BIG DATA 2

Technológie spracovania veľkých dát

Peter Bednár, Martin Sarnovský

Paralelné výpočty

- Architektúra počítačov
- Hardvérová paralelizácia
  - Pipelining, Superskalárna architektúra, SIMD inštrukcie
- Softvérová paralelizácia
  - Viacprocerosorové a viacjadrové systémy
  - GPU
  - Dekompozícia úloh
  - Synchronizácia výpočtov
  - Funkcionálne programovanie

Architektúra počítačov (1)

- Architektúra bežných počítačov je zložená z:
  - Procesora
  - Operačnej pamäte
  - Vstupno/výstupnej dátovej zbernice na ktorej sú pripojené periférne zariadenia (napr. pevný disk, sieťová karta)
- von Neumannova architektúra
  - V operačnej pamäti sú uložené dáta aj inštrukcie programu – náhodný prístup, čítanie/zápis
  - Procesor je zložený z:
    - Kontrolná jednotka načítava a dekóduje inštrukcie programu
    - Aritmeticko-logická jednotka vykonáva nad dátami základné aritmetické operácie

Architektúra počítačov (2)

- Inštrukcie najčastejšie priamo pracujú s dátovými registrami
  - Šírka registrov (počet bitov) a ich počet definuje koľko dát môže inštrukcia naraz spracovať
- Spracovanie inštrukcie je rozdelené do viacerých krokov – fáz, (napr. načítanie inštrukcie, dekódovanie inštrukcie, načítanie dát, ...)
- Rýchlosť spracovania závisí na taktovacej frekvencii
  - Závislá na úrovni integrácie a výrobnej technológii
  - Fyzikálne obmedzená – zvyšuje sa spotreba a pracovná teplota
  - V súčasnosti max. 4 GHz, bežná prevádzková frekvencia 2.5-3 GHz

Paralelizácia výpočtov

- Vysoká miera integrácie umožňuje umiestniť do jedného čipu viac tranzistorov – zložitejšie riadiace jednotky, viac aritmeticko-logických jednotiek
  - 10-7 nm, prechod na 5 nm
- Vertikálne zvyšovanie výkonu procesorov – paralelizácia výpočtov
  - Vykonať viac inštrukcií paralelne v rovnakom čase
- Metódy paralelyzácie výpočtov môžu byť implementované:
  - Implicitne v hardvéry (transparentné pre programátora)
  - V softvéry – programátor musí upraviť svoj kód aby mohol bežať paralelne

Flynnova klasifikácia (1)

- Je viacero spôsobov ako rozdeliť a definovať paralelné systémy
- Jedna z metód je Flynnova klasifikácia
  - Rozdeľuje systémy podľa toho ako sa spracujú v systéme inštrukcie (*instruction flow*) a ako sa pristupuje k dátam (*data flow*)
  - Každé z kritérií môžu byť implementované paralelne (*multiple*) alebo jednoducho (*single*)

Flynnova klasifikácia (2)

- SISD – Single Instruction Single Data
  - Naraz sa vykonáva iba jedna inštrukcia, ktorá spracováva iba jedny dáta
- SIMD – Single Instruction Multiple Data
  - Naraz sa vykonáva iba jedna inštrukcia, ktorá však naraz môže spracovať viacero dátových hodnôt
- MISD – Multiple Instruction Single Data
  - Naraz sa vykonáva viacero inštrukcií nad tými istými dátami
  - Bežne sa nevyužíva - napr. viacero filtrov, ktoré sa paralelne aplikujú na tie isté dáta pri spracovaní signálov alebo viacero dešifrovacích algoritmov lúštiacich tú istú správu

Flynnova klasifikácia (3)

- MIMD – Multiple Instruction Multiple Data
  - Naraz sa vykonáva viacero inštrukcií nad rôznymi dátami
  - V súčasnosti najbežnejšia architektúra
  - Procesory môžu mať rozličnú inštrukčnú sadu – napr. hlavný procesor (CPU) a grafický procesor (GPU)
  - Ak systém obsahuje viacero rovnakých procesorov s rovnakou úplnou inštrukčnou sadou – symetrický multiprocesing

Hardvérová paralelizácia (1)

- Pipelining
  - Vykonávanie fáz viacerých inštrukcií sa „prekrýva“., tzn. jedna inštrukcia môže byť napr. vykonávaná, ďalšia sa dekóduje a ďalšia sa číta z pamäte v tom istom cykle
- Superskalárna architektúra
  - Viacero pipeline jednotiek, ktoré pracujú paralelne
  - Napr. súčasné Intel procesory majú 4-cestnú superskalárnu architektúru
- Rýchlosť spracovania je obmedzená najpomalšou fázou vykonávania inštrukcie – moderné procesory rozdeľujú spracovanie inštrukcií do veľkého počtu fáz (do 20)

Hardvérová paralelizácia (2)

- V bežnom programu je každá 5-6 inštrukcia podmienka, ktorá preskočí na ďalšiu inštrukciu mimo poradia
  - Špekulatívne predpovedanie pri vetveniach a cykloch, ktorá inštrukcia bude v programe nasledujúca
- Dátové závislosti medzi inštrukciami
  - Výstup jednej operácie je vstupom do druhej operácie
  - Pre-usporiadanie inštrukcií tak aby sa znížili priame dátové závislosti medzi inštrukciami

