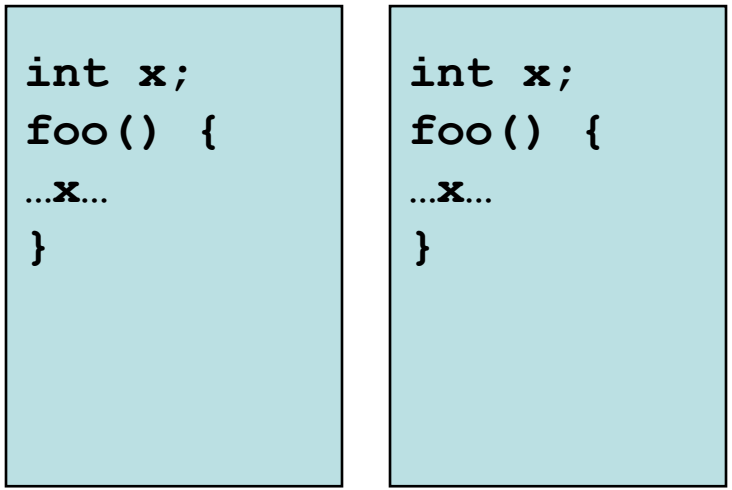
Synchronizace

- Bariéra
- Zámek / semafor
- Synchronní komunikační operace

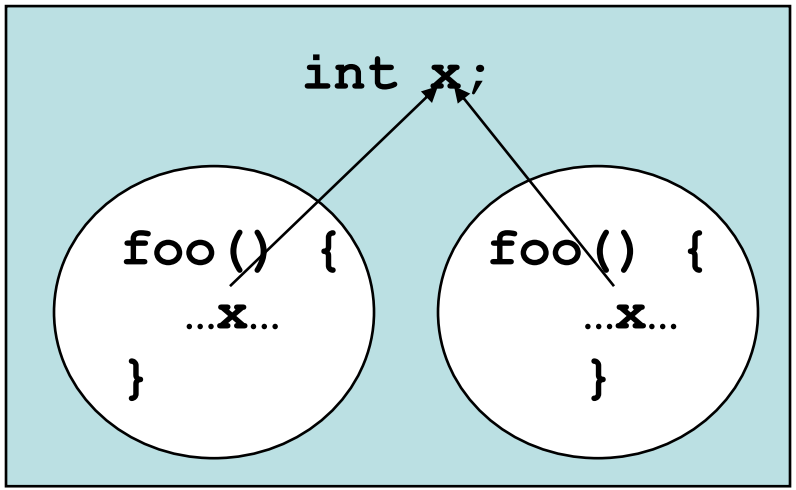
Procesy vs. Vlákna

- Procesy
  - Náročnější na vytvoření (systémové volání)
  - Komunikují předáváním zpráv
  - Náročnější pro programování
  - Dobře škálovatelné (tisíce procesů)
- Vlákna
  - Méně náročné na syst. prostředky
  - Sdílí paměť
  - Snadnější pro programování
  - Omezená škálovatelnost (desítky vláken)

