Prednáška 1: Úvod

1. Čo je paralelná architektúra a čo je paralelný počítač

Paralelný počítač si môžeš predstaviť ako **súbor výpočtových (procesných) elementov**, ktoré **pracujú súčasne** a **spolupracujú** na riešení jedného problému. Dôležité nie je len to, že prvkov je veľa, ale to, že systém je navrhnutý tak, aby vedel efektívne riešiť veľké úlohy: rozdeliť prácu, prenášať dáta, synchronizovať sa a dosahovať reálny nárast výkonu.

Paralelná architektúra sa preto nezaoberá iba „koľko jadier máme“, ale hlavne tým, **ako je systém organizovaný** a aké kompromisy robí medzi výkonom, cenou, spotrebou energie a programovateľnosťou.

Pri návrhu paralelného systému sa prirodzene objavia otázky ako:

- aká veľká má byť architektúra (koľko prvkov, aká kapacita pamäte),
- aký výkonný má byť každý prvok,
- koľko a aký typ pamäte potrebujeme,
- ako budú prvky komunikovať (zdieľaná pamäť, správy, sieť),
- ako zabezpečiť správnosť pri súbehu (synchronizácia a konzistencia dát),
- a ako to všetko ovplyvní výkon a škálovanie.

Úloha počítačového architekta je navrhovať systém tak, aby sa **maximalizoval výkon a programovateľnosť**, ale zároveň sa ostalo v rámci limitov technológie (fyzika čipov, pamäte) a nákladov (cena, energia, chladenie).

Paralelizmus je zároveň **alternatíva k samotnému zvyšovaniu frekvencie** a stáva sa „vodiacou perspektívou“ naprieč celým návrhom počítačov: od mikroarchitektúry jadra až po superpočítače.

---

2. Prečo sa to študuje práve dnes

Historicky existovalo veľa rôznych a často veľmi odlišných paralelných architektúr, často pevne zviazaných s konkrétnymi programovacími modelmi. Dnes však technologické trendy spôsobujú, že sa návrhy „zbiehajú“ ku podobným princípom – **notebook a superpočítač sú v základe prekvapivo podobné**, len v inej mierke.

Zároveň platí, že paralelné počítanie sa stáva nevyhnutným, lebo:

1. zvyšovanie výkonu už nevieme „lacno“ dosiahnuť iba zvyšovaním taktu,
2. rastie počet tranzistorov (máme z čoho stavať), ale treba ich využiť rozumne,
3. výpočty stále viac narážajú na pamäťové a energetické limity, ktoré sa riešia paralelizmom a lokalitou dát.

Dôležitý odkaz: nestačí učiť sa len názvy typov paralelných systémov (taxonómie). Treba rozumieť princípom a kompromisom: **organizácia zdrojov, replikácia, komunikácia, synchronizácia, výkon a škálovanie**.

---

3. Trendy aplikácií: čo tlačí na rast výkonu

Paralelizmus je odpoveďou na rastúce požiadavky aplikácií. Najsilnejší tlak prichádza z dvoch oblastí:

Vedecké výpočty

Sem patria simulácie a výpočty ako CFD (prúdenie), fyzika, chémia, biológia, modelovanie klímy, veľké numerické úlohy a podobne. Tieto úlohy často vyžadujú obrovský výpočtový výkon aj veľké objemy dát. V praxi existuje široké spektrum úloh: niektoré potrebujú relatívne málo výpočtu a viac pamäte, iné sú extrémne výpočtovo náročné (tzv. „grand challenge“ typ problémov).

Inžinierske a priemyselné výpočty

Veľké paralelné systémy sú oporou v mnohých odvetviach: automobilový priemysel (crash simulácie), letectvo (aerodynamika), ropný a plynárenský priemysel (spracovanie geodát), farmácia (modelovanie molekúl), CAD/CAE, vizualizácia (napr. filmové efekty), ale aj finančné modelovanie či databázové systémy.

V týchto oblastiach je častým cieľom buď:

- **znížiť čas riešenia** (rýchlejšie dostať výsledok),
- alebo zvýšiť **priepustnosť** (spracovať viac úloh za rovnaký čas).

---

4. Zrýchlenie (speedup) a čo v praxi znamená „paralelné je rýchlejšie“

Keď použijeme viac procesorov/jadier, prirodzene čakáme zrýchlenie. Pre fixnú veľkosť problému sa často používa jednoduchá interpretácia: výkon je úmerný 1/čas. Zrýchlenie pri použití p procesorov sa vyjadruje ako pomer výkonu (alebo ekvivalentne času):

- **Speedup(p) = čas na 1 procesore / čas na p procesoroch**
  (čo je to isté ako Performance(p) / Performance(1), ak Performance ~ 1/čas)

Ideálny stav by bol, že speedup(p) = p, ale v realite to zvyčajne neplatí. Dôvody sú typicky:

- režijné náklady na komunikáciu a synchronizáciu,

- časti programu, ktoré sa nedajú paralelizovať,
- nerovnomerné rozdelenie práce (niekto čaká),
- úzke hrdlá pamäte (veľa jadier čaká na dáta).

Preto je v paralelných systémoch kľúčová téma **škálovateľnosť**: ako sa výkon mení, keď pridávame ďalšie výpočtové zdroje.

---

5. Ako sa zvyšoval výkon počítačov: technológia + architektúra

Zvyšovanie výkonu bolo historicky výsledkom kombinácie dvoch vecí:

1. **Pokrok v polovodičovej technológii** – zmenšovanie „feature size“, zvyšovanie hustoty tranzistorov, zvyšovanie možnej rýchlosti obvodov.
2. **Pokrok v architektúre** – nové princípy návrhu procesorov a systémov, ktoré lepšie využívajú dostupné tranzistory.

S rastom výkonových požiadaviek sa presadzovali prístupy, ktoré zvyšovali efektivitu vykonávania programov (napr. RISC ako zjednodušenie a zrýchlenie inštrukčného vykonávania). V širšom kontexte sa architektúry posúvali tak, aby bolo možné stavať výkonnejšie a zároveň praktickejšie (menšie, lacnejšie) počítače a systémy.

---

6. Hype Cycle (Gartner) a prečo je to v úvode

Technológie často prechádzajú typickým „cyklom očakávaní“: najprv veľký optimizmus, potom sklamanie, potom realistické nasadenie. Hype Cycle je grafická predstava toho, ako sa vyvíja viditeľnosť technológie, očakávania, adopcia a reálny prínos.

Typických je 5 fáz:

1. **Technologický impulz** – objaví sa novinka, prvé prototypy, veľká pozornosť.
2. **Vrchol prehnaných očakávaní** – veľa publicity, sľuby často preháňajú realitu.
3. **Údolie vytriezvenia** – prídu limity, časť projektov zlyhá, záujem klesne.
4. **Svah osvietenia** – postupne sa ukazuje, kde technológia skutočne dáva zmysel, vznikajú dobré postupy.
5. **Vrchol produktivity** – technológia je bežná, prínosy sú stabilné a merateľné.

Pre paralelné systémy je poučenie jednoduché: hodnotu technológie netreba posudzovať podľa „hype“, ale podľa toho, **aký má prínos v konkrétnom použití**, aké má náklady (energia, cena, zložitosť) a akú má programovateľnosť.

---

7. Technologické trendy: tranzistory pribúdajú, ale frekvencia a energia limitujú

Všeobecné technologické trendy ukazujú, že:

- výkon procesorov rástol dlhodobo, ale nie neobmedzene,
- počet tranzistorov rástol veľmi výrazne (Mooreov zákon ako historická aproximácia),
- kapacita pamätí rástla rýchlo, ale latencie a priepustnosť nerástli rovnakým tempom.

Kľúčová otázka je: **ako efektívne využiť rastúci počet tranzistorov**, keď samotné zvyšovanie taktu je obmedzené? Dve typické odpovede, ktoré sa opakujú v celej oblasti paralelných systémov, sú:

- **zvýšiť paralelizmus vo výpočtoch** (robiť viac práce naraz),
- **zlepšiť lokalitu prístupu k dátam** (mať dáta „bližšie“, používať cache a hierarchie).

Tieto dve stratégie sa často premietajú do snahy znižovať CPI (počet cyklov na inštrukciu) a znižovať čas strávený čakaním na pamäť.

---

8. Energetická bariéra: prečo zvyšovanie frekvencie naráža na strop

Výkon sa nedá donekonečna zvyšovať iba zvyšovaním taktu, pretože spotreba a teplo rastú príliš rýchlo. Typická závislosť dynamickej spotreby sa popisuje vzťahom približne:

$$P \sim C \cdot V^2 \cdot f$$

kde C súvisí s kapacitami prepínajúcich obvodov, V je napätie a f je frekvencia. Praktický problém je, že pri snahe o vyššie frekvencie často rastie aj napätie, a kvôli V^2 potom spotreba rastie ešte rýchlejšie. Výsledkom je, že teplo na jednotku plochy čipu sa stáva limitujúce a chladenie prestáva byť jednoduché.

To je jeden z hlavných dôvodov, prečo sa moderné procesory posunuli smerom k **viacjadrovosti a energetickej efektivite**, namiesto nekonečného zvyšovania taktov.

---

9. CPU vs. GPU: dva prístupy k výkonu

Moderné systémy často kombinujú CPU a GPU, pretože majú odlišné silné stránky.

CPU je typicky optimalizované na:

- nízku latenciu,

- zložité riadenie toku programu,
- širokú univerzálnosť,
- menší počet veľmi výkonných jadier.