Hardvérová paralelizácia – príklad (1)

load R, [A] – načíta dáta z pamäte na adrese A do registra R

add R, [A] – spočíta hodnotu v registri R s hodnotou v pamäti na adrese A a výsledok uloží do R

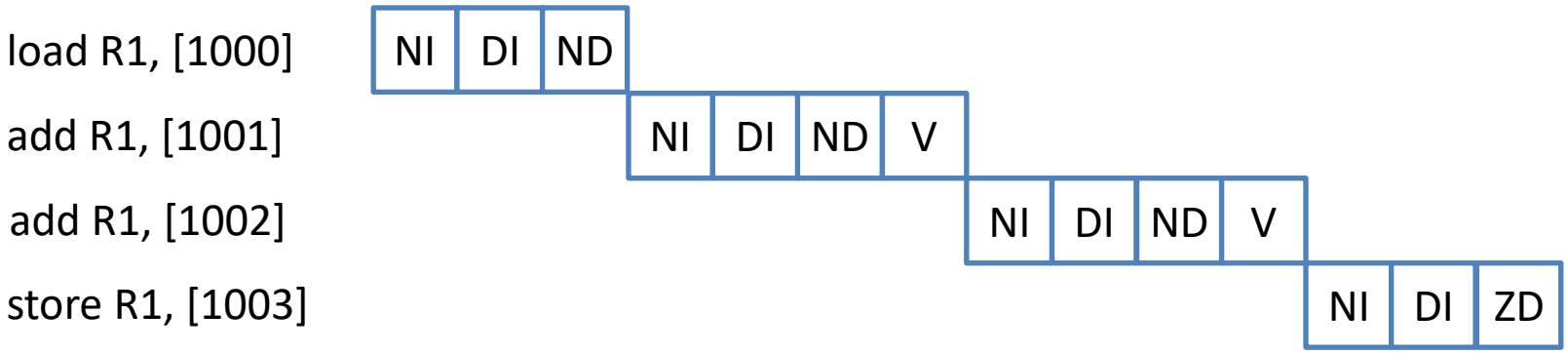
add R1, R2 – spočíta hodnoty v registroch R1 a R2 a výsledok uloží do R1

store R, [A] – uloží dáta z registra R do pamäte na adresu A

NI – načítanie inštrukcie
DI – dekódovanie inštrukcie
ND – načítanie dát
V – vykonávanie operácie
ZD – spätný zápis dát

Hardvérová paralelizácia – príklad (2)

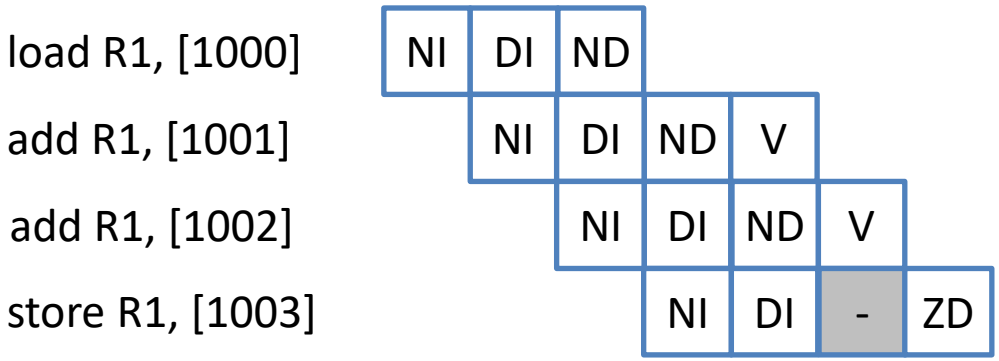
SISD – trvanie výpočtu 14 taktov



NI – načítanie inštrukcie
DI – dekódovanie inštrukcie
ND – načítanie dát
V – vykonávanie operácie
ZD – spätný zápis dát

Hardvérová paralelizácia – príklad (3)

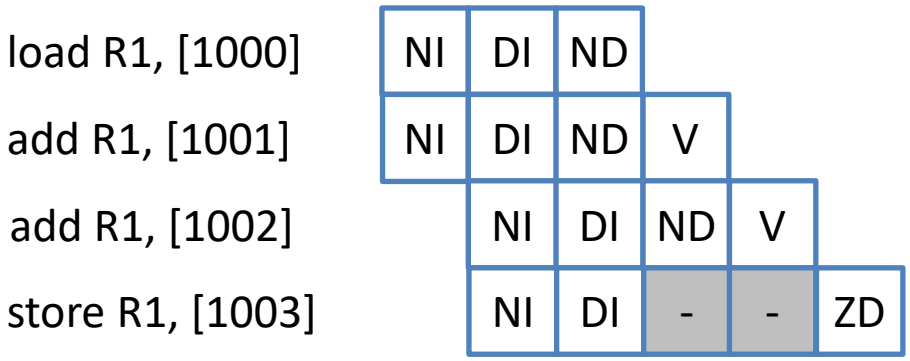
MIMD – pipelining – trvanie výpočtu 7 taktov



NI – načítanie inštrukcie
DI – dekódovanie inštrukcie
ND – načítanie dát
V – vykonávanie operácie
ZD – spätný zápis dát

Hardvérová paralelizácia – príklad (4)