Procesy vs. Vlákna

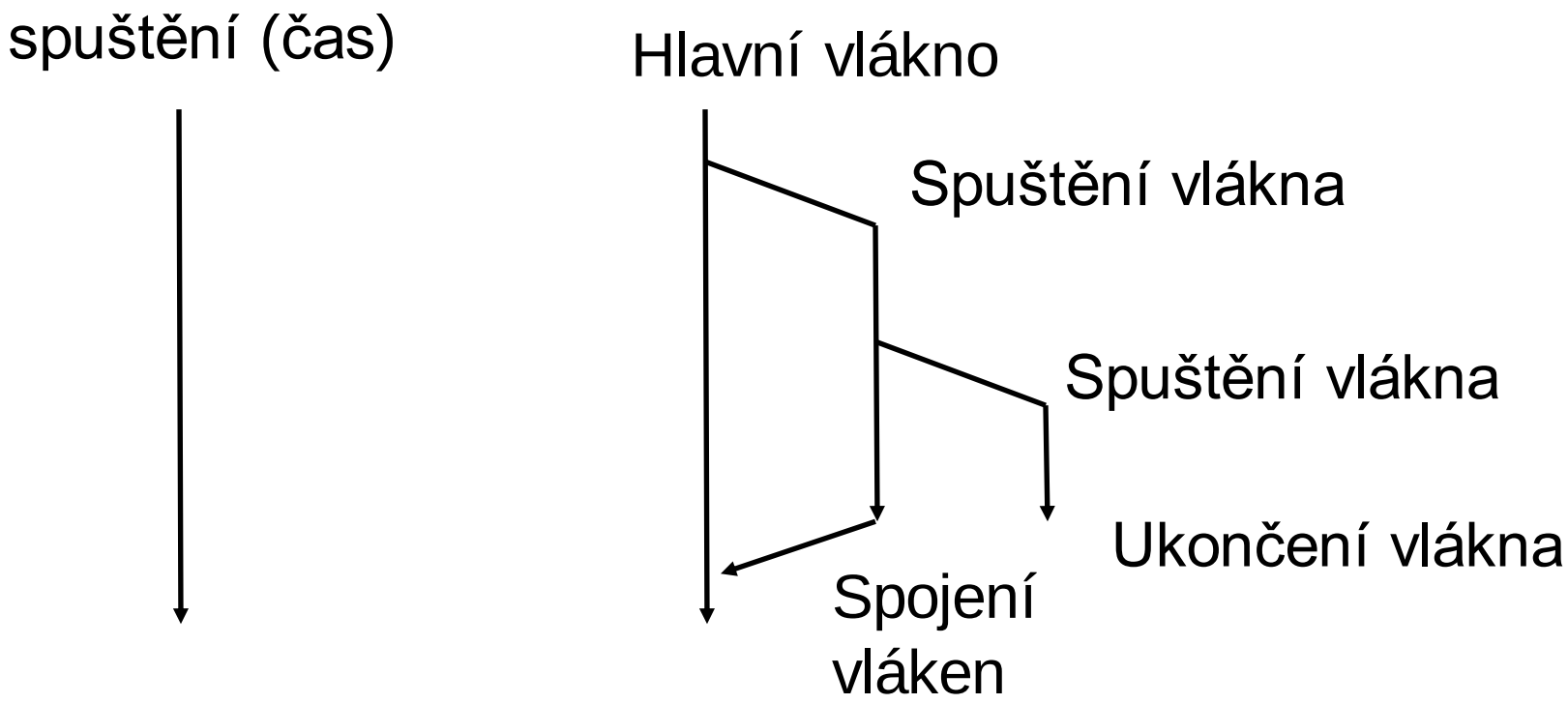


Procesy nesdílí data

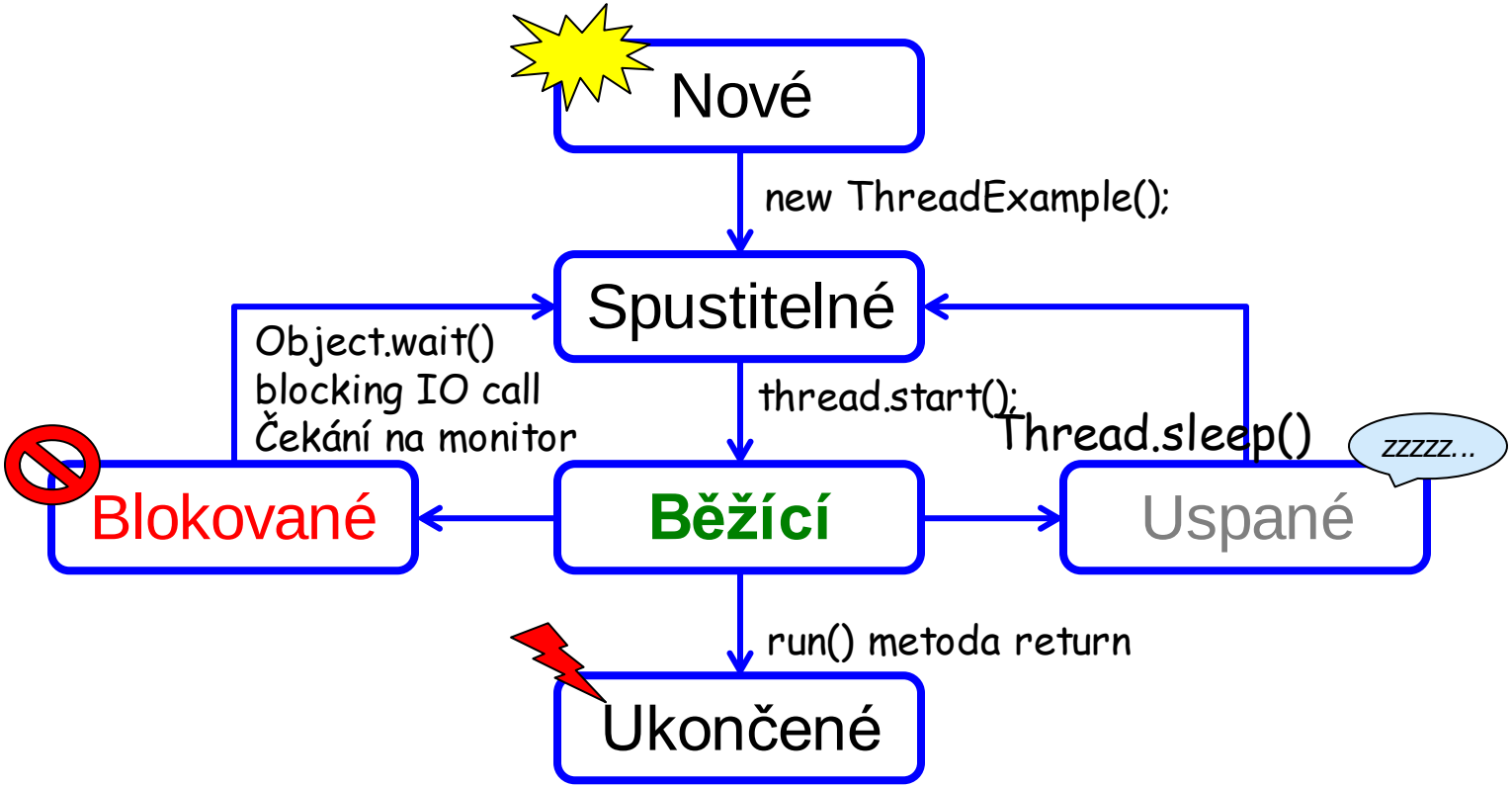


Vlákna sdílí data

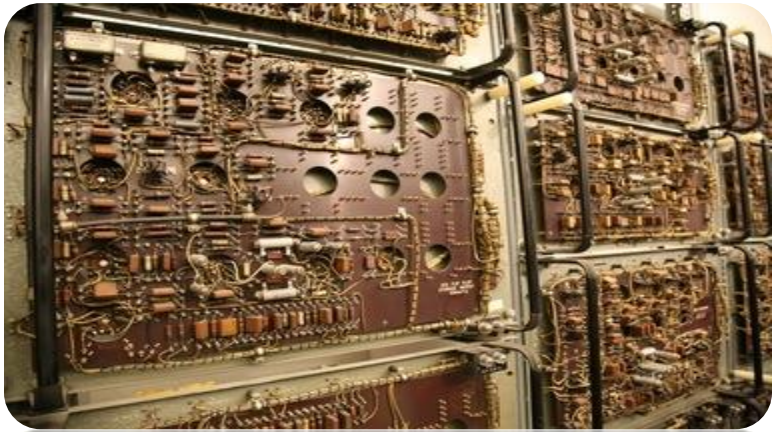
Vytváření vláken



Životní cyklus vlákna



Procesy a vlákna – JAVA



Vytvoření procesu – příklad 1

```
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;

public class MediocreExecJavac {
    public static void main(String args[]) {
        try {
            Runtime rt = Runtime.getRuntime();
            Process proc = rt.exec("javac");
            InputStream stderr = proc.getErrorStream();
            InputStreamReader isr = new InputStreamReader(stderr);
            BufferedReader br = new BufferedReader(isr);
            String line = null;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
            int exitVal = proc.waitFor();
            System.out.println("Vystupni hodnota:" + exitVal);
        } catch (Throwable t) {
            t.printStackTrace();
        }
    }
}
```

Vytvoření procesu

Čtení jeho odezvy

Vytvoření vlákna – příklad 1

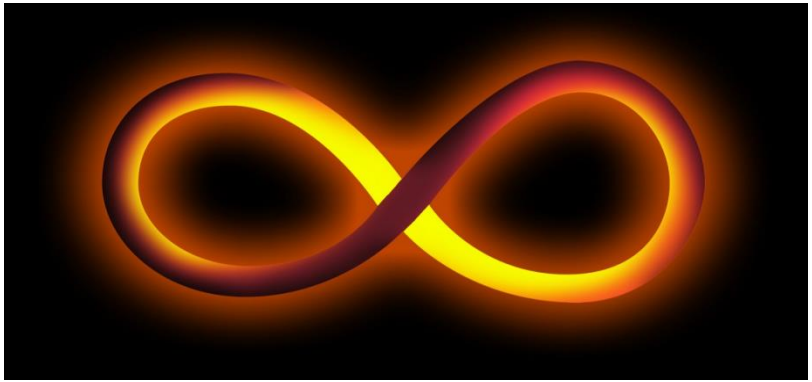
```
class PrimeThread extends Thread {
        PrimeThread() {
        }
        public void run() {
         //běží v rámci vlákna (po skončení metody končí i vlákno)
        }
    }
**********************************************************************
PrimeThread p = new PrimeThread();
 p.start();
```

Vytvoření vlákna – příklad 2

```
class PrimeRun implements Runnable {
        PrimeRun() {
        }
        public void run() {
         //běží v rámci vlákna (po skončení metody končí i vlákno)
                while(condition) {          }
        }
}**********************************************************************
PrimeRun p = new PrimeRun();
new Thread(p).start();
```

Vytvoření vlákna – metoda *run()*

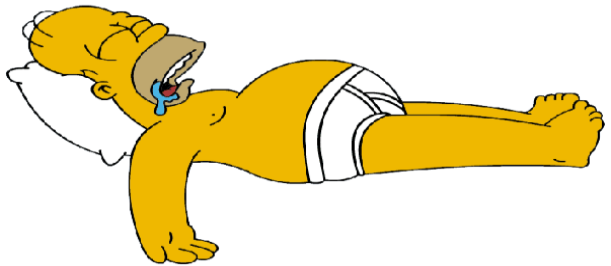
- Obvykle obsahuje nějaký druh smyčky s podmínkou, která určuje platnost vlákna.
- Smyčka je často nekonečná (vlákno je aktivní po celou dobu běhu aplikace).
- S dokončením metody *run()*, zaniká také samotné vlákno.