GPU je typicky optimalizované na:

- vysokú priepustnosť (throughput),
- veľa jednoduchších paralelných výpočtových jednotiek,
- efektívne spracovanie dátovo-paralelných úloh (rovnaká operácia na veľa dátach),
- často aj lepší pomer výkon/watt v určitých triedach úloh.

Zmysel pre paralelné systémy: ak sa úloha dá dobre rozdeliť na veľa podobných operácií, GPU vie priniesť obrovský výkon. Ak úloha obsahuje zložité vetvenie a nepravidelné prístupy k dátam, CPU môže byť vhodnejšie. Preto sa v praxi bežne používa heterogénny prístup: CPU + akcelerátor.

---

10. Pamäťová a úložisková „stena“: kapacita rastie rýchlo, rýchlosť nie

Pri pamätiach (a podobne aj pri úložiskách) je dôležité pochopiť, že kapacita sa zvyšovala rýchlejšie než rýchlosť a latencia. To vytvára situáciu, kde procesor často čaká na dáta. Preto architektúry masívne používajú:

- **hierarchiu pamätí** (registre → cache → RAM → disk/SSD),
- **cache a buffre** na držanie nedávno použitých dát,
- snahu o **lokalitu** (pracovať s dátami tak, aby sa čo najčastejšie opakovane využívali dáta, ktoré už sú „blízko“),
- zvyšovanie **šírky pásma** (preniesť viac dát za čas).

Dôležitá myšlienka: paralelizmus sa neobjavuje len vo výpočte, ale aj v pamäťových systémoch. Napríklad pamäťové čipy často pracujú interne s veľkými blokmi dát, ktoré sa potom prenášajú cez užšie rozhranie, a preto má zmysel prenášať dáta po „väčších kusoch“ (riadky cache, burst prenosy).

---

11. Prenosové rýchlosti a prečo je šírka pásma kritická

Pri hodnotení pamäte a zberníc sa často rieši, koľko dát sa dá preniesť za sekundu. Typická predstava je, že prenosy prebiehajú po pevnej šírke (napr. 64 bitov = 8 bajtov na prenos) a celková priepustnosť závisí od počtu prenosov za sekundu. Pri paralelných systémoch sa veľa problémov zredukuje na to, či pamäťový subsystém dokáže zásobovať všetky výpočtové jednotky dátami bez dlhého čakania.

---

12. Nové pamäťové technológie: hľadanie lepšieho kompromisu

V oblasti pamätí sa hľadajú nové technológie, ktoré ponúknu lepší kompromis medzi výkonom, spotrebou a spoľahlivosťou. Objavujú sa smery ako nové typy nevolatilných pamätí a technológie s cieľom priblížiť pamäť výkonu procesora alebo znížiť energetické náklady (príklady smerov zahŕňajú rôzne rezistívne a magnetické pamäte či evolúciu flash pamätí). Pointa nie je poznať fyziku detailne, ale chápať motiváciu: dát je viac, výpočtu je viac, a pamäť sa stáva kľúčovým limitom, takže inovácie v pamäti priamo ovplyvňujú výkon paralelných systémov.

---

13. Architektonické trendy: paralelizmus vs. lokalita

Architektúra „premieňa technologické možnosti“ (tranzistory, pamäte, spoje) na reálny výkon a funkčnosť. Dlhodobo sa v návrhu opakuje kompromis:

- viac paralelizmu vo výpočte (viac jednotiek),
- versus viac investície do lokality dát (väčšie a lepšie cache, menšie latencie).

Koľko zdrojov ide do výpočtu a koľko do pamäťového subsystému je rozhodujúce pre výkon. Pri mnohých úlohách totiž nie je limitom počet operácií, ale to, či má procesor dáta včas.

Z historického pohľadu sa často spomínajú „generácie“ architektúr podľa dominantnej technológie: elektrónky, tranzistory, integrované obvody a VLSI. Najvýraznejším trendom éry VLSI je práve rast paralelizmu.

---

14. Úrovne paralelizmu: bit-level → ILP → thread-level

Paralelizmus sa v počítačoch objavuje na viacerých úrovniach:

Bit-level paralelizmus

Zvyšovanie šírky dátovej cesty (napr. 8-bit → 16-bit → 32-bit → 64-bit) umožnilo spracovať viac bitov naraz. Tento trend však prirodzene narazí na limity užitočnosti (pre veľa aplikácií je 64-bit „dosť“ a ďalšie rozširovanie už nie je hlavný zdroj výkonu).

ILP – Instruction-Level Parallelism (paralelizmus inštrukcií)

Procesor sa snaží vykonať viac inštrukcií súčasne pomocou techník ako pipeline, superskalárne vykonávanie, preusporiadanie inštrukcií a predikcia vetvenia. Toto vie zrýchliť

program bez toho, aby programátor písal viac vlákien. ILP však naráža na limity: závislosti medzi inštrukciami, vetvenie, chyby predikcie, čakanie na pamäť a rastúca zložitosť hardvéru.

Thread-/Task-level paralelizmus (hrubozrnný paralelizmus)

Keď ILP prestáva stačiť, najviac „škálovateľný“ prístup je spúšťať viac vlákien alebo úloh paralelne na viacerých jadrách či procesoroch. Dnešné mikroprocesory sú preto prakticky viacjadrové systémy a mnoho moderných platforiem ide ešte ďalej – do multiprocesorových serverov, clusterov a systémov s GPU.

---

15. Pipeline hĺbka a frekvencia: prečo sa trend menil

Jedným z historických prístupov k vyššej frekvencii bolo prehlbovanie pipeline (viac stupňov). Hlbšia pipeline môže pomôcť dosiahnuť vyšší takt, ale prináša aj nevýhody: vyššiu citlivosť na vetvenie (tresty pri zlej predikcii), väčšiu režijnú zložitosť a často horšiu energetiku. Aj to prispelo k tomu, že sa pri zvyšovaní výkonu viac presadilo pridávanie jadier a paralelizmus na vyššej úrovni.

---

16. Vedecké „superpočítanie“, clustre a cloudové služby

V HPC (High Performance Computing) bol paralelizmus vždy základom. Mikroprocesory výrazne zlepšili výkon aj vo floating-point operáciách, ale ak chceme riešiť naozaj veľké problémy, potrebujeme kombinovať veľa výpočtových prvkov.

Dve veľké výzvy sú:

- **vývoj paralelného softvéru** (ako program rozdeliť, ako sa vyhnúť chybám a dosiahnuť zrýchlenie),
- **návrh vhodných architektúr** (pamäť, sieť, škálovanie, energia).

Moderné vedecké výpočty sa často opierajú o clustre a infraštruktúry poskytované aj ako služby (IaaS/PaaS/SaaS). V praxi to znamená, že výpočtové zdroje a platformy môžu byť poskytované „ako služba“ a používateľ využíva veľké paralelné prostredie bez toho, aby fyzicky vlastnil celý superpočítač.

Pri veľkých systémoch je kritická aj **prepojovacia sieť (interconnect)** medzi uzlami, pretože komunikácia môže výrazne ovplyvniť škálovateľnosť.

---

17. Zhrnutie úvodu: čo je hlavná myšlienka prednášky

Hlavný dôvod, prečo paralelné systémy dominujú modernému výkonu, je kombinácia technologických faktov: tranzistorov je veľa, ale frekvencia a energia sú limitované, a pamäťový subsystém je často úzke hrdlo. Preto sa výkon posúva cez paralelizmus na viacerých úrovniach a cez dôraz na lokalitu dát.

Oblasť je stále atraktívnejšia aj kvôli novým aplikáciám (strojové učenie, bioinformatika, geoinformatika, veľké dátové analýzy). V rámci predmetu sa dôraz kladie najmä na multiprocesorové/viacjadrové systémy, veľké paralelné platformy a systémy využívajúce GPGPU.

Prednáška 2-a: Konvergencia paralelných architektúr

1) Prečo sa hovorí o „konvergencii“

Historicky boli paralelné architektúry často úzko zviazané s konkrétnymi programovacími modelmi. To viedlo k tomu, že sa vyvíjali „divergentne“ – existovalo veľa odlišných smerov (napr. SIMD, zdieľaná pamäť, odovzdávanie správ, dataflow, systolické polia) a nebolo jasné, ktorý prístup bude dominovať. Táto neistota paradoxne pomohla aj vývoju paralelného softvéru, lebo vznikali rôzne modely a techniky.

Dnes však technológia, cena a požiadavky aplikácií tlačia systémy k podobnému jadru. „Konvergencia“ znamená, že aj keď sa architektúry líšia detailmi, v praxi sa čoraz viac podobajú v tom, ako vyzerá uzol (CPU + pamäť) a hlavne v tom, ako riešia komunikáciu medzi uzlami alebo vláknami.

---

2) Rozšírenie pojmu architektúra: od ISA ku komunikačnej architektúre

Starý pohľad na „počítačovú architektúru“ bol zameraný najmä na ISA (inštrukčnú sadu) – teda čo vie procesor vykonať a ako vyzerajú inštrukcie. Novší pohľad rozširuje architektúru o podporu spolupráce a komunikácie: do popredia sa dostáva **komunikačná architektúra**. Tá neurčuje len „akú aritmetiku vieme spraviť“, ale najmä:

- aké abstrakcie a primitíva komunikácie sú k dispozícii,
- kde sú hranice medzi HW a SW,
- aké organizačné štruktúry to implementujú (či je to priamo hardvér, operačný systém, knižnice alebo používateľský runtime).

Dôležitý fakt je, že **kompilátory, knižnice a operačný systém** sa stali „mostami“, ktoré spájajú programovací model s reálnou implementáciou.