MIMD – pipelining + 2 cestná superskalárna architektúra – trvanie výpočtu 6 taktov



NI – načítanie inštrukcie
DI – dekódovanie inštrukcie
ND – načítanie dát
V – vykonávanie operácie
ZD – spätný zápis dát

Hardvérová paralelizácia (3)

- SIMD inštrukcie
- Jedna inštrukcia naraz spracuje viacero dát
- Príklad: spočítame dva 4-prvkové vektory čísel

```
load R1, [1000]
add R1, [2000]
store R1, [3000]
load R1, [1001]
add R1, [2001]
store R1, [3001]
load R1, [1002]
add R1, [2002]
store R1, [3002]
load R1, [1003]
add R1, [2003]
store R1, [3003]
```

```
mload M1, [1000-1003]
madd M1, [2000-2003]
mstore M1, [3000-3003]
```

Veľkosť registra M1 = 4x veľkosť R1

Prístup k pamäti (1)

- Prístup k operačnej pamäti je charakterizovaný:
  - Latenciou - čas od inicializácie čítania/zápisu až pokiaľ sú dáta dostupné pre procesor
  - Šírkou zbernice – koľko bitov sa naraz prenesie z pamäti do procesora pri jednom prístupe
- Súčasné technológie DRAM – 80-100 ns náhodný prístup
  - Pre porovnanie fotón potrebuje 1 ns aby preletel 30 cm
  - Prístup k registrom vyžaduje približne 0.5 ns (1-2 cykly procesora)

Prístup k pamäti – príklad

- Ak máme procesor s frekvenciou 1GHz a 4-cesnou superskalárnou architektúrou - v každom 1ns takte môžu byť ukončené 4 inštrukcie, tzn. teoretický výpočtový výkon je 4G operácií za sekundu - 4GFLOPS
- Ak máme pamäť so šírkou zbernice 1 slovo a latenciou 100 ns, procesor musí čakať 100 taktov pri každom prístupe k pamäti – tzn. môže vykonať iba 1 inštrukciu za 100 ns a jeho výkon klesne na 100 MFLOPS

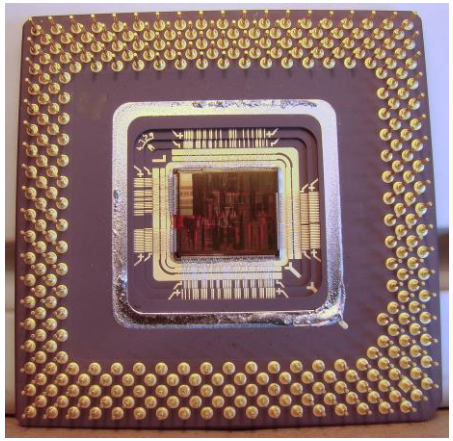
- Pre väčšinu výpočtov je najviac limitujúci prístup k pamäti než výpočtový výkon

Prístup k pamäti (2)

- Vyrovnávacia pamäť procesora – cache
  - Dočasná veľmi rýchla pamäť procesora
  - Prístupová doba 5-7ns
  - Obmedzená veľkosť (napr. do 64 kB)
  - Samostatná vyrovnávacia pamäť pre inštrukcie a pre dáta
  - Môže byť viacúrovňová – napr. L1 cache, L2 cache – väčšia kapacita

Historický príklad – Intel Pentium P55C

- Rok vydania 1996
- 0.35 µm technológia, 4.5 mil. tranzistorov
- Taktovacia frekvencia 233 MHz
- Superskalárna 2-cestná architektúra, 5 fáz
- SIMD inštrukcie (MMX – MultiMedia eXtensions)
- 32 KB cache