Ovlivnění chování vlákna – *sleep*

- Vlákno je možné také dočasně zastavit na stanovenou dobu.
- Tento krok navíc umožní plánovači úloh (thread scheduler), přepnout mezi jednotlivými vlákny.
- Vynechání tohoto kroku může na některých platformách způsobit spotřebu 100 % času CPU jedním vláknem.

```
// Starý způsob:
// Thread.sleep(100);
// Java SE5/6 - styl:
TimeUnit.MILLISECONDS.sleep(100);
```



Ovlivnění chování vlákna – *wait*

- Vlákno je možné také dočasně zastavit a čekat na dokončení jiného vlákna
- V tomto případě nečeká po stanovenou dobu
- K tomu slouží **wait()**, používá se ve spojení s **notify() / notifyAll()**
- **Umožňuje čekání na události, kde notify jej ze stavu probudí**

- Hlavní vlákno bude čekat na dokončení výpočtů, které běží v paralelních vláknech

Ovlivnění chování vlákna – *interrupt*

- Vlákno je možné také ze stavu sleep či wait předčasně probudit
- K tomu slouží příkaz **interrupt**

- Příklad: Po pěti odesláních zprávy „hello“ ukonči (ihned) vysílání vlákna, které má na starosti vysílání zpráv „update“

Ovlivnění chování vlákna – *join*

- Vlákno je možné nechat počkat na dokončení jiného vlákna
- K tomu slouží příkaz **join**

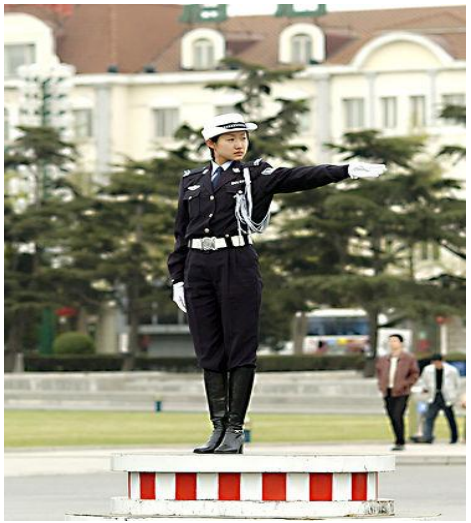
- Příklad: Po pěti odesláních zprávy „hello“ ukonči (ihned) vysílání vlákna, které má na starosti vysílání zpráv „update“

Příklad

- Navrhněte protokol, který bude každých 5 sekund posílat zprávu „hello“ a každých 7 sekund zprávu „update“

Ovlivnění chování vlákna – *yield*

- Jedná se o nepřímý pokyn (naznačení) plánovači úloh, že aktuální vlákno již v daném běhu dokončilo svoji práci a že jiné (se stejnou prioritou) může nyní mít přidělen čas CPU.
- Není zde žádná garance, že se tak skutečně stane (nedeterministický přístup).



Řízení vláken operačním systémem

- Aktuálně běžící vlákno, může být přerušeno z několika důvodů, např.:
  - sleep, yeild (viz předchozí)
  - čekání na dokončení I/O operace, příchod paketu…
  - přerušení operačním systémem ve prospěch jiného vlákna s větší **prioritou**
- Systém priorit lze částečně ovlivňovat:

  Thread.currentThread().setPriority(priority);

- Znovu se jedná o nedeterministický přístup.

Možné problémy při návrhu a implementaci více vláknových aplikací….

Race condition (souběh)

Vyžaduje odlišný pohled na věc……..

Souběh dvou vláken - příklad

Tento kód mohou vykonávat 2 vlákna


```
int value = 7; !!!!
public void increment() {
    if(value == 5){
        value = value + 1;
            //platí zde vždy, že value = 6 ??
    }
}
```

Souběh – příklad (vysvětlení)

- Nemusí platit pokud je metoda volána více než jedním vláknem a ty volají metodu *nad stejným objektem*.
- První vlákno se může dostat za podmínku rovnosti (value==5), ale ještě před samotnou inkrementací je přerušeno plánovačem úloh ve prospěch druhého vlákna.
- To vykoná v rámci přiděleného času metodu celou.
- První vlákno po opětovném přidělení CPU (již v bloku if {}) dokončí inkrementaci proměnné a nečekaný výsledek je na světě……

Souběh dvou vláken - příklad

Tento kód nemohou vykonávat 2 vlákna současně


```
int value = 5;
public synchronized void increment() {
    if(value == 5){
        value = value + 1;
            //platí zde vždy, že value = 6 ??
    }
}
```

Můžeme si být ve více vláknových aplikacích vůbec něčím jisti…..? ☺

- Nutno identifikovat objekty (datové struktury) v paměti sdílené několika vlákny.
- Zajistit vzájemně výlučný (synchronizovaný) přístup ke sdíleným prostředkům.
- Možno využít existující podpory v programovacích jazycích.

Případně vytvořit vlastní synchronizační mechanizmy……..

Synchronizace - podpora

- Synchronizované metody a bloky kódu:
  - Konstrukce *synchonized* (Java), *Lock* (C#),…
- Synchronizované datové struktury:
  - Např. *Vector* (Java), *SynchronizedCollection<T>* (C#)
- Mechanismus monitorů, případně semaforů….

Nadbytečné použití synchronizace může zpomalovat aplikace……..

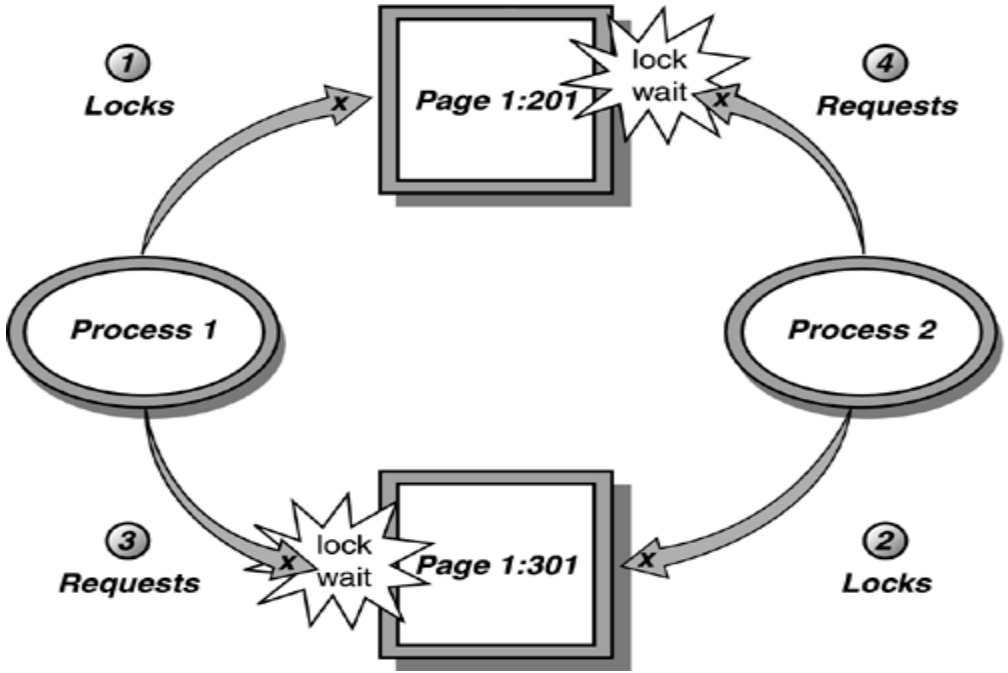
Častý příklad: problém producent-konzument

- Jedno vlákno (producent) přijímá pomocí UDP protokolu pakety a ukládá je do určité datové struktury (sdílené objekt).
- Druhé vlákno (konzument) pakety z datové struktury čte a zpracovává.
- Nutno zajistit vzájemně výlučný přístup ke sdílené datové struktuře….