3) Moderný vrstvený rámec: čo je programovací model, komunikačná abstrakcia a implementácia

Je užitočné myslieť na systém vrstvovo. Na vrchu sú aplikácie (vedecké výpočty, databázy, CAD, paralelné aplikácie). Tie programátor píše v určitom **programovacom modeli**. Pod tým je **komunikačná abstrakcia** – teda to, aké komunikačné primitíva sú dostupné na úrovni používateľa (napr. zdieľaná pamäť, send/receive, globálne operácie nad dátami). Pod tým je používateľsko-systémové rozhranie a implementácia cez prekladač/knižnice, OS podporu, komunikačný hardvér a nakoniec prepojovaciu sieť.

Programovací model (čo „vidí“ programátor)

Programovací model určuje, ako programátor vyjadruje komunikáciu a synchronizáciu. Typické príklady:

- **Multiprogramovanie**: procesy bežia „vedľa seba“, typicky bez explicitnej komunikácie na úrovni programu.
- **Zdieľaný adresný priestor**: programy komunikujú cez spoločné premenné – ako nástenka (bulletin board).
- **Odovzdávanie správ (message passing)**: explicitná bod-bod komunikácia – ako posielanie listov alebo telefonátov.
- **Dátový paralelizmus**: viac regimentovaný model s globálnymi operáciami nad dátami; dá sa implementovať nad zdieľanou pamäťou aj nad správami.

Komunikačná abstrakcia (kľúčové rozhranie)

Komunikačné primitíva na používateľskej úrovni „realizujú“ programovací model a musia byť:

- jednoznačne definované (aby ten istý program fungoval správne na rôznych strojoch),
- dosť jednoduché a skladateľné (aby sa dali optimalizovať),
- a zároveň nie príliš špecifické (aby architektúra neblokovala výkonové triky a nové technológie).
  Preto sa na túto vrstvu pozerá ako na „zmluvu“ medzi HW a SW.

Komunikačná architektúra (rozhranie + implementácia)

Komunikačná architektúra je v praxi:

- **rozhranie používateľ/systém** (aké primitíva sú dostupné),
- plus **implementácia** (ako sú tieto primitíva realizované a integrované do uzla; aká je sieť, sieťové rozhranie, kontroléry atď.).

Ciele sú typicky: výkon, široká použiteľnosť, programovateľnosť, škálovateľnosť a nízka cena.

---

4) Jadro historických modelov, ktoré viedli ku konvergencii

Vývoj sa často vysvetľuje cez niekoľko „jadrových“ konceptov:

1. zdieľaný adresný priestor (SAS),
2. odovzdávanie správ (message passing),
3. dátovo paralelný prístup,
   a ako doplnkové smery sa uvádzajú dataflow a systolické polia.

---

5) Zdieľaný adresný priestor (SAS, shared memory): čo to je a prečo bol prirodzený

V SAS architektúrach môže každý procesor priamo adresovať ľubovoľné miesto v pamäti a komunikácia prebieha implicitne ako dôsledok load/store operácií. Veľká výhoda je **transparentnosť umiestnenia**: programátor zvyčajne nerieši, kde presne dáta fyzicky sú. Model je prirodzeným rozšírením uniprocesorového sveta (time-sharing), len s tým rozdielom, že vlákna/procesy bežia na rôznych procesoroch. SAS sa dá škálovať od malého počtu procesorov až po stovky. Zároveň sa často hovorí „zdieľaná pamäť“, ale to môže byť nejednoznačné, lebo pamäť môže byť fyzicky distribuovaná.

Dôležitá vlastnosť SAS je, že zápisy do zdieľanej adresy majú byť (nejakým spôsobom) viditeľné aj pre iné vlákna. Na bežnú komunikáciu sa používajú pamäťové operácie a na synchronizáciu typicky špeciálne atomické operácie. OS tiež využíva zdieľanú pamäť na koordináciu procesov.

---

6) Komunikačný hardvér pri SAS: od „mainframe“ prístupu po SMP

Ak už máme procesor, pamäťové moduly a I/O kontroléry prepojené interconnectom, prirodzený nápad je: kapacitu pamäte zväčšujem modulmi, I/O kontrolérmi, a pre vyšší výpočtový výkon jednoducho **pridám procesory**. Motivácia môže byť vyššia priepustnosť multiprogramovania alebo podpora paralelných programov.

„Mainframe“ prístup (crossbar a viacstupňové prepoje)

V klasickom „mainframe“ štýle sa používali prepoje typu crossbar (alebo ich viacstupňové varianty). Výhodou je škálovanie šírky pásma s počtom procesorov, nevýhodou sú vysoké náklady (najprv limitoval procesor, neskôr cena prepoja).

„Minicomputer“ prístup (zbernica) = SMP

S rozšírením mikroprocesorov sa rozšíril zbernicový prístup, často motivovaný multiprogramovaním a transakčným spracovaním. Tak vznikol pojem **SMP (symetrický multiprocesor)**. Latencia býva vyššia než pri uniprocesore a zbernica sa stáva úzkym hrdlom šírky pásma, preto je kľúčové kešovanie (a s ním súvisiaca koherencia). Výhodou je relatívne nízky prírastkový náklad pri zväčšovaní systému.

Ako príklady sa v materiáli uvádzajú reálne SMP systémy (napr. Intel Pentium Pro Quad a servery SUN/Oracle SPARC Enterprise s Jupiter interconnectom), kde je cieľom vysoká šírka pásma a konzistentne nízka latencia medzi komponentmi.

Typické charakteristiky SMP

V SMP všetky procesory „vidia“ všetko (pamäť, I/O, prerušenia). Existuje jedno „jadro“ plánovania – scheduler rozhoduje, ktorá aplikácia beží na ktorom procesore a aplikácia môže migrovať. Typicky procesory nezdieľajú cache (výnimkou sú viacjadrové CPU, kde sa často zdieľa L3/L4).

Problémy SMP

SMP môže byť náročné na zostavenie a údržbu (duplicitný hardvér, chladenie). Horšie škáluje, lebo zdieľaná pamäť môže vytvárať „hot spoty“ – veľa vlákien sa serializuje na rovnakých dátach. Migrácia procesov môže zhoršiť využitie cache (flush pri migrácii). A s rastom počtu procesorov rastie riziko neželaných súbehov (race conditions), takže treba dôslednú synchronizáciu.

---

7) Škálovanie „nahor“: prečo prichádza NUMA a distribuovaná pamäť

Pri škálovaní nahor sa prepoj stáva problémom: buď je crossbar príliš drahý, alebo zbernica nemá šírku pásma. Jeden z konceptov je „dance-hall“ – šírka pásma sa dá škálovať, ale latencie do pamäte sú uniformné, lenže uniformne veľké. Preto sa objavuje **distribuovaná pamäť** alebo **NUMA (Non-Uniform Memory Access)**, kde nie je prístup do pamäte rovnaký pre všetky adresy (lokálna pamäť je lacnejšia/rychlejšia než vzdialená). V takýchto systémoch často „skladáš“ zdieľaný adresný priestor pomocou správ v rámci siete (napr. read-request/read-response).

Ako príklad škálovania sa uvádza napr. Cray T3E, ktorý využíva prepojovaciu sieť (napr. 3D torus), škáluje na tisíce procesorov a spolieha sa na cache hierarchiu.

---

8) Many-core a „zlá správa“: viac jadier ≠ automaticky vyšší výkon

Materiál upozorňuje na dôležitú modernú realitu: keď sa výkon procesorov prestal zvyšovať najmä taktom, výrobcovia pridávajú viac jadier na jeden čip. To však prináša problém pre dátovo náročné aplikácie a superpočítače. Simulácie (Sandia National Laboratories) ukázali, že pri 8/16/32-jadrových mikroprocesoroch sa výkon môže prestať zvyšovať, prípadne aj klesať, lebo limitom je priepustnosť pamäte a nevhodné mechanizmy správy pamäte pre takúto mieru paralelizmu.

Jadro problému sa dá zhrnúť ako **pamäťový múr**: počet jadier rastie, ale počet prepojení z čipu do zvyšku počítača nerastie rovnakým tempom. Udržať všetky jadrá zásobené dátami je ťažké a dáta môžu byť fyzicky veľmi ďaleko (v racku, cez sieťové prvky). Preto sa ako smerovanie uvádza tesnejšia (a možno „inteligentnejšia“) integrácia pamäte a procesorov.

---

9) Architektúry s odovzdávaním správ (message passing): základná idea

V message passing architektúrach je základným stavebným blokom kompletný počítač (uzol) vrátane I/O. Proces má priamy prístup len k **privátnemu adresnému priestoru** a komunikácia prebieha explicitne cez správy (send/receive). Takýto systém je podobný klastrom pracovných staníc, len s tesnejšou integráciou. Programovací model je viac oddelený od základných hardvérových operácií a často vyžaduje zásah knižnice alebo OS.

Abstrakcia send/receive (čo to znamená)

Send typicky špecifikuje buffer, ktorý sa má preniesť, a cieľový proces. Receive špecifikuje zdrojový proces a aplikačné úložisko, kam sa dáta majú uložiť. Prakticky ide o kopírovanie pamäť→pamäť a je nutné vedieť pomenovať procesy. Často sa používa aj tag (značka) a receive používa pravidlo párovania (matching). V najjednoduchšej forme párovanie send/recv zároveň realizuje bod-bod synchronizáciu.

Nevýhodou sú overheady: kopírovanie, správa bufferov, ochrana (protection) a podobne. Preto je dôležité, kde je hranica medzi HW a SW – čo rieši hardvér a čo knižnice/OS.