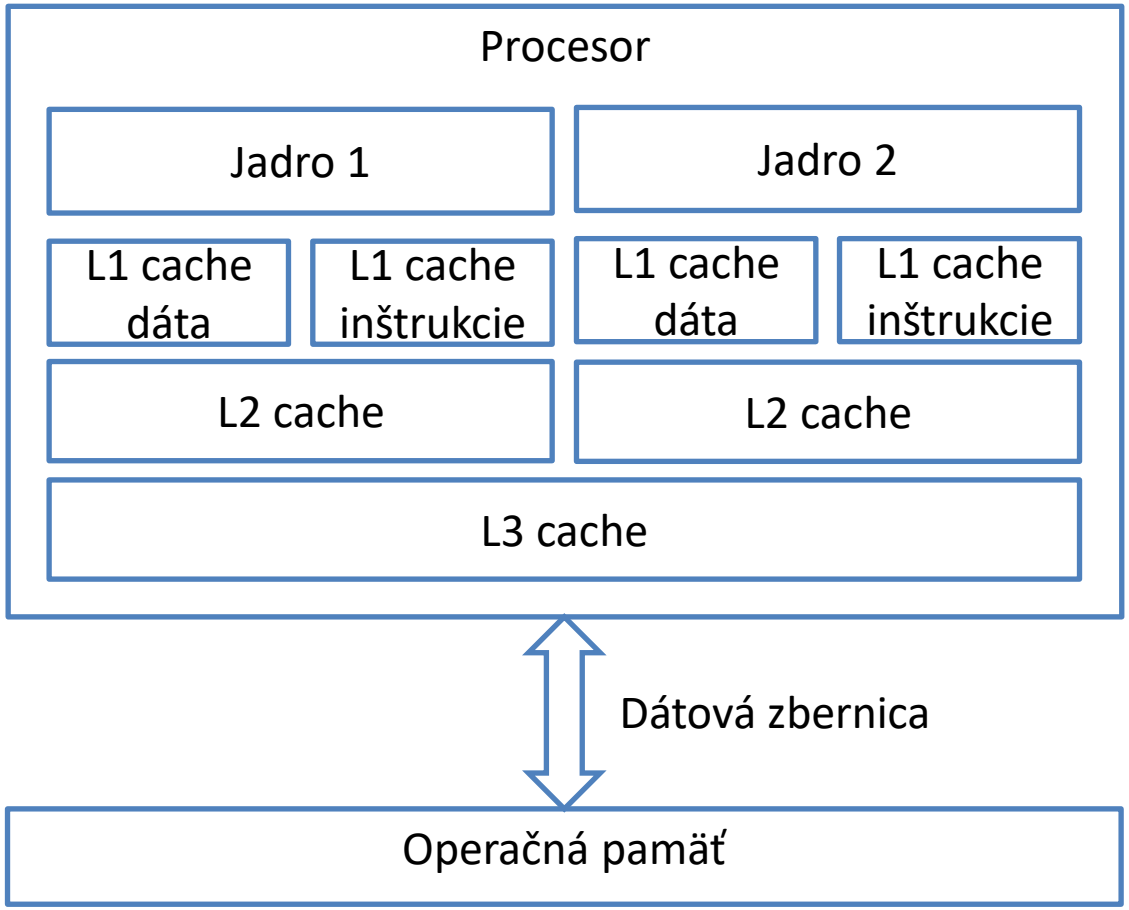


Viacprocesorové architektúry

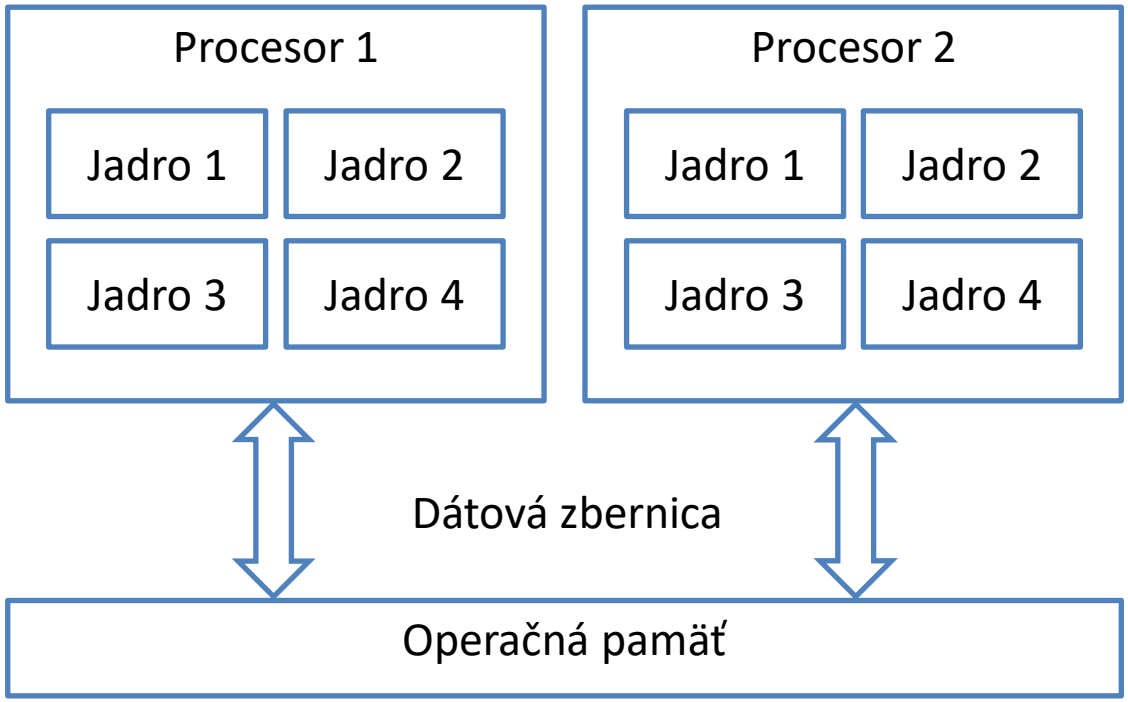
- MIMD architektúra
  - Moderné procesory majú viacero procesorových jadier – nezávislých procesorov na jednom čipe, ktoré plne umožňujú vykonávať nezávislé programy
  - V jednom počítači môže byť viacero procesorov

- Procesory/jadrá zdieľajú prístup do pamäti, podľa prístupu sa systémy rozdeľujú na:
  - UMA – Uniform Memory Access – každý procesor má rovnaký prístup ku zdieľanej pamäti
  - NUMA – Non-Uniform Memory Access – časť pamäti je lokálna iba pre jeden procesor, prístup do priestoru iného procesora vyžaduje ďalší hardvér a dlhšiu dobu