Deadlock (uváznutí) procesů/vláken

Nastane v případě, když dvě vlákna/procesy na sebe čekají až se uvolní jimi uzamčené sdílené zdroje.

- **Nesprávné použití synchronizačních mechanismů.**



Třída Thread

Konstruktory

- Thread(Runnable r) vytváří run() a volá r.run()

Hlavní metody

- start() aktivuje run() a vrací se k volajícímu
- isAlive() vrací pravdu, pokud je vlákno po start a před stop
- join() čeká na ukončení (volitelně časovač)
- interrupt() přeruší wait, sleep, nebo join
- isInterrupted() vrací stav přerušení
- getPriority() vrací plánovací prioritu
- setPriority(int priorityFromONEtoTEN) nastavuje ji

Příklad: synchronizace

- Vytvořte program, kde jedno vlákno bude posílat zprávy do vyrovnávací paměti. Poté, co bude doručeno 5 paketů se spustí přehrávání ve druhém vlákně.

Výpočetní grid

A neb, co když jedno PC nestačí…

Výpočetní grid

- Jedná se o síť distribuovaných zdrojů, zahrnující počítače, přepínače, směrovače, data a další periferie..
- Zdroje mohou být vlastněny různými organizacemi (několik administrativních domén).
- Jsou řízeny specializovaným SW (jedná se o druh *middleware* vrstvy, tj. střední) jenž umožňuje sdílení a správu zdrojů.
- Tyto zdroje jsou využívány k řešení společného problému.
- Jde o jistou formu „*super virtuálního počítače*“.

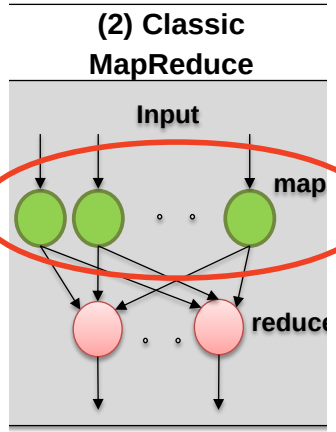
Výpočetní grid vs. superpočítač

- Výpočetní grid je složen z nezávislých počítačů (vlastní CPU, datové uložiště, napájení, síťové rozhraní…) navzájem propojených lokální počítačovou sítí či přes Internet.
- Superpočítač vyžaduje vysoce specializovaný HW propojující několik CPU rychlou datovou sběrnicí.
- Také vyžaduje specializovaný SW, využívající možnosti HW

Výpočetní gridy - rozdělení.

- Existuje mnoho typů výpočetních gridů.
  - Soukromé vs. veřejné.
  - Lokální vs. globální.
  - Univerzální vs. specializované (vědecké)
  - Výpočetní, znalostní, kolaborační
- Data ke kódu vs. Kód k datům
  - Průmyslový standard pro tyto výpočty: Hadoop

Příklad: Map-Reduce (Hadoop) 1/3

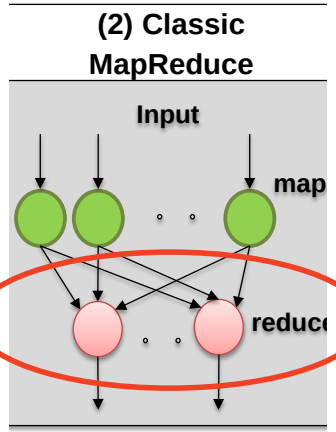


Map

```
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;


public class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{
    @Override
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException
    {
        String line = value.toString();

        //Split the line into words
        for(String word: line.split("\\W+"))
        {
                //Emit the word as you see it
                context.write(new Text(word), new IntWritable(1));
        }
    }
}
```

Rozděl výpočet po slovech (pro demo)

Příklad: Map-Reduce (Hadoop) 2/3



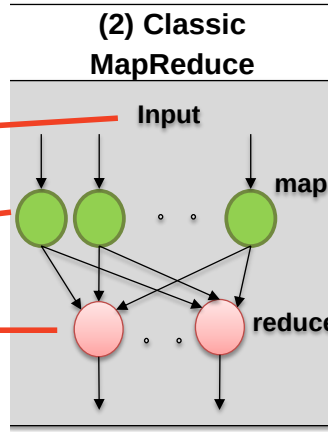
Reduce

```
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;


public class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable>{

    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException
    {
        //Initializing the word count to 0 for every key
        int count=0;

        for(IntWritable value: values)
        {
            //Adding the word count counter to count
            count += value.get();
        }

        //Finally write the word and its count
        context.write(key, new IntWritable(count));
    }
}
```

Počítá sumu všech mezivýsledků

Příklad: Map-Reduce (Hadoop) 3/3

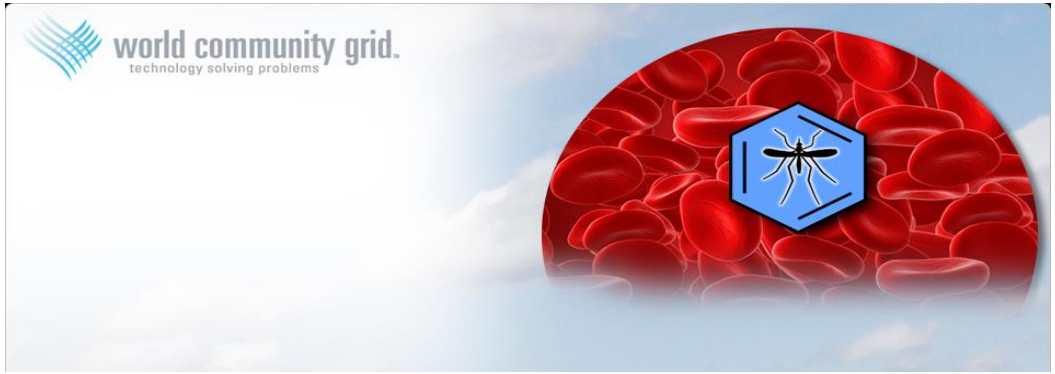
```
public static void main(String args[]) throws Exception
    {
        //Instantiate the job object for configuring your job
        Job job = new Job();

        //Specify the class that hadoop needs to look in the JAR file
        //This Jar file is then sent to all the machines in the cluster
        job.setJarByClass(WordCount.class);

        //Set a meaningful name to the job
        job.setJobName("Word Count");

        //Add the path from where the file input is to be taken
        FileInputFormat.addInputPath(job, new Path(args[0]));

        //Set the path where the output must be stored
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        //Set the Mapper and the Reducer class
        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        //Set the type of the key and value of Mapper and reducer
        /*
         * If the Mapper output type and Reducer output type are not the same then
         * also include setMapOutputKeyClass() and setMapOutputKeyValue()
         */
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        //Start the job and wait for it to finish. And exit the program based on
        //the success of the program
        System.exit(job.waitForCompletion(true)?0:1);
    }
```