Ako sa vyvíjali message passing stroje

Skoré stroje mali FIFO na každom linku, hardvér bol blízko programovaciemu modelu a operácie boli skôr synchrónne. Neskôr sa presadilo DMA, čo umožnilo neblokujúce operácie a prijímač mohol správy bufferovať až do momentu, keď program vykoná receive. Zároveň sa postupne znižovala úloha topológie, lebo prechod od „ulož-a-posuň“ smerovania k

prúdovému/zreťazenému smerovaniu zjednodušoval používanie siete. Čoraz viac „ceny“ a výkonu bolo koncentrovaných v rozhraní uzol–sieť.

Ako príklady sa uvádzajú konkrétne systémy ako Intel Paragon a IBM SP-2, kde vidno integráciu sieťového rozhrania, DMA a limity vyplývajúce z toho, cez akú zbernicu sa komunikácia pripája.

---

10) Prečo sa modely začali zlievať: softvér rozmazal hranice

Kľúčová myšlienka konvergencie je, že moderný softvér dokáže „emulovať“ alebo realizovať vlastnosti jedného modelu na inom hardvéri, a tým sa hranice medzi modelmi rozmazali.

Príklady:

- Na zdieľanej pamäti sa dajú správy realizovať cez buffre: „send“ znamená zapísať dáta do buffra, „receive“ číta zo zdieľaného úložiska, a signalizácia (napr. príchod správy) sa rieši príznakmi alebo zámkami.
- Na message passing stroji sa dá zostrojiť globálny adresný priestor: čítanie sa spraví odoslaním požiadavky procesu, ktorý objekt vlastní, a prijatím odpovede. Pre používateľa to môže byť skryté (napr. kompilátor vygeneruje kód, ktorý pri prístupe ku „zdieľanej premennej“ v skutočnosti robí transakciu správ).
- Dokonca aj zdieľaný virtuálny adresný priestor sa dá vytvoriť na message passing stroji na úrovni strán: procesy majú zdieľanú oblasť adries, ale lokálne stránky; pri prístupe k vzdialenej stránke vznikne page fault a OS iniciuje prenos stránky a namapovanie do adresného priestoru.

Toto je jadro konvergencie: programovací model nemusí priamo prezrádzať, ako vyzerá hardvér – medzi nimi je komunikačná abstrakcia a softvérové „mosty“.

---

11) Dátovo paralelné systémy: čo je ich podstata

Dátový paralelizmus znamená, že sa operácie vykonávajú paralelne nad prvkami dátovej štruktúry. Logicky existuje jedno riadiace vlákno, ktoré vykonáva kroky (sekvenčné alebo paralelné), a konceptuálne je ku každému dátovému prvku „priradený“ procesor. Architektonicky to vyzerá ako pole mnohých jednoduchých lacných procesorov (PE), každý s malou pamäťou, pripojených k riadiacemu procesoru, ktorý vydáva inštrukcie. Výhodou býva lacná globálna synchronizácia a špeciálna komunikácia prispôsobená tomuto štýlu spracovania. Pôvodná motivácia bola riešenie jednoduchých numerických úloh (napr. diferenciálne rovnice) a snaha centralizovať drahé načítanie a „serializáciu“ inštrukcií.

---

12) Dataflow architektúry: výpočet ako graf závislostí

Dataflow prístup reprezentuje výpočet ako graf závislostí. Uzol (logický procesor) sa aktivuje až vtedy, keď má dostupné operandy. Správy (tokeny) nesú tag ďalšej inštrukcie a posielajú sa ďalej. V module spájania (matching store) sa tag porovnáva s ostatnými a zhoda spustí vykonanie. Pointa je, že riadenie toku nevychádza zo sekvenčného programu, ale z dostupnosti dát.

---

13) Systolické architektúry: pole PE a orchestrácia toku dát

Systolické architektúry nahrádzajú jeden procesor poľom rovnakých procesných elementov. Cieľ je vysoká priepustnosť pri menšom počte prístupov do pamäte tým, že sa orchestruje tok dát v pravidelnom vzore. Je to odlišné od klasického prúdového spracovania, lebo ide o nelineárne pole s viac-smerovým tokom dát a každý PE môže mať malú lokálnu pamäť inštrukcií a dát.

Dôležité je aj odlíšenie od SIMD: v systolickom poli môže každý PE robiť niečo iné. Pôvodná motivácia súvisela s VLSI – dali sa vyrábať lacné špeciálne čipy a algoritmy boli priamo „vtelené“ do pravidelne prepojených elementov. V praxi (napr. iWARP) sa používali aj všeobecnejšie procesory, ktoré umožnili rôzne algoritmy na tom istom hardvéri, ale s dedikovanými kanálmi (prenos register→register). Tento prístup je však stále špecializovaný a pri všeobecných systémoch často naráža na podobné problémy ako SIMD; zároveň platí, že moderné všeobecné systémy dokážu tie isté algoritmy robiť dobre vďaka lokalite a cache.

---

14) Výsledok konvergencie: „generická“ paralelná architektúra

Materiál uzatvára konvergenciu predstavením spoločného rámca: typický paralelný systém sa dá chápať ako množina uzlov, kde každý uzol má procesor(y), pamäťový systém a komunikačnú podporu (sieťové rozhranie a komunikačný kontrolér) napojenú na škálovateľnú sieť. Konvergencia neznamená koniec inovácií – práve naopak, inovácie pokračujú, ale väčšinou už „v rámci“ tohto spoločného rámca (napríklad ako efektívne integrovať komunikačnú podporu do uzla a aké operácie podporiť).

Prednáška 2-b: Základné otázky návrhu

1) Prečo „programovací model“ ani „hardvér“ samé o sebe nestačia

Pri návrhu paralelnej architektúry nestačí pozerať sa len na tradičné taxonómie typu SIMD/MIMD ani iba na programovací model. Dôvod je praktický: často existujú veľmi odlišné implementačné architektúry, ktoré podporujú rovnaký programovací model, a zároveň tie isté hardvérové prvky sa dajú použiť rôznymi spôsobmi. Preto má zmysel sústrediť sa na také architektonické rozdiely, ktoré sa reálne prejavia v tom, ako bude vyzerať softvér: čo musí robiť kompilátor, ako vyzerá dobre optimalizovaná knižnica a ako sa paralelná aplikácia „premietne“ do strojového kódu.

Z tohto pohľadu sa návrh dá chápať ako problém obmedzený „zhora“ tým, ako programy architektúru používajú, a „zdola“ tým, čo je architektúra schopná poskytnúť. Preto je kľúčové rozumieť operáciám, ktoré sú dostupné na používateľskej úrovni komunikačnej abstrakcie, a vedieť, ako sa na ne mapujú programovacie modely a ako sa tieto primitíva mapujú na skutočný hardvér.

---

2) Komunikačná abstrakcia ako „zmluva“ medzi softvérom a hardvérom

Komunikačná abstrakcia je vrstva, ktorá tvorí kľúčové rozhranie medzi programovacím modelom a implementáciou systému. Z pohľadu softvéru musí mať presne definovaný význam, aby ten istý program bežal správne na rôznych implementáciách. Zároveň operácie v tejto vrstve majú byť jednoduché a skladateľné, aby sa dali efektívne optimalizovať. Z pohľadu hardvéru musí mať tá istá abstrakcia tiež jasný význam, aby architekt vedel, kde môže robiť výkonové optimalizácie bez porušenia softvérových predpokladov.

Abstrakcia pritom nemá byť „prešpecifikovaná“. Architekt chce, aby nezakazovala užitočné techniky zvyšovania výkonu a aby nebránila využitiu vlastností novších technológií. Preto sa o nej hovorí ako o „zmluve“: softvér aj hardvér sa môžu zlepšovať nezávisle, ale musia spolu pracovať korektne.

---

3) Čo presne špecifikuje paralelný programovací model

Paralelný program pozostáva z jedného alebo viacerých riadiacich vlákien, ktoré pracujú s dátami. Programovací model potom v zásade odpovedá na tri otázky: aké dáta môžu vlákna pomenovať, aké operácie môžu nad týmito dátami robiť a aké poradie medzi týmito operáciami existuje (teda čo je „dovolené“ a čo by už mohlo viesť k nejednoznačnosti alebo chybe).

Aby bolo jasné, čo to znamená, materiál najprv pripomenie uniprocesorový model: vlákno môže pomenovať miesta vo svojom virtuálnom adresnom priestore a registre stroja. Rozdiely v detailoch (segmentovaný vs. plochý adresný priestor, jazyky s/bez ukazovateľov a dynamickej alokácie) sú z pohľadu architektúry druhoradé – ISA vždy poskytuje základné operácie nad pomenovanými miestami. V RISC je typické, že do registrov sa načítava a

ukladá z pamäte a aritmetika sa robí nad registrami, kým staršie ISA môžu robiť aritmetiku aj priamo nad pamäťou. Kompilátor tieto rozdiely zväčša maskuje, takže programátor vníma operácie nad premennými.

---

4) Poradie operácií: programátor vidí „sekvenčný svet“, ale HW/SW preusporadúvajú

V uniprocesorovom programovom pohľade sa operácie vykonávajú v „sekvenčnom programovom poradí“: premenné sa čítajú a menia v poradí, ako sú napísané v programe, a čítanie má vrátiť poslednú zapísanú hodnotu v tomto poradí. Tento predpoklad je zásadný pre logiku programov.