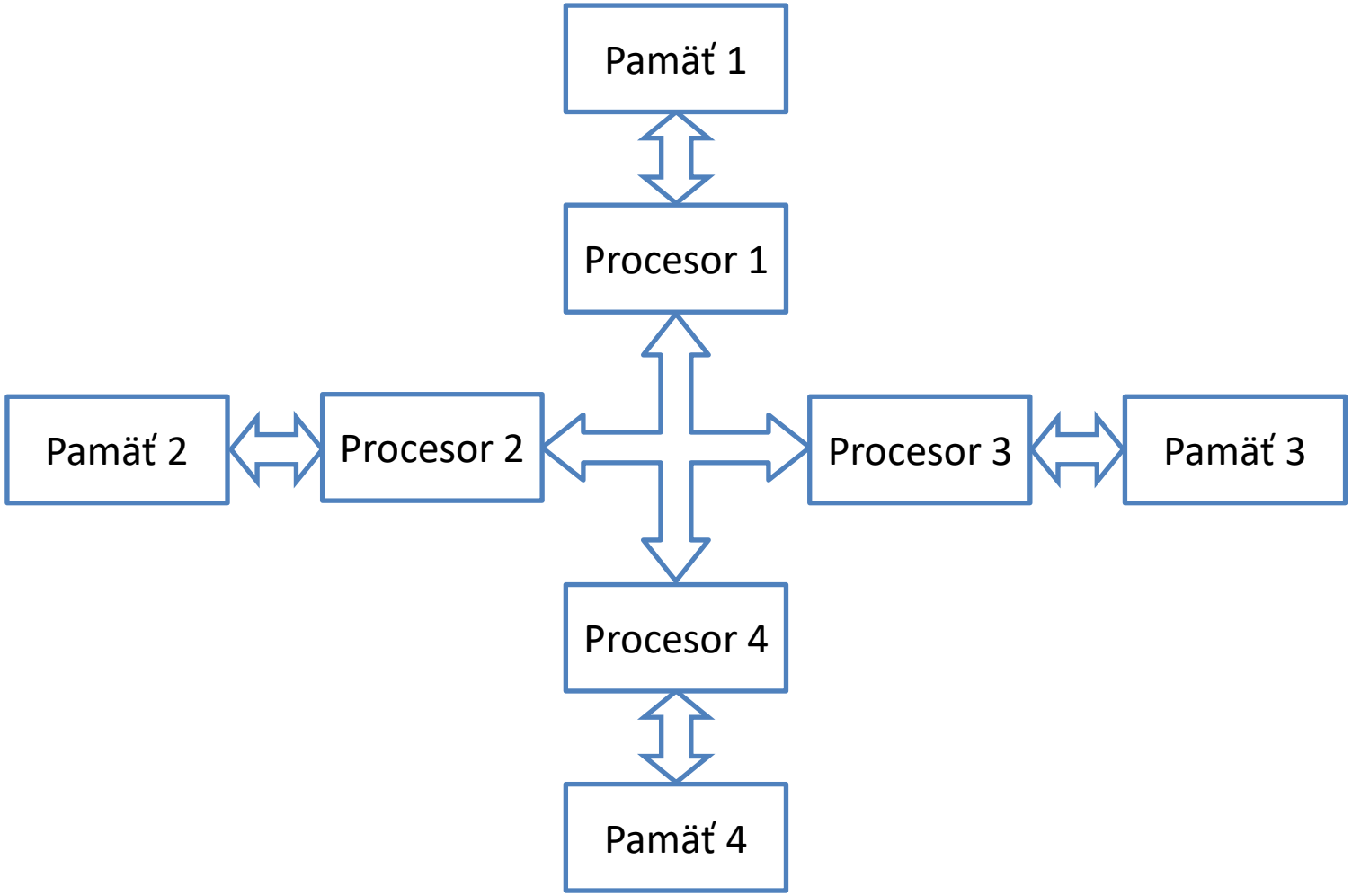
MIMD/SIMD + UMA - viacjadrový procesor



Viacprocesorová UMA architektúra



Viacprocesorová NUMA architektúra



Využitia výkonu grafických kariet (1)

- Procesory grafických kariet sú špeciálne navrhnuté na rýchle spracovanie vektorov s reálnymi číslami
  - Sada SIMD inštrukcií (obmedzená oproti hlavnému procesoru)
- Masívna paralelizácia
  - Bežný procesor grafickej karty má rádovo 100-vky až 1000-cky jadier
- NVIDIA Tesla K 40 - 2880 jadier, 12 GB grafickej pamäte, výkon 1.43 TFLOPS – $ 3000

Využitia výkonu grafických kariet (2)

- Výpočet musí byť preložený do kódu špecifického pre daný grafický procesor
  - Štandardizované programátorské rozhrania
    - CUDA – grafické čipy od firmy Nvidia
    - OpenCL – zjednotené rozhranie pre hlavné a grafické procesory rôznych výrobcov
- Grafický procesor má zvyčajne oddelenú pamäť – úzkym miestom je grafická zbernica cez ktorú musia byť dáta prenesené z operačnej pamäti do pamäti grafickej karty

Superpočítače

- Frontier - v súčasnosti najvýkonnejší počítač na svete
- 1,685.65 PFLOPS operácií
- 9,248 x 64 jadrových CPU, 36,992 x 220 jadrový GPU + každý uzol má 5 TB lokálnej pamäte
- Celkovo 606,208 CPU jadier, 8,335,360 GPU jadier, 700 PB (1PB = 112589990684262 4 B) úložisko
- 21 MW spotreba
- $ 600 mil.