Spuštění v gridu

Výpočetní gridy

Berkeley Open Infrastructure for Network Computing



Heterogenní výpočty

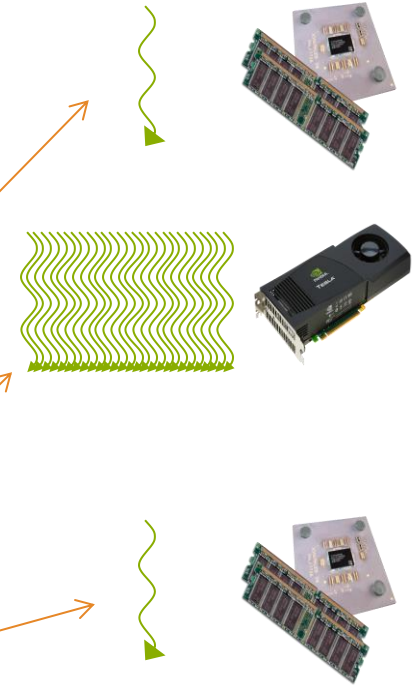
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N 1024
#define RADIUS 3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}

void fill_ints(int *x, int n) {
    fill_n(x, n, 1);
}

int main(void) {
    int *in, *out;          // host copies of a, b, c
    int *d_in, *d_out;      // device copies of a, b, c
    int size = (N + 2*RADIUS) * sizeof(int);

    // Alloc space for host copies and setup values
    in = (int *)malloc(size); fill_ints(in, N + 2*RADIUS);
    out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

    // Alloc space for device copies
    cudaMalloc((void d **)&d_in, size);
    cudaMalloc((void d **)&d_out, size);

    // Copy to device
    cudaMemcpy(d_in, in, size, cudaMemcpyHost To Devic e) ;
    cudaMemcpy(d_o ut, out, size, cudaMemcpyHost To Devic e) ;

    // Launch stencil_1d() kernel on GPU
    stencil_1d<<<N/B LOCK _SI ZE,BLOCK _SIZE >>>(d _in +RADIUS,
d_out + RADIUS);

    // Copy result back to host
    cudaMemcpy(o ut, d_out, size, cudaMemcpyDevic eToH ost) ;