Realita je však taká, že kompilátor aj hardvér často preusporadúvajú čítania a zápisy kvôli výkonu, len to musia urobiť tak, aby program „nespoznal“, že sa poradie zmenilo. Kľúčové je zachovanie poradia závislostí: ak sa niečo zapíše a neskôr sa to číta, neskoršia operácia musí dostať správnu hodnotu. Nezávislé operácie sa však môžu presúvať, zápisy na rôzne adresy sa môžu meniť, ak sa neporušia závislosti vyplývajúce z čítaní. Takéto transformácie vznikajú pri kompilácii (napr. alokácia do registrov, úpravy výrazov, transformácie slučiek) aj v samotnom procesore (pipeline, vykonávanie viacerých inštrukcií za cyklus, write buffre na skrytie pamäťovej latencie).

Táto téma je v paralelných systémoch ešte citlivejšia, lebo pri viacerých vláknach sa otázka „aké poradie kto pozoruje“ stáva zdrojom chýb aj výkonových kompromisov.

---

5) Dva základné paralelné programovacie modely: SAS vs. message passing

Materiál potom postaví vedľa seba dva základné modely:

a) Zdieľaný adresný priestor (SAS – shared address space)

V SAS môže každý proces pomenovať ľubovoľnú premennú v zdieľanom priestore a základné operácie sú load/store (plus operácie potrebné pre (pre)usporiadanie). Najjednoduchší model poradia hovorí, že v rámci vlákna je poradie sekvenčné, medzi vláknami dochádza k „prekladaniu“ (ako pri timesharingu) a iné poradie sa vynucuje synchronizáciou. Zároveň platí, že kompilátor/hardvér môžu poradie porušiť (pre výkon), preto v praxi existujú aj jemnejšie pamäťové modely.

b) Odovzdávanie správ (message passing)

Tu procesy priamo pomenúvajú len privátne dáta – zdieľaný adresný priestor neexistuje. Komunikácia je explicitná cez send a receive: send prenáša dáta z privátneho priestoru

odosielateľa k prijímačovi a receive kopíruje dáta do privátneho priestoru prijímača. Programové poradie v rámci procesu ostáva, a send/receive môžu zároveň zabezpečiť bod-bod synchronizáciu. V message passing sa dá riešiť aj vzájomné vylúčenie (napr. zámkami/kritickými sekciami na logickej úrovni) a dokonca sa dá zostrojiť globálny adresný priestor ako „meno = (číslo procesu + adresa)“, lenže nad takými menami neexistujú priame hardvérové operácie – treba ich realizovať protokolom.

---

6) Adresovanie a operácie sa môžu líšiť medzi programovacím modelom a HW rozhraním

Veľmi dôležitá návrhová myšlienka je, že to, čo vidí programátor v programovacom modeli, nemusí byť totožné s tým, čo priamo podporuje hardvér. Ako príklad sa uvádza „zdieľaný virtuálny adresný priestor“: jedna implementácia môže mať priamu hardvérovú podporu cez mapovanie virtuálna→fyzická adresa (v→p) nad zdieľaným fyzickým priestorom.

Iná implementácia môže mať pre každý proces nezávislý fyzický adresný priestor a SAS sa poskytne cez operačný systém: mapovanie v→p je lokálne a vzdialené prístupy vyvolajú page fault, ktorý OS rieši (napr. presunom/namapovaním dát). Ešte vyššie sa to dá riešiť cez kompilátor alebo runtime: pracuje sa so zdieľanými objektmi a prístupy sa inštrumentujú, aby sa „zdieľanie“ simulovalo protokolom. Pointa je, že rovnaký programovací model môže mať úplne odlišné hardvérové požiadavky a výkonové dôsledky.

Podobne je to pri message passing: hardvér môže poskytovať len základný mechanizmus prenosu dát, no párovanie (matching) a buffrovanie často dáva zmysel riešiť flexibilnejšie v softvéri. Tu sa dostávame k typickému kompromisu na rozhraní používateľ/systém: ak OS zasahuje pri každom volaní, býva to drahé; výhodnejší býva prístup, keď sa OS inicializuje raz (alebo zriedkavo) a pri bežnej komunikácii je potrebná len minimálna účasť softvéru. Ďalšia možnosť je, že nižšie rozhrania poskytujú SAS a send/receive sa „poskladá“ z load/store operácií.

---

7) Poradie (ordering) v multiprocesoroch: prečo je to „jemná“ téma

Pri message passing sa typicky nepredpokladá všeobecné poradie medzi procesmi (okrem toho, čo si explicitne vynútiš protokolom a synchronizáciou). Pri SAS je otázka poradia zásadná: ide o to, ako procesy pozorujú poradie prístupov iných procesov. Keďže uniprocesory už samy „čarujú“ s poradím kvôli výkonu, v multiprocesoroch je táto téma ešte citlivejšia: treba rozumieť, ktoré staré optimalizačné triky stále platia a aké nové mechanizmy treba pridať, aby boli programy korektné aj rýchle.

---

8) Synchronizácia: dve hlavné potreby (vzájomné vylúčenie a poradie udalostí)

Synchronizácia rieši dve triedy problémov.

Prvá je **vzájomné vylúčenie (mutual exclusion)**: potrebujeme zabezpečiť, aby určitú časť kódu (kritickú sekciu) vykonávala naraz len jedna entita, typicky kvôli tomu, aby sa operácie nad zdieľanými dátami dali „usporiadať“ a aby nevznikli preteky. Intuícia je ako miestnosť, do ktorej môže vstúpiť iba jeden človek naraz.

Druhá je **synchronizácia udalostí**: potrebujeme vynucovať poradie udalostí tak, aby sa zachovali závislosti (napríklad producent musí vyrobiť dáta skôr, než ich konzument použije). Na to slúžia bariéry, podmienené premenné (condition variables) a mechanizmy typu signal/wait. Materiál rozlišuje tri základné typy synchronizácie udalostí: bod-bod, skupinovú a globálnu.

---

9) Replikácia dát: výkonová zbraň, ale s rôznymi dôsledkami podľa modelu

Replikácia je veľmi dôležitá, lebo znižuje prenos dát a odľahčuje komunikáciu. V uniprocesore to často „robí automaticky“ cache – cieľ je znížiť komunikáciu s pamäťou.

V message passing je replikácia prirodzená už samotným receive: prijatím dát si vytvoríš lokálnu kópiu (v istom zmysle nové „meno“) a následné zápisy odosielateľa už nie sú automaticky viditeľné. Preto je replikácia v tomto modeli explicitná a programátor (alebo knižnica) musí riešiť, kedy a ako sa kópie aktualizujú.

V SAS môže replikácia vzniknúť transparentne: load „prinesie“ dáta a v cache vznikne kópia bez toho, aby sa zmenilo meno objektu. OS môže replikovať aj na úrovni strán v zdieľanom virtuálnom adresnom priestore. Výsledkom je, že bez explicitného premenovania môže existovať veľa kópií toho istého „mena“, čo následne otvára otázku koherencie (aby všetky kópie dávali zmysel).

---

10) Výkon komunikácie: latencia, šírka pásma a „cena“

To, aké komunikačné mechanizmy (a aké algoritmické stratégie) sa v praxi použijú, je silne ovplyvnené výkonovými charakteristikami. Materiál vyzdvihuje tri základné charakteristiky:

- **Latencia** je čas potrebný na vykonanie operácie (napr. jedného prenosu alebo jedného vzdialeného prístupu).

- **Šírka pásma** je rýchlosť prenosu dát po rozbehu (steady-state), teda koľko dát za sekundu sa dá preniesť, keď už prenos „beží“.
- **Cena** je vplyv komunikácie na celkový čas vykonania programu (niekedy nejde len o samotný čas prenosu, ale aj o to, koľko práce procesor nemôže robiť popri tom).

V úplne jednoduchom svete, kde procesor robí vždy len jednu vec, by platilo približne, že šírka pásma je úmerná 1/latencii. V moderných systémoch je to zložitejšie, lebo komunikácia sa dá prekrývať s výpočtom, dá sa „pipelineovať“, môže existovať paralelnosť prenosov a do hry vstupuje kontencia (súťaženie o zdroje).

---

11) Lineárny model latencie prenosu dát: prečo je užitočný a kde nestačí

Základný model pre čas prenosu n bajtov (alebo slov) je:

$T(n) = T_o + n / B$

kde T_o je fixná zložka (štart/latencia) a B je limitná šírka pásma. Tento model sa dá použiť nielen pre message passing, ale aj pre prístup do pamäte, vektorové operácie či I/O. Ako n rastie, dosiahnutá priepustnosť sa približuje k B; to, ako rýchlo sa k nej priblíži, závisí od T_o. Typická „polovičná priepustnosť“ (keď dosiahneš B/2) vyjde pri veľkosti približne $\mathbf{n_{1/2} = T_o \cdot B}$ (vyplýva to priamo z definície B/2).

Zároveň však platí, že lineárny model nestačí na všetko. Pri reálnych systémoch je dôležité, kedy môžeš iniciovať ďalší prenos, či vieš komunikáciu prekrývať s výpočtom (overlap), a aký je konkrétny mechanizmus realizácie prenosu.

---

12) Model nákladov komunikácie: rozklad na zložky a význam kontencie

Pre detailnejšie uvažovanie sa používa rozklad času komunikácie na správu:

Tcomm = ov + oc + l + n/B + Tc

kde ov sú režijné náklady (overhead), oc je obsadenie linky (čas, počas ktorého zdroje „držím“), l je sieťové oneskorenie, n/B je prenosová zložka a Tc je kontencia (súťaženie o zdroje). Režijné náklady aj obsadenie linky môžu, ale nemusia závisieť od veľkosti správy n. Pri komunikácii cez viac prvkov (viac liniek/switchov) sa celkové oneskorenie správa ako súčet oneskorení, kým celkové obsadenie (teda efektívna 1/šírka pásma) býva dané najpomalším miestom v ceste.