Softvérová paralelizácia (1)

- Aby bolo možné plne využiť výkon viacprocesorových/viacjadrových systémov, programátor musí rozdeliť program na viacero podprogramov, ktoré sú spustené paralelne vo viacerých vláknach
- Vlákna spravuje operačný systém, ktorý prideľuje ich vykonávanie jednotlivým procesorom/jadrám
  - Operačný systém poskytuje programátorské rozhranie na spúšťanie a správu vlákien
  - Programátor musí zabezpečiť synchronizáciu prístupu k pamäti

Softvérová paralelizácia (2)

- Na jednom procesorovom jadre je možné spúšťať viacero vlákien – tzv. primtívny multitasking
  - Ak jedno vlákno čaká na periférne zariadenia (napr. na disk), operačný systém môže prepnúť vykonávanie na ďalšie vlákno a lepšie využiť procesor
  - Prepínanie vlákien však môže narušiť prístup k pamäti – nové vlákno môže vyžadovať dáta, ktoré nie sú vo vyrovnávacej pamäti
- Celkovo je teda možné mať viac vlákien než je počet procesorov/jadier
  - Výhody sa však prejavia iba ak sú vlákna závislé na prístupe k externým zariadeniam ako napr. k disku, alebo sieťovému zariadeniu

Rozdelenie výpočtu na pod-úlohy (1)

- Dátová (doménová) dekompozícia
  - Napr. máme vypočítať skalárny súčin 2 vektoroch X a Y o veľkosti 1 000 000 prvkov a máme k dispozícii 4 procesory

1. Rozdelíme vektory na 4 rovnaké časti – podvektory X_1 X_2 X_3 X_4 a Y_1 Y_2 Y_3 Y_4
2. Paralelne vypočítame skalárny súčin pre každú dvojicu podvektorov, napr. procesor 1 vypočíta súčin $X_1.Y_1$, procesor 2 súčin X_2 .Y_2 atď.
3. Spočítame výsledný súčin ako sumu súčinov podvektorov

Rozdelenie výpočtu na pod-úlohy (2)

- Funkcionálna dekompozícia
  - Napr. máme spracovať prúd dát postupnosťou dátových filtrov f_1, f_2 , f_3 , f_4 na štyroch procesoroch (výstup jedného je vstupom nasledujúceho)
  - Vstupný prúd sa rozdelí na časovú postupnosť dát T_1, T_2, T_3, ...
  - Procesor 1 vykonáva výpočet f_1, procesor 2 výpočet f_2, atď.

1. P 1 - T_1
2. P 1 - T_2 a paralelne P 2 - T_1
3. P 1 - T_3 a paralelne P 2 - T_2 a paralelne P 3 - T_1
4. atď.

Rozdelenie výpočtu a prístup k pamäti

- Prístup k súvislej oblasti pamäti je oveľa efektívnejší
  - Napr. pre 64 bitovú šírku zbernice vieme naraz čítať/zapisovať dve 32 bitové čísla v jednoduchej presnosti
  - Podobne do cache sa presúvajú z operačnej pamäti väčšie súvislé oblasti po sebe idúcich adries
- Pri dekompozícii sa snažíme usporiadať dáta v pamäti v poradí v akom k nim bude pristupovať procesor počas výpočtov

Prístup k pamäti – príklad (1)

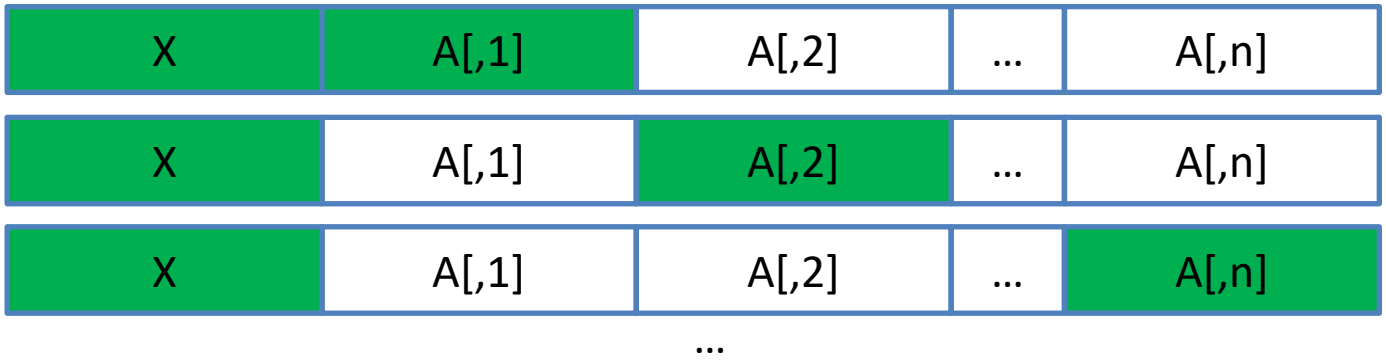
- Násobenie vektora a matice X . A
- Ak je matica uložená v pamäti po riadkoch

- Pri násobení musíme násobiť a spočítavať X[1].A[1,1] + X[2].A[2,1] + X[3].A[3,1] + ... X[n].A[n,1] v cykle pre n stĺpcov