    // Cleanup
    free(in); free(out);
    cudaFree(d_i n); cudaFree(d_o ut );
    return 0;
}
```

Paralelní kód fn

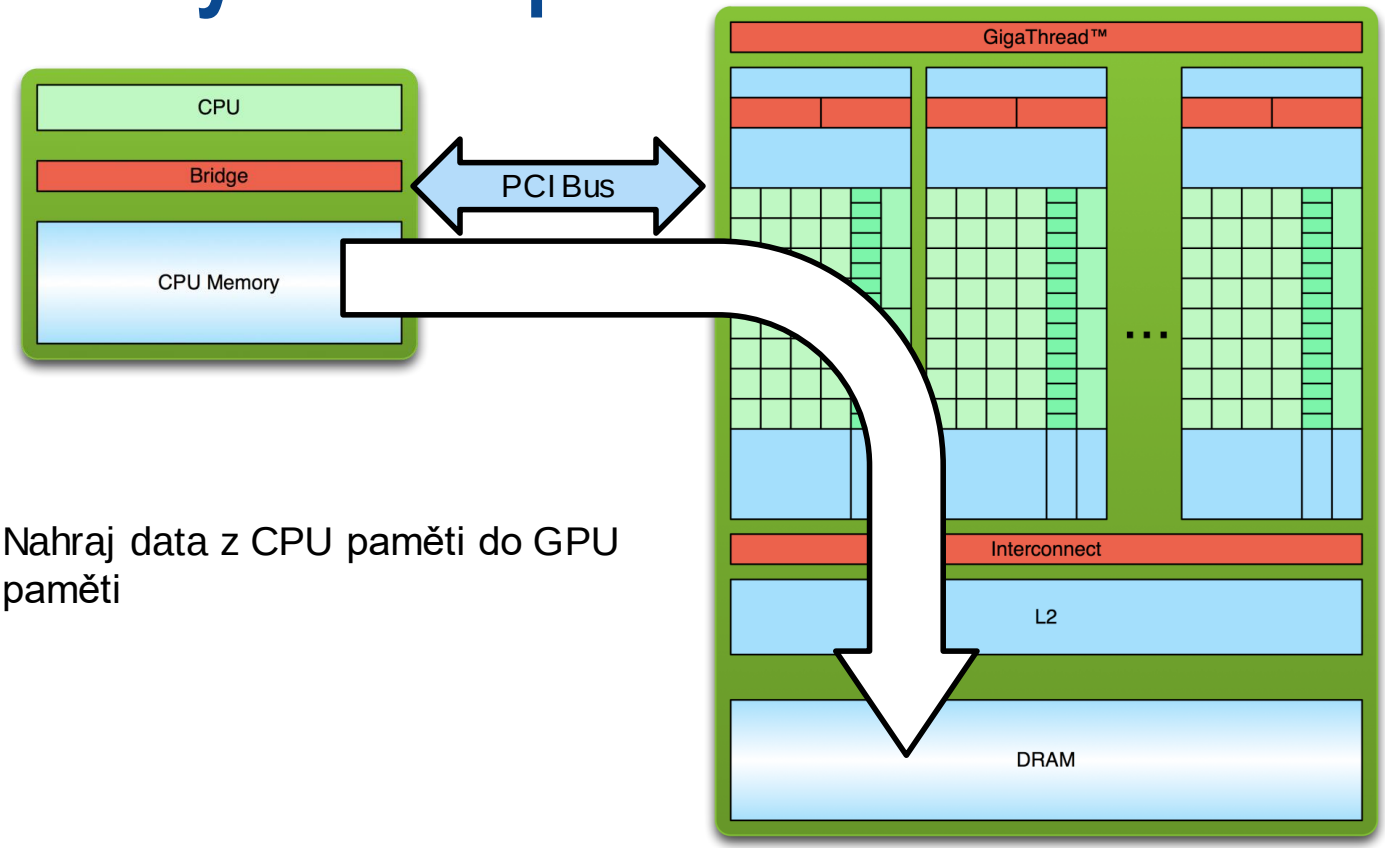
sériový kód

Paralelní kód

sériový kód

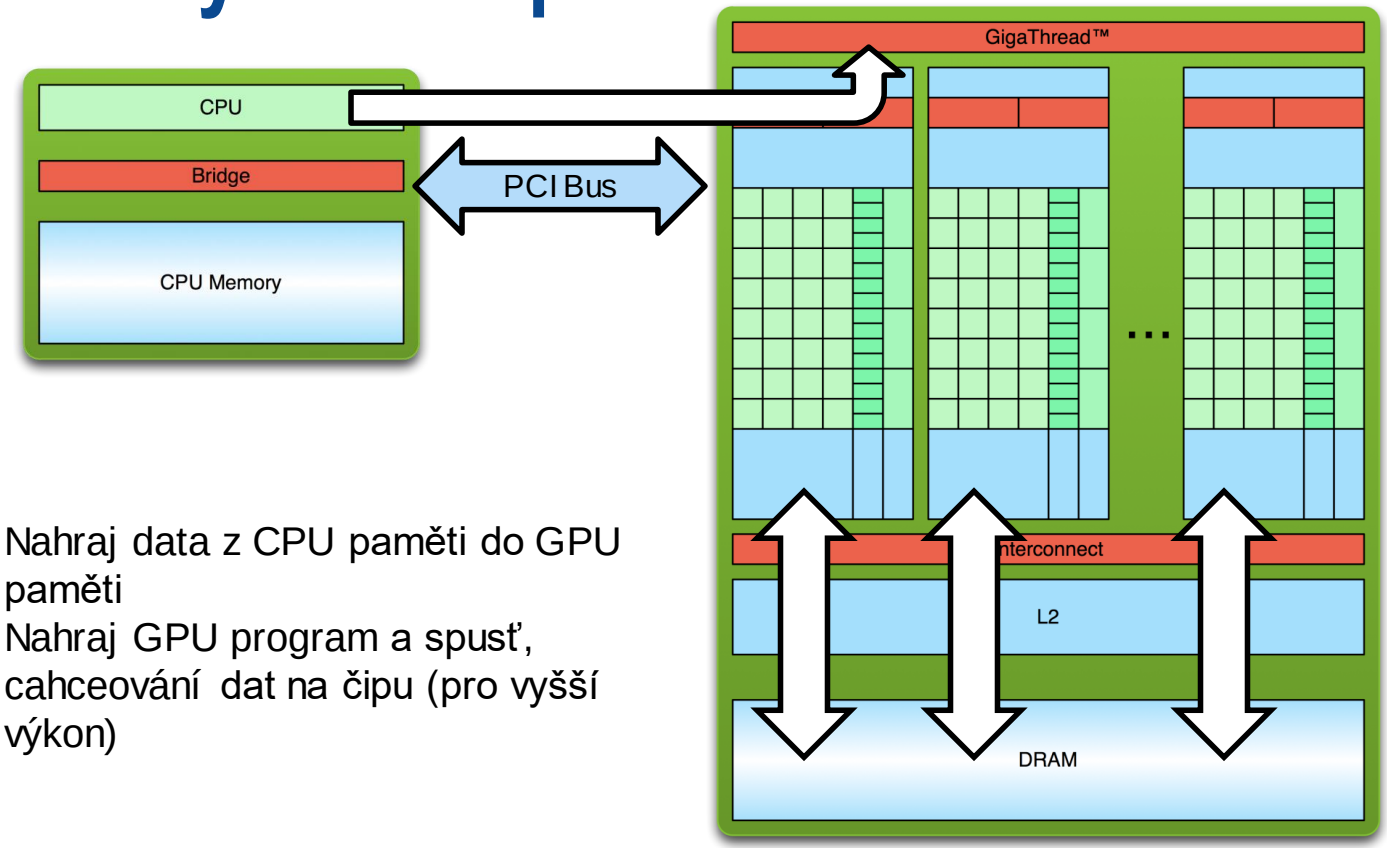


Jednoduchý model zpracování na GPU



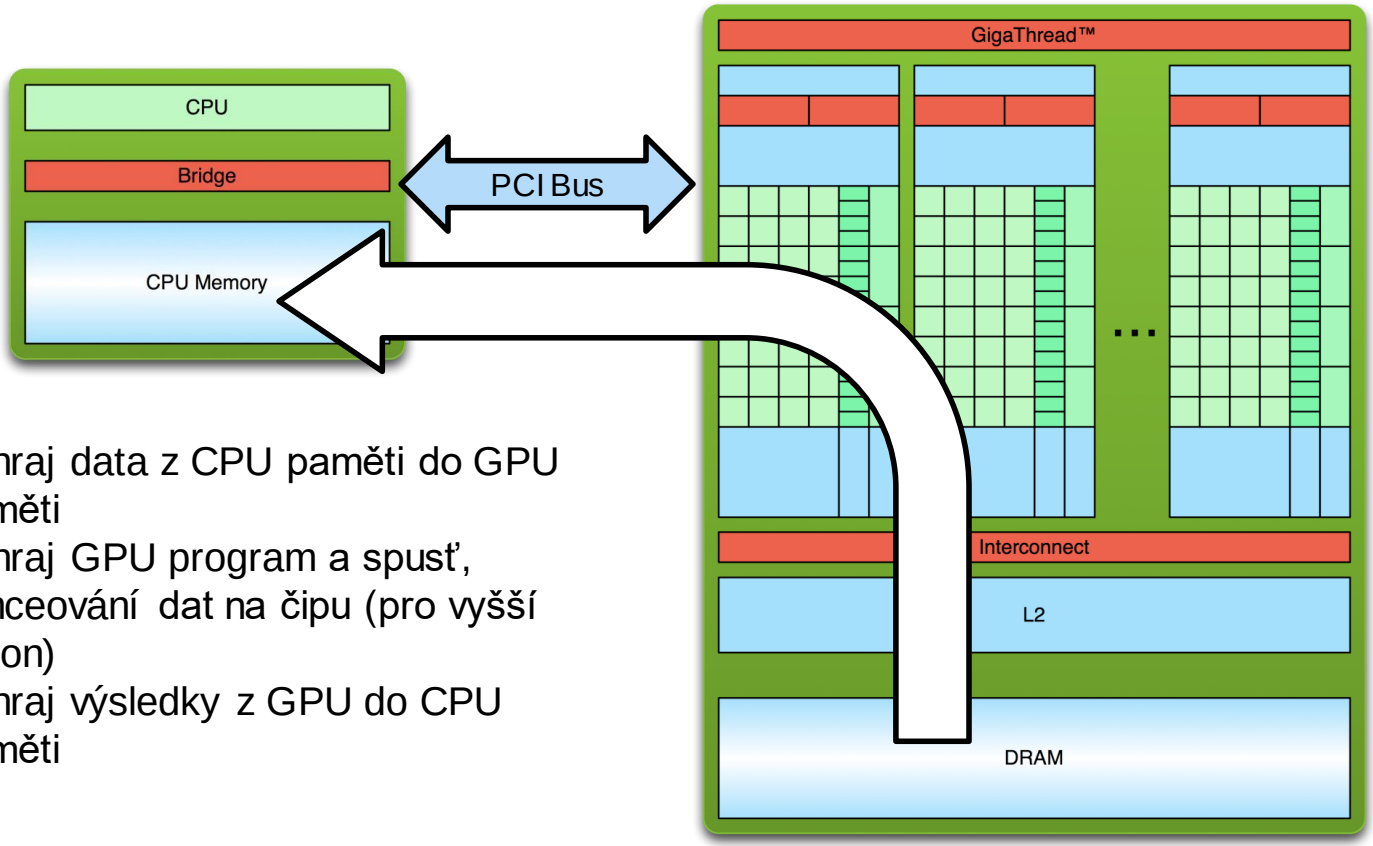
1. Nahraj data z CPU paměti do GPU paměti

Jednoduchý model zpracování na GPU



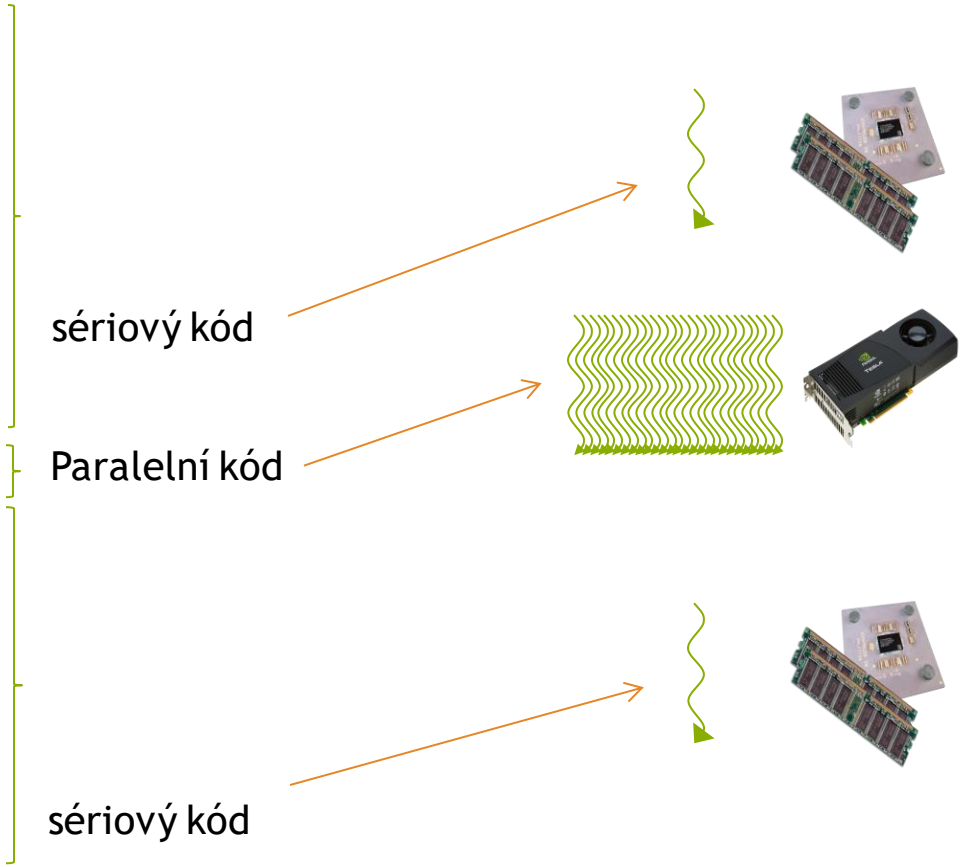
1. Nahraj data z CPU paměti do GPU paměti
2. Nahraj GPU program a spusť, cahceování dat na čipu (pro vyšší výkon)

Jednoduchý tok zpracování



1. Nahraj data z CPU paměti do GPU paměti
2. Nahraj GPU program a spusť, cahceování dat na čipu (pro vyšší výkon)
3. Nahraj výsledky z GPU do CPU paměti

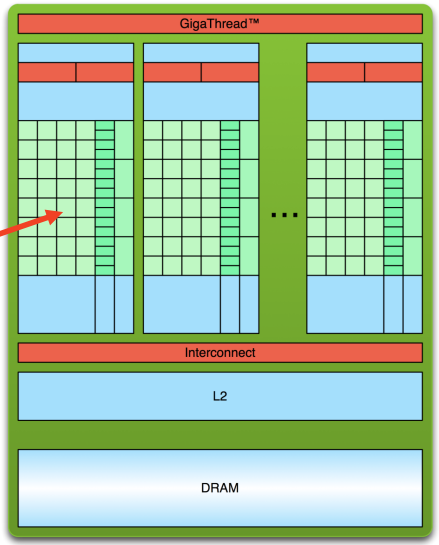
```
// declare the vectors' number of elements and their size in bytes
static const int n_el = 512;
static const size_t size = n_el * sizeof(float);