Dôležitý je aj pohľad na „náklad“ v programe: ak sa komunikácia opakuje s frekvenciou f, potom celkový dopad závisí od toho, koľko z komunikácie vieš prekrývať (overlap). Materiál to vyjadruje ideou, že náklad komunikácie rastie s frekvenciou a so zvyškom komunikácie, ktorý sa nepodarí prekryť výpočtom. Tento model je všeobecný a dá sa použiť aj na situácie typu „cache miss“, kde tiež platí, že prenos dát má fixnú aj veľkostnú zložku a môže byť ovplyvnený kontenciou.

---

13) Záverečné zhrnutie: aké sú „základné otázky návrhu“

Materiál uzatvára, že paralelná architektúra je dnes súčasťou hlavného prúdu výpočtovej techniky a že kľúčová architektonická otázka je v komunikačnej architektúre – teda ako je komunikácia integrovaná do pamäťového a I/O systému uzla.

Základné návrhové otázky sa delia na dve skupiny:

- **Funkčné otázky:** adresovanie, operácie a poradie (ordering) – teda čo systém „znamená“ a akú korektnosť garantuje.
- **Výkonnostné otázky:** organizácia, replikácia a výkonové charakteristiky komunikácie – teda čo rozhoduje o reálnej rýchlosti.

A nakoniec: návrhové rozhodnutia sa majú robiť „workload-driven“, čiže na základe hodnotenia typických záťaží a reálneho spôsobu použitia systému. To je integrálna súčasť inžinierskeho prístupu k návrhu paralelných architektúr.

Prednáška 3: Klasifikácia počítačových architektúr

---

1) Koncept uloženého programu: z čoho sa skladá „klasický“ počítač

Východiskom pre väčšinu architektúr je **koncept uloženého programu**. Znamená to, že počítač vykonáva **sekvenciu inštrukcií**, ktoré pracujú nad dátami uloženými v **registroch** a v **pamäti**. Program si môžeme predstaviť ako množinu inštrukcií, pričom každá inštrukcia obsahuje:

- **kód operácie** (čo sa má spraviť),
- **operandy** (nad akými dátami sa to spraví).

Reálne je inštrukcia binárny kód a pri vykonávaní sa často rozkladá na jemnejšie kroky – **mikrooperácie** (to je už vnútorná „mechanika“ procesora). Z pohľadu architektúry je dôležité, že počítač typicky obsahuje pamäť, riadiace registre, riadiacu jednotku, vykonávaciu jednotku a inštrukčnú sadu (ISA).

Často sa spomína rozdiel **von Neumann vs. Harvard architektúra**. V praxi ide o to, či inštrukcie a dáta zdieľajú jednu pamäť a jeden prenosový kanál (von Neumann), alebo sú oddelené (Harvard). V moderných procesoroch sa tieto myšlienky často kombinujú (napr. oddelené cache pre inštrukcie a dáta, ale spoločná hlavná pamäť).

---

2) Čo znamená „klasifikácia“ architektúr a prečo sa robí

Klasifikácia je snaha zaradiť architektúry do tried podľa toho, **ako sú organizované** a **aký typ paralelizmu využívajú**. Nejde iba o pomenovanie typu (napr. „SIMD“), ale o pochopenie, čo to v praxi znamená: aké úlohy sa na tom riešia dobre, aké sú typické výhody a obmedzenia a čo to znamená pre výkon.

Ešte pred paralelnými taxonómiami sa architektúry dajú triediť aj všeobecne:

- podľa **účelu**: architektúry na všeobecné použitie vs. špeciálne použitie (špecializované stroje, akcelerátory),
- podľa **spracovania dát**: analógové, digitálne a hybridné systémy.

Hlavná časť materiálu však rieši klasifikáciu najmä z pohľadu paralelizmu.

---

3) Flynnova taxonómia: SISD, SIMD, MIMD, MISD

Flynnova taxonómia patrí medzi najznámejšie triedenia. Rozdeľuje architektúry podľa toho, koľko „prúdov“ (**streams**) inštrukcií a dát sa spracúva naraz:

- **SISD** (Single Instruction, Single Data): jeden prúd inštrukcií a jeden prúd dát
- **SIMD** (Single Instruction, Multiple Data): jeden prúd inštrukcií a viac prúdov dát
- **MIMD** (Multiple Instruction, Multiple Data): viac prúdov inštrukcií a viac prúdov dát
- **MISD** (Multiple Instruction, Single Data): viac prúdov inštrukcií nad jedným prúdom dát

Dôležité je brať to ako konceptuálny model. Reálne systémy často kombinujú viac princípov (napr. MIMD systém môže vnútri jadra používať SIMD inštrukcie).

---

4) SISD – sekvenčný počítač

SISD je tradičný **sériový (neparalelný)** počítač s deterministickým vykonávaním. V každom takte spracúva procesor iba **jeden prúd inštrukcií** a používa sa **jeden prúd dát**. Historicky sem patria staršie mainframy, minipočítače, pracovné stanice a PC s jedným procesorom/jadrom.

Keď si to predstavíš blokovo, typická SISD schéma je: pamäť inštrukcií → riadiaca jednotka → vykonávacia jednotka, a osobitne pamäť dát, z ktorej sa berú vstupy a do ktorej sa ukladajú výsledky. Všetko sa deje „jednou líniou“ riadenia.

V materiáli sa ako historické príklady uvádzajú:

- **Univac I (1951)** – prvý všeobecne použiteľný elektronický digitálny počítač,
- **TX-0 (1955–1956)** – jeden z prvých plne tranzistorových počítačov.

---

5) SIMD – dátový paralelizmus pod jedným riadením

SIMD znamená, že systém má deterministické správanie a v danom takte **všetky výpočtové jednotky vykonávajú tú istú inštrukciu**, ale každá jednotka môže pracovať s **iným dátovým prvkom**. Toto je prirodzený model pre úlohy, kde sa tá istá operácia opakuje nad veľkým množstvom dát (vysoká regularita) – typicky grafika alebo spracovanie obrazu.

Materiál zdôrazňuje, že SIMD sa často spája s dvomi príbuznými realizáciami:

- **vektorové architektúry**, ktoré síce majú podobnú „SIMD“ povahu (rovnaká operácia nad veľa prvkami), ale paralelizmus dosahujú najmä prúdovým spracovaním (**pipeliningom**) nad vektormi,
- **procesorové polia (array processors)**, kde existuje množstvo výpočtových jednotiek (ALU/PE), ktoré naraz vykonávajú rovnakú operáciu na rôznych dátach.

Procesorové pole (array processor) – ako vyzerá a čo znamenajú riadiace registre

Pri procesorovom poli je typický obraz taký, že existuje „hlavné“ riadenie, ktoré dekóduje inštrukciu, a táto inštrukcia sa potom vykoná paralelne na viacerých výpočtových prvkoch (ALU), pričom každý pracuje so svojimi dátami (A1,B1,C1 … AN,BN,CN).

Pri tejto schéme sa v materiáli vysvetľujú aj základné registre riadenia:

- **IP (Instruction Pointer, často PC – Program Counter)**: register s adresou nasledujúcej inštrukcie, ktorá sa má vykonať.
- **MAR (Memory Address Register)**: register, do ktorého sa uloží adresa pamäťovej bunky, s ktorou sa práve pracuje (čítanie/zápis).
- **MDR (Memory Data Register; niekedy MBR – Memory Buffer Register)**: register, ktorý drží dáta prenášané medzi CPU a pamäťou. Pri čítaní sem prídu dáta z pamäte, pri zápise sem CPU pripraví dáta, ktoré sa majú uložiť do pamäte.

Príklady SIMD v materiáli

Ako príklady sa uvádzajú:

- **ILLIAC IV (1965–1966)** – jeden z prvých masívne paralelných počítačov,
- **Connection Machine 1 a 2 (1987)** – prvý komerčný počítač navrhnutý priamo na simuláciu inteligencie a „života“.

---

6) MIMD – všeobecný paralelizmus (viac nezávislých tokov)

MIMD je dnes najbežnejší typ paralelného počítača. Vyznačuje sa tým, že:

- každý procesor (alebo jadro) môže vykonávať **vlastný prúd inštrukcií**,
- každý procesor pracuje so „svojimi“ dátami,
- procesory môžu bežať **asynchrónne**,
- správanie je často **nedeterministické** (pretože do výsledného „poradia“ udalostí vstupuje plánovanie, komunikácia a súbeh).

Do tejto kategórie spadajú viacjadrové procesory, multiprocesorové systémy a väčšina moderných superpočítačov. Blokovo si MIMD predstav ako viac „SISD jednotiek“ vedľa seba – každá má vlastnú pamäť inštrukcií, riadenie, výpočtovú jednotku a pamäť dát, pričom medzi nimi existuje nejaký mechanizmus komunikácie.

Príklady uvedené v materiáli:

- **IBM Blue Gene/L (1999)** – prvý superpočítač, ktorý pri reálnej aplikácii dosiahol udržateľne viac než 100 TFLOPS,
- **IBM Deep Blue (1996)** – prvý stroj, ktorý v šesťzápasovom súboji porazil úradujúceho majstra sveta v šachu (G. Kasparova).