Prístup k pamäti – príklad (2)

- Násobenie vektora a matice X . A
- Ak je matica uložená v pamäti po stĺpcoch

- Pri násobení musíme násobiť a spočítavať X[1].A[1,1] + X[2].A[2,1] + X[3].A[3,1] + ... X[n].A[n,1] v cykle pre n stĺpcov



Synchronizácia výpočtov (1)

- Ak viacero paralelných výpočtov (programových vlákien) zdieľa tie isté premenné v pamäti, programátor musí zabezpečiť synchronizáciu
- Programovacie techniky pre synchronizáciu:
  - Bariéra
  - Zámok
  - Semafor

Bariéra

- Ak chceme aby výpočet A pokračoval až keď skončí výpočet B
- Z príkladu pre výpočet skalárneho súčinu musí hlavný program počkať kým pod-úlohy vypočítajú čiastočný skalárny súčin podvektorov
- Premenná do ktorej môže zapisovať iba jedno vlákno a ostatné ju môžu iba čítať
- Ak máme dve vlákna A a B a premennú b ktorá slúži ako bariéra
- Na začiatku je premenná b = 0
- Po skončení výpočtu B nastaví b = 1
- A v cykle číta premennú b a čaká až kým sa b = 1

Zámok (1)

- Príklad: chceme zvýšiť premennú, ktorá uchováva stav konta:

  int konto = 0; ... konto += 1000;
- Na úrovni inštrukcií procesora to nie je atomická operácia!
- Ak máme dva paralelné procesy, ktoré chcú aktualizovať stav konta:

čas:	vlákno 1	vlákno 2
1	load r0, @konto	-
2	add r0, 800	load r0,@konto
3	store r0, @konto	add r0, 600
4	-	store r0, @konto

- Postupnosť príkazov musí byť uzavretá do kritickej sekcie ktorú v danom čase môže vykonávať iba jedno vlákno

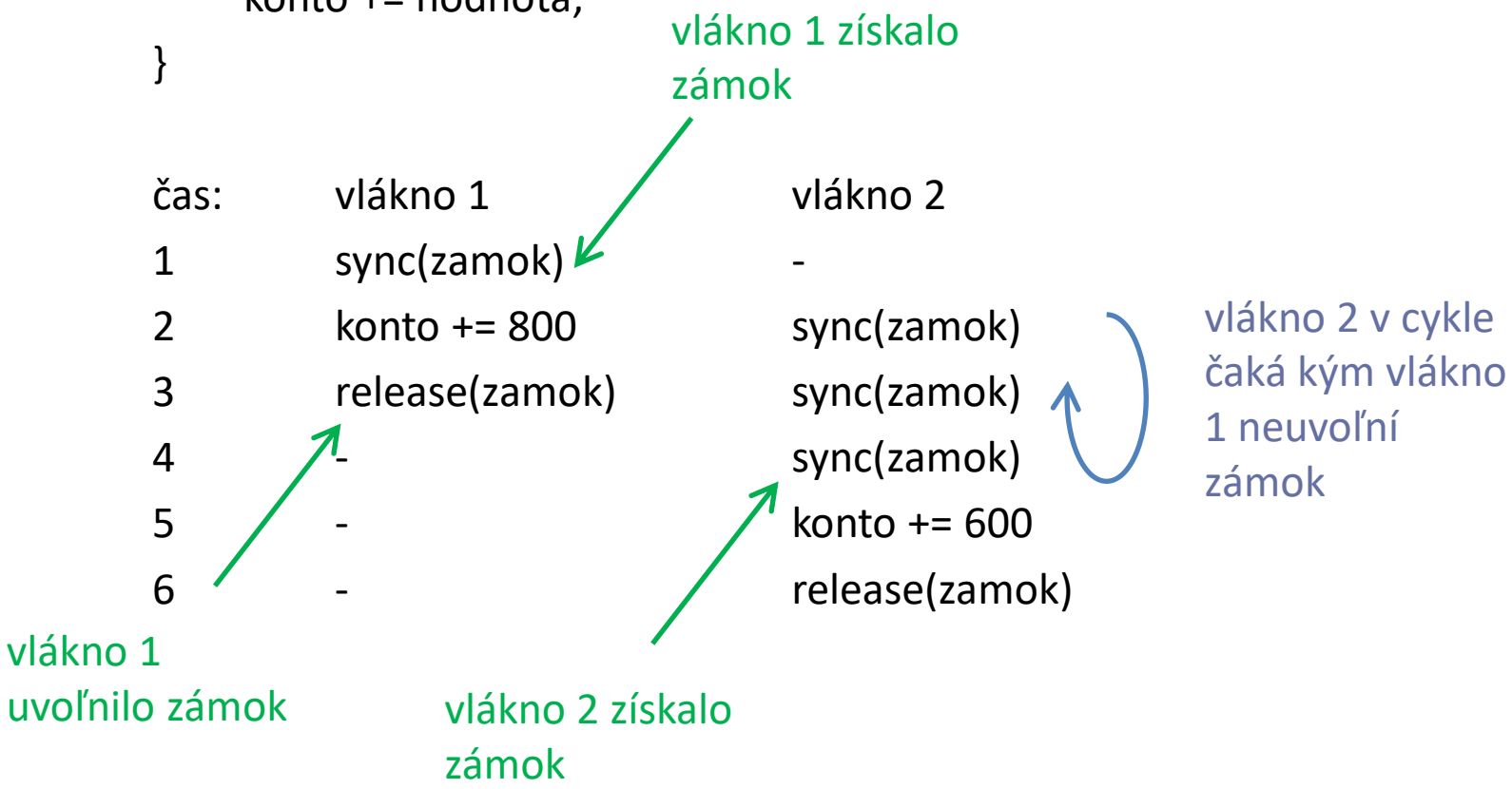
Zámok (2)