int main(){
  // declare and allocate input vectors h_A and h_B in the host (CPU) memory
  float* h_A = (float*)malloc(size);
  float* h_B = (float*)malloc(size);
  float* h_C = (float*)malloc(size);

  // declare device vectors in the device (GPU) memory
  float *d_A,*d_B,*d_C;

  // initialize input vectors
  for (int i=0; i<n_el; i++){
    h_A[i]=sin(i);
    h_B[i]=cos(i);
  }

  // allocate device vectors in the device (GPU) memory
  cudaMalloc(&d_A, size);
  cudaMalloc(&d_B, size);
  cudaMalloc(&d_C, size);

  // copy input vectors from the host (CPU) memory to the device (GPU) memory
  cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

  // call kernel function
  sum(d_A, d_B, d_C, n_el);

  // copy the output (results) vector from the device (GPU) memory to the host (CPU) memory
  cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
  // free device memory
  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);

  // compute the cumulative error
  double err=0;
  for (int i=0; i<n_el; i++) {
    double diff=double((h_A[i]+h_B[i])-h_C[i]);
    err+=diff*diff;
    // print results for manual checking.
    std::cout << "A+B: " << h_A[i]+h_B[i] << "\n";
    std::cout << "C: " << h_C[i] << "\n";
  }
  err=sqrt(err);
  std::cout << "err: " << err << "\n";

  // free host memory
  delete[] h_A;
  delete[] h_B;
  delete[] h_C;