---

7) MISD – viac inštrukčných tokov nad jedným dátovým tokom

MISD má deterministické vykonávanie a znamená, že každý procesor vykonáva **odlišný prúd inštrukcií**, ale všetky procesory pracujú nad **rovnakým prúdom dát**. V praxi sa táto kategória takmer nevyskytuje. Niekedy sa uvádza ako teoretická trieda alebo sa pripomína podobnosť s viacstupňovým prúdovým spracovaním (pipeline), kde dáta prechádzajú viacerými rôznymi „fázami“ spracovania.

---

8) Zhrnutie Flynnovej taxonómie (význam pre pochopenie paralelizmu)

Zmysel Flynnovej taxonómie je hlavne didaktický:

- **SISD**: sekvenčné spracovanie (1 prúd inštrukcií + 1 prúd dát),
- **SIMD**: dátový paralelizmus (1 prúd inštrukcií + viac prúdov dát),
- **MIMD**: úlohový/riadiaci paralelizmus (viac prúdov inštrukcií + viac prúdov dát),
- **MISD**: zriedkavé/teoretické (viac prúdov inštrukcií nad 1 prúdom dát).

Je dobré pamätať si aj to, že reálne systémy často kombinujú prístupy (napr. MIMD stroj, kde každé jadro používa SIMD inštrukcie).

---

9) Fengova klasifikácia: stupeň paralelizmu na úrovni bitov a slov

Tse-yun Feng navrhol klasifikáciu založenú na „stupni paralelizmu“ a pozerá sa na to, či sa spracovanie deje paralelne na úrovni **bitov** a/alebo na úrovni **slov**.

Základná idea je, že architektúra môže byť:

1. **WSBS** (Word Serial, Bit Serial): sériové slovo – sériový bit,
2. **WSBP** (Word Serial, Bit Parallel): sériové slovo – paralelný bit,
3. **WPBS** (Word Parallel, Bit Serial): paralelné slovo – sériový bit,
4. **WPBP** (Word Parallel, Bit Parallel): paralelné slovo – paralelný bit.

Intuícia je jednoduchá: niektoré stroje zrýchľujú tým, že spracúvajú širšie slová (bit-paralelizmus), iné tým, že spracúvajú viac slov naraz (word-paralelizmus), a niektoré robia oboje. Fengova schéma sa často kreslí ako graf, kde jedna os je dĺžka „bit slice“ a druhá os dĺžka slova; do grafu sa potom vkladajú príklady architektúr (v materiáli sa objavujú napr. PDP11, IBM370, Illiac IV, PEPE, STARAN…).

---

10) Erlangenský klasifikačný systém (ECS): tri úrovne zložitosti

Erlangenský systém pristupuje ku klasifikácii tak, že rozdeľuje systém do troch úrovní:

- **ELC (Elementary Logic Circuit level)**: úroveň elementárnych logických obvodov pracujúcich na úrovni jedného bitu,
- **ALU level**: úroveň aritmeticko-logickej jednotky, ktorá vykonáva sekvencie operácií postavené na ELC,

- **PCU (Program Control Unit) level**: úroveň programovej riadiacej jednotky, ktorá interpretuje program po inštrukciách.

ECS používa zápis:
⟨ K×K', D×D', W×W' ⟩
kde sa osobitne popisuje paralelizmus v oblasti **riadenia**, **dát** a **slova**.

V materiáli sú uvedené príklady zápisu pre konkrétne stroje (TI-ASC, CDC 6600, C.mmP, PEPE, Cray-1). Zmysel nie je učiť sa tieto konkrétne zápisy naspamäť, ale chápať, že ECS sa snaží zachytiť paralelizmus „systematicky" v rôznych vrstvách stroja, nie iba jedným jednoduchým označením.

---

11) Klasifikácia Sima–Fountain–Kacsuk (SFK): moderný pohľad podľa mechanizmu paralelizmu

SFK sa nepýta primárne „koľko prúdov inštrukcií/dát", ale **ako sa paralelizmus dosahuje**. Rozlišuje dve hlavné vetvy:

1. **Dátový paralelizmus** – paralelizmus dosiahnutý spracovaním viacerých dát naraz.
2. **Funkčný (riadiaci) paralelizmus** – paralelizmus dosiahnutý tým, že sa súbežne vykonáva viac funkcií/úloh, čo v praxi zodpovedá architektúram s paralelnými tokmi inštrukcií.

Tento pohľad je užitočný, lebo lepšie zodpovedá dnešnej realite: často riešime buď masívne dátovo-paralelné výpočty, alebo súbeh úloh/vlákien, prípadne kombináciu oboch.

---

12) Architektúry s dátovým paralelizmom: čo sem patrí a prečo

Medzi dátovo-paralelné architektúry sa v materiáli zaraďujú:

- **vektorové architektúry**,
- **SIMD architektúry**,
- **systolické architektúry**,
- **asociatívne a neurónové architektúry**.

Spoločná idea je, že výkon získavaš tým, že tú istú alebo veľmi podobnú operáciu aplikuješ nad veľkým množstvom dátových prvkov.

Vektorový procesor (zreťazené spracovanie)

Vektorový procesor sa často kreslí podobne ako „klasické“ jadro (IP, MAR, pamäť, MDR, dekóder, ALU), ale kľúčové je, že pracuje s vektormi a využíva **zreťazené (pipelined) spracovanie**, takže po rozbehnutí pipeline dokáže produkovať výsledky s vysokou priepustnosťou.

Procesorové pole

Pri procesorovom poli sa opäť objavuje schéma s mnohými ALU prvkami (A1,B1,C1 … AN,BN,CN), ktoré naraz vykonávajú tú istú operáciu. Výhoda je, že pri vysoko pravidelných úlohách vieš získať veľký výkon relatívne priamo.

---

13) Architektúry s paralelizmom riadenia: ILP, TLP, procesová úroveň

Druhá veľká vetva (funkčný/riadiaci paralelizmus) sa v materiáli rozdeľuje podľa „úrovne“, na ktorej paralelizmus využívaš:

Paralelizmus na úrovni inštrukcií (ILP)

Tu ide o to, koľko inštrukcií vieš vykonávať súbežne v rámci jedného vlákna. Materiál sem zaraďuje:

- **superskalárne procesory**,
- **VLIW (Very Long Instruction Word)**,
- **DF (dataflow)** architektúry.

Zjednodušená myšlienka: ILP je silne viazaný na **závislosti** medzi inštrukciami – teda koľko nezávislých operácií vôbec existuje.

Paralelizmus na úrovni vlákien (TLP)

Tu už ide o súbeh viacerých vlákien. Materiál uvádza:

- **SMT (Simultaneous Multithreading)** – viac vlákien v jednom jadre,
- **CMP (Chip Multiprocessor)** – viac jadier na jednom čipe,
- všeobecne **viacjadrové (multicore)** procesory.

Tu je dôležitý posun od závislostí v jednom vlákne k tomu, koľko nezávislej práce vieš robiť paralelne ako viac vlákien.

Paralelizmus na úrovni procesov

Tu sú typicky **MIMD** systémy a materiál rozlišuje:

- **MIMD so zdieľanou pamäťou**,

- **MIMD s distribuovanou pamäťou**.

Toto delenie je dôležité, lebo zásadne ovplyvňuje, ako sa programy komunikujú (zdieľané premenné vs. správy) a aké sú limity škálovania.

---

14) Výkon paralelných architektúr: čo znamená „rýchlejšie“ a ako to meriame

Pri výkone sa typicky používajú dve základné metriky:

- **čas odozvy / čas vykonania** (response time, execution time),
- **priepustnosť** (throughput), teda koľko úloh systém spracuje za jednotku času.

Často sa používa aj porovnávanie typu „zrýchlenie X vzhľadom na Y“, teda napríklad o koľko je nový systém rýchlejší oproti starému alebo o koľko je paralelná verzia rýchlejšia než sekvenčná.

Čas vykonania: reálny čas vs. CPU čas

Materiál rozlišuje:

- **reálny čas (wall-clock time)**: čas, ktorý reálne „odtiká“ od spustenia programu po jeho koniec. Zahŕňa aj režijné náklady systému (čakanie, plánovanie, I/O, blokovania…).
- **CPU čas**: čas, počas ktorého bol program skutočne vykonávaný na CPU (čistý výpočtový čas bez čakania).

Toto rozlíšenie je dôležité, lebo dva programy môžu mať podobný wall-clock čas, ale úplne inú štruktúru režijných nákladov, alebo naopak.

---

15) Benchmarky: ako sa výkon porovnáva v praxi

Výkon sa porovnáva pomocou benchmarkov, teda programov alebo sád programov navrhnutých na meranie.

Materiál uvádza viac typov:

- **kernely** (napr. násobenie matíc) – malé jadrové úlohy reprezentujúce typické výpočty,
- **ukážkové programy** (napr. triedenie),
- **syntetické benchmarky** (napr. Dhrystone),
- **súpravy benchmarkov** (napr. SPEC06, TPC-C).

Osobitne sa spomínajú desktopové benchmarky (napr. SPEC CPU2006: CINT2006 a CFP2006) a serverové benchmarky (napr. SPEC CPU2000, TPC).

Ďalšie bežné benchmarky (mikrobenchmarky a HPC sady)

Materiál uvádza aj často používané mikrobenchmarky:

- pre numerické výpočty: **LAPACK**, **ScaLAPACK**,
- priepustnosť pamäte: **STREAM**,
- „kernel“ benchmarky: **NPB (NAS Parallel Benchmarks)**, **PARKBENCH**, **SPEC**, **Splash**.

Pointa je, že rôzne benchmarky testujú rôzne „slabé miesta“: niektoré trápia CPU, iné pamäť, iné komunikáciu a škálovanie.