- Procesor musí poskytovať inštrukciu, ktorá atomicky testuje hodnotu premennej a podľa výsledku testu ju zmení
- Ak máme dve vlákna A a B a premennú l ktorá slúži ako zámok

1. Na začiatku označenej kritickej sekcie sa každé vlákno pokúsi získať zámok:
2. Ak l = 0 potom nastav l = 1 a pokračuj výpočtom kritickej sekcie, inak v cykle čakaj kým l = 0 – atomická operácia
3. Po skončení výpočtu kritickej sekcie vlákno ktoré získalo zámok ho uvoľní nastavením l = 0 (ostatné vlákna môžu zámok získať a vykonať svoju sekciu)

Zámok (3)

```
synchronize(zamok)  {
        konto += hodnota;
}
```



Zámok (4)

- Chceme naprogramovať funkciu prevod pre prevod medzi kontami – atomicky sa musí hodnota z jedného konta odpočítať a do druhého pripočítať

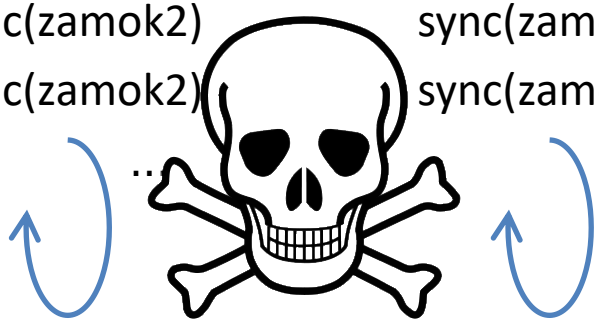
```
prevod(debet_konto, debet_zamok, kredit_konto, kredit_zamok, hodnota)
{
        sync(debet_zamok);
        sync(kredit_zamok);
        debet_konto -= hodnota;
        kredit_konto += hodnota;
        release(kredit_zamok);
        release(debet_zamok);
}
```

```
// Každé konto má svoj zámok
// (nechceme zbytočne blokovať všetky
// kontá)
```

Zámok (4)

- Čo sa stane ak vlákno 1 volá prevod(konto1, zamok1, konto2, zamok2, 800) a vlákno 2 prevod(konto2, zamok2, konto1, zamok1, 600) v tom istom čase?

čas:	vlákno 1	vlákno 2
1	sync(zamok1)	sync(zamok2)
2	sync(zamok2)	sync(zamok1)
3	sync(zamok2)	sync(zamok1)
	...	



vlákno 1 čaká kým vlákno 2 neuvoľní zámok 2

vlákno 2 čaká kým vlákno 1 neuvoľní zámok 1

Funkcionálne programovanie (1)

- Funkcionálne programovanie je vhodná paradigma pre paralelné výpočty
- Je možné oddeliť riadenie kódu od operácií, ktoré spracúvajú dáta
- Napr. namiesto zmiešaného cyklu:

```
for (int i = 0; i < v.length; i++) {
        v[i] = v[i] * v[i];
}
```

Oddelíme iteráciu od operácie nad dátami do samostatných funkcií

```
square(x) { return x * x }          v.forEach(square)
```

- Stačí zmeniť implementáciu forEach na jednom mieste na paralelnú verziu a všetky operácie, ktoré využívajú interovanie cez veľké vektory sa zrýchlia

Funkcionálne programovanie (2)

- Funkcie majú na vstupe parametre z ktorých vypočítajú výstupnú hodnotu
- Funkcia priamo nezmení hodnoty parametrov, na výstupe sa vytvorí nová hodnota ktorá sa už ďalej nebude meniť
- Zložitejšie spracovanie je možné dosiahnuť zreťazením funkcií
- Keďže sa po vytvorení už hodnoty dát nikdy nemenia, nie je potrebná synchronizácia
  - Menej zámkov a kritických sekvencií zjednodušuje programovanie a zlepšuje výkon, pretože sa vlákna navzájom neblokujú

Knižnice pre paralelné programovanie

- BLAS – Basic Linear Algebra Subprograms
  - Základné funkcie pre prácu s maticami a vektormi
  - Podpora viacjadrových/procesorových systémov, automatická detekcia GPU
- LAPACK - Linear Algebra PACKage
  - Využíva BLAS pre riešenie rôznych úloh z lineárnej algebry, napr. SVD dekompozícia, riešenie sústavy lineárnych rovníc, atď.
- Všeobecné optimalizačné metódy
  - Vhodné aj pre učenie modelov, napr. gradientové metódy
- Grafové algoritmy (napr. max. strom pokrývajúci všetky uzly grafu, max. spojené komponenty, najkratšia cesta, atď.)
- Spracovanie signálov - Furiérova fransformácia