  return cudaDeviceSynchronize();
}
```

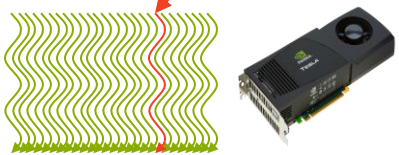


Příklad: CUDA – sečtení vektorů

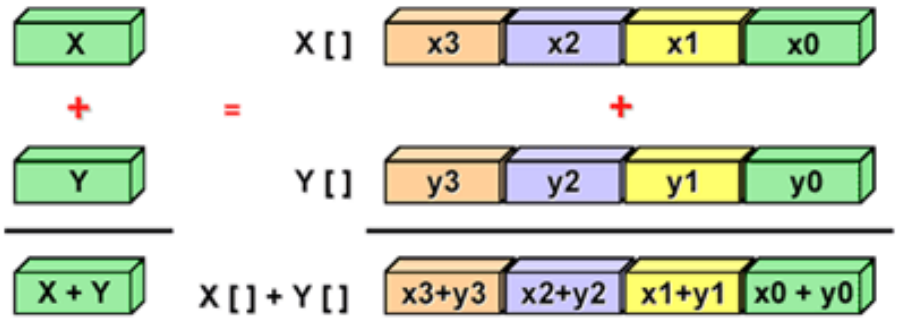
```
extern "C"
__global__ void sum(int n, float *a, float *b, float *sum)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i<n)
    {
        sum[i] = a[i] + b[i];
    }
}
```

ID procesoru

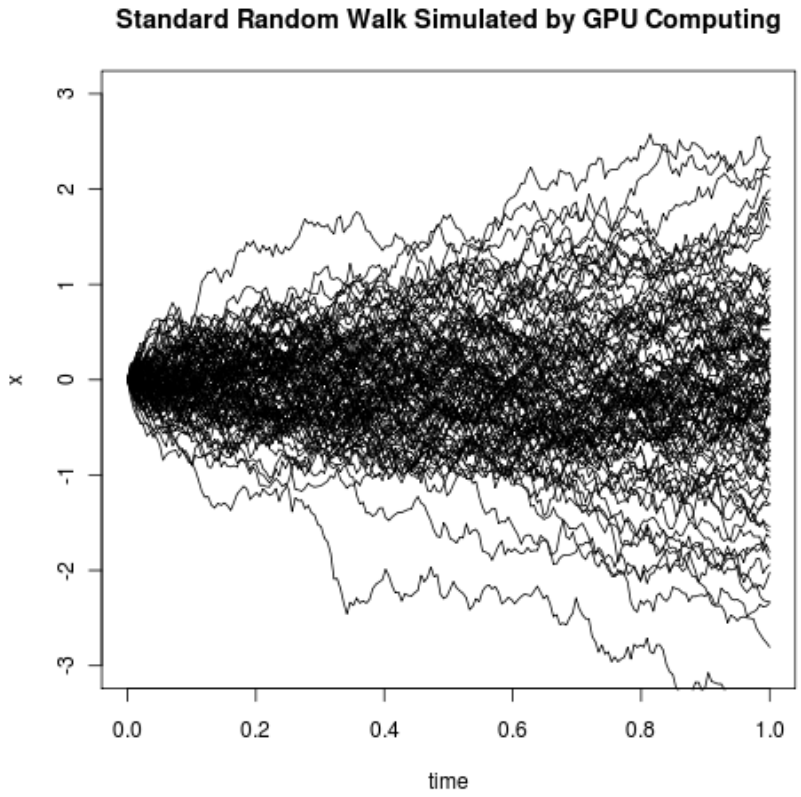


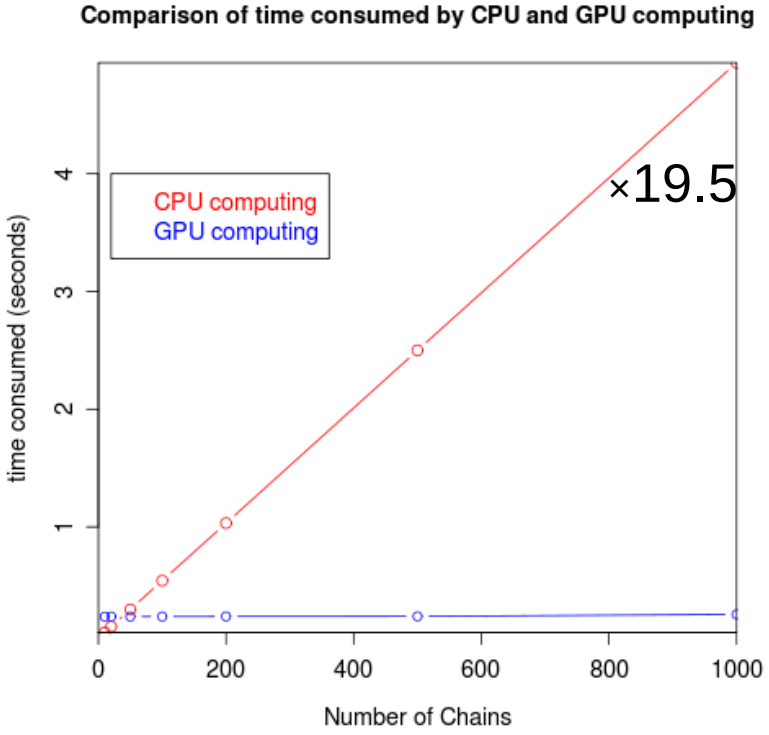


Každé tid pracuje s jinou pamětí

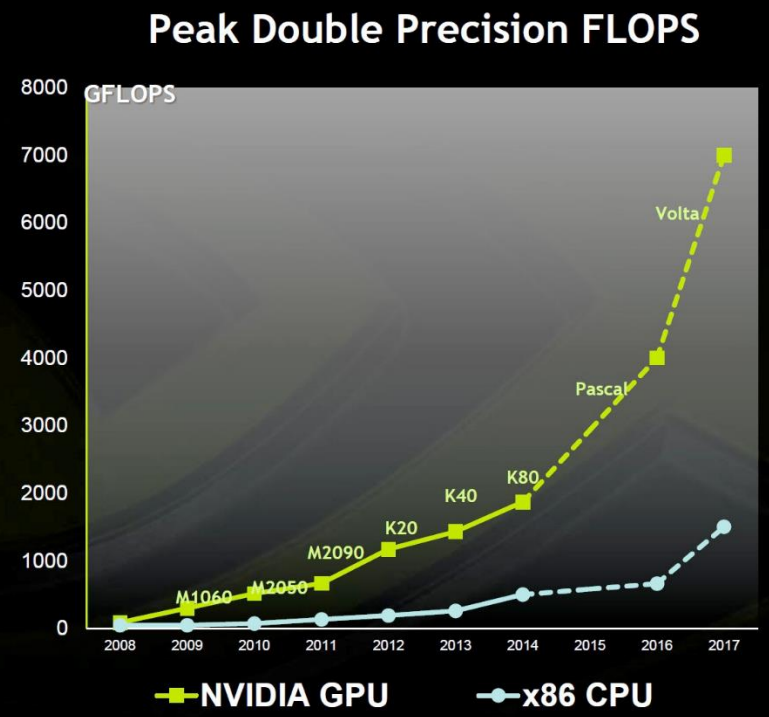


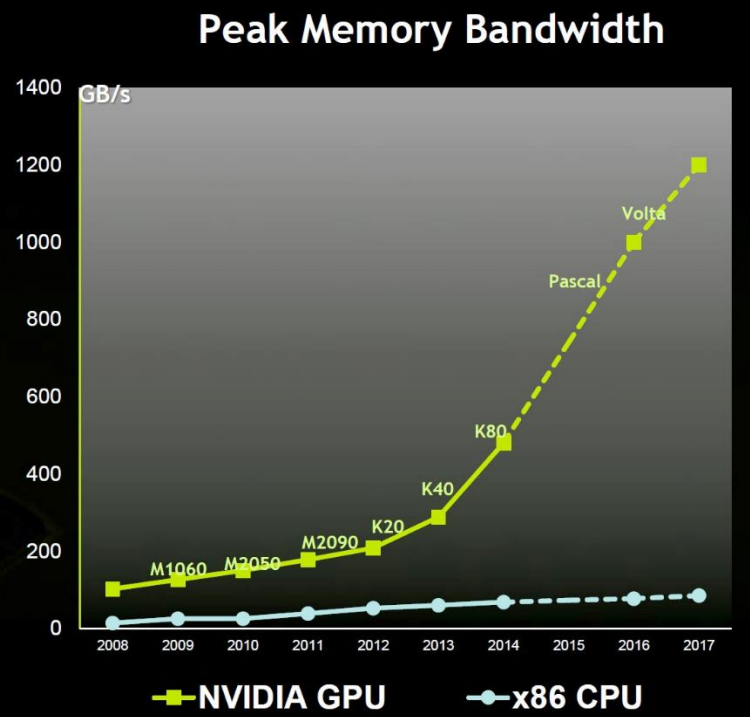
Random walk simulation by GPU





GPU Motivation (I): Performance Trends





7

Děkuji za pozornost