---

16) Bežné metriky výkonu: MIPS, FLOPS, SPECRatio

Materiál uvádza klasické metriky:

MIPS

MIPS (milión inštrukcií za sekundu) sa definuje ako:
MIPS = počet inštrukcií / (čas vykonania × 10^6)
Zároveň sa spomína aj **GIPS = 10^3 MIPS**.

Dôležité je chápať, že MIPS samo o sebe nemusí férovo porovnávať odlišné architektúry (odlišné ISA, inštrukčný mix, rôzne CPI), ale ako hrubý ukazovateľ sa stále objavuje v praxi.

FLOPS

Pre numerické výpočty sa používa:
MFLOPS = počet FP operácií / (čas vykonania × 10^6)
a jednotky sa škálujú: **GFLOPS = 10^3 MFLOPS**, **TFLOPS = 10^3 GFLOPS**, atď.

Materiál spomína aj **SPECRatio** ako metriky používané v rámci benchmarkov SPEC (ide o relatívnu metriku oproti referenčnému systému).

---

17) Praktická ukážka: 64× 7-ZIP rebríček (MIPS)

V materiáli je uvedený príklad rebríčka založeného na 7-ZIP benchmarku (64×), ktorý uvádza hodnoty v MIPS pre konkrétne procesory a podmienky (frekvencia a typ chladenia). Príklady uvedené v materiáli:

- AMD Ryzen Threadripper Pro 9995WX (5 840 MHz, kvapalný dusík): **1 454 026 MIPS**

- AMD Epyc 9654 (2 400 MHz, vzduch): **1 122 039 MIPS**
- AMD Ryzen Threadripper 9980X (5 163 MHz, vodné chladenie): **858 426 MIPS**
- Intel Xeon 678X (5 799 MHz, kvapalný dusík): **758 049 MIPS**
- AMD Ryzen Threadripper Pro 9975WX (5 540 MHz, suchý ľad): **536 096 MIPS**

Zmysel tejto ukážky je uvedomiť si, že výsledky benchmarkov závisia nielen od architektúry, ale aj od frekvencie, chladenia, konfigurácie a konkrétnej záťaže.

---

18) Jednotky FLOPS a kontext „dnešného“ výkonu (vrátane príkladov)

Materiál prehľadne uvádza škálu:

- **kiloFLOPS (10^3)**
- **megaFLOPS (10^6)**
- **gigaFLOPS (10^9)** – typicky „dnešný“ CPU (v určitých úlohách)
- **teraFLOPS (10^{12})** – typicky „dnešný výkonný“ GPU (FP32/64 podľa režimu)
- **petaFLOPS (10^{15})**
- **exaFLOPS (10^{18})** – „exascale“ úroveň

V materiáli sú uvedené aj konkrétne príklady (ako ilustrácia rozdielu medzi Rmax a Rpeak a ako predstava mierky):

PetaFLOPS príklady (PERUN – SK):

- PERUN (SK, SAV): **Rmax = 14,04 PFLOPS**, **Rpeak = 18 PFLOPS**
- PERUN (SK, TUKE): **Rmax = 10,21 PFLOPS**, **Rpeak = 11,43 PFLOPS**

Exascale kontext („sme tu“ podľa údajov v materiáli):

- El Capitan (US): **Rmax = 1 809 PFLOPS**, **Rpeak = 2 821,1 PFLOPS**
- Frontier (US): **Rmax = 1 353 PFLOPS**, **Rpeak = 2 055,72 PFLOPS**
- JUPITER Booster (GER): **Rmax = 1 000 PFLOPS**, **Rpeak ≈ 1 226,28 PFLOPS**

Tu je dôležité chápať, že PFLOPS v tisícoch znamená približovanie sa k 1 EFLOPS (pretože 1 EFLOPS = 1000 PFLOPS).

---

19) Špičkový výkon vs. udržateľný výkon: Rpeak a Rmax

Pri superpočítačoch a HPC sa výkon často uvádza dvojako:

Špičkový výkon (Rpeak)

Rpeak je teoreticky najvyšší možný výkon (typicky vo FLOPS), ktorý by systém dosiahol, keby nerobil nič iné než ideálne numerické výpočty. Je to **hrubý hardvérový ukazovateľ**, ktorý však sám o sebe málo hovorí o tom, ako sa systém bude správať pri reálnych aplikáciách.

Udržateľný výkon (Rmax)

Rmax je výkon, ktorý konkrétny program alebo benchmark dosiahne **počas celého behu**. Je to prakticky dôležitejšie číslo, lebo zahŕňa reálne obmedzenia (pamäť, komunikáciu, režijné náklady, neideálnu využiteľnosť).

Preto sa často uvádza aj:
efektivita = Rmax / Rpeak
a typicky platí, že Rpeak býva vyššie (často výrazne) než Rmax.

Ako sa meria udržateľný výkon

Materiál uvádza benchmark **HPL (High Performance LINPACK)** ako meradlo výpočtovej sily systému v operáciách s plávajúcou desatinnou čiarkou (**FP64**) a zároveň pripomína, že sa používa na zostavenie rebríčka superpočítačov v zozname **TOP500**.

---

20) Princípy návrhu počítačov: prečo sa architektúry navrhujú práve takto

Materiál na záver pripája princípy, ktoré sa opakovane používajú pri návrhu (aj paralelných) architektúr.

Využiť paralelizmus

Paralelizmus sa dá využívať na viacerých úrovniach:

- **úroveň systému**: dôraz na škálovateľnosť (pridanie zdrojov má priniesť reálny prínos),
- **úroveň CPU**: prúdové spracovanie (**pipelining**) na zvýšenie priepustnosti,
- **digitálny návrh**: rýchla asociatívna pamäť (**cache rôznych úrovní**) a vysokovýkonná ALU.

Princíp lokality

Výkon nie je len o výpočte, ale aj o tom, či má procesor dáta včas. Preto sa používa **cache** a opiera sa to o princíp lokality:

- **časová (temporálna) lokalita**: to, čo bolo použité nedávno, bude pravdepodobne použité znovu v blízkej budúcnosti,

- **priestorová lokalita**: položky s blízkymi adresami sa zvyknú používať v čase blízko seba.

---

21) Amdahlov zákon: „zamerať sa na bežný prípad“

Amdahlov zákon vysvetľuje, prečo celkové zrýchlenie systému limituje tá časť programu, ktorú nevieš zrýchliť.

Materiál definuje:

- **fraction_enhanced (f)**: podiel celkového času behu programu v pôvodnom systéme, ktorý pripadá na tú časť výpočtu, ktorú vieme urýchliť ($0 \le f \le 1$),
- **speedup_enhanced (S)**: zrýchlenie tejto „vylepšenej“ časti, definované ako pomer času tejto časti pred zlepšením k času tej istej časti po zlepšení ($S > 1$).

Celkové zrýchlenie potom vyjadruje vzťah:
Speedup_total = 1 / ((1 − f) + f / S)

Interpretácia je dôležitejšia než samotný vzorec: ak je (1 − f) veľké, tak aj veľmi veľké S prinesie len obmedzený celkový efekt. Preto sa oplatí optimalizovať najmä tú časť, kde program trávi najviac času.

---

22) Rovnica výkonu procesora a význam CPI

Pri analýze výkonu CPU sa používa klasická rovnica:

CPU time = Instruction count × CPI × Clock cycle time
(alebo ekvivalentne: CPU time = Instruction count × CPI / Clock rate)

Tento zápis je užitočný, lebo ukazuje tri miesta, kde sa dá výkon zlepšovať:

- znížiť počet vykonaných inštrukcií (lepší kód/kompilátor/algoritmus),
- znížiť CPI (lepšia mikroarchitektúra, ILP, menšie čakanie na pamäť),
- znížiť čas cyklu (vyšší takt, ale limitovaný spotrebou a teplom).

Materiál pripomína aj veľmi praktický detail: **rôzne typy inštrukcií majú rôzne CPI**. Preto celkové CPI nie je jedna pevná konštanta, ale typicky vážený priemer podľa inštrukčného mixu programu. Dá sa to zapísať napríklad: **$CPI_{avg} = (\Sigma\ CPI_i \times IC_i) / (\Sigma\ IC_i)$** kde IC_i je počet inštrukcií i-teho typu.

Prednáška 4: Charakteristické vlastnosti paralelizmu

1. Čo sa v tejto prednáške rozumie pod paralelizmom

Táto prednáška sa pozerá na paralelizmus nie iba ako na „viac jadier naraz“, ale ako na všeobecnú vlastnosť výpočtu, pri ktorej sa viac činností vykonáva súbežne. V úvode sa pripomína, že klasické von Neumannove architektúry narážajú na rôzne výkonnostné prekážky, a preto sa výkon zvyšuje nielen rýchlejšou technológiou, ale aj tým, že sa do výpočtu vnáša súbežnosť. Prednáška zároveň spomína, že okrem konvenčných architektúr existujú aj nové smery, napríklad kvantové alebo biopočítačové prístupy, ale jadrom predmetu zostáva paralelizmus v klasických počítačových systémoch. Dôležitá myšlienka je, že rôzne aplikácie narážajú na rôzne limity: dátovo náročné aplikácie limituje pamäťová priepustnosť, serverové aplikácie často sieťová priepustnosť a vedecké výpočty naraz výpočtový výkon aj pamäť. Preto nestačí hovoriť o jednom type paralelizmu – treba rozumieť tomu, kde presne vzniká úzke hrdlo a aký druh súbežnosti ho vie zmierniť. Prednáška tu rozlišuje najmä „časový paralelizmus“, typický pre prúdové spracovanie, a „priestorový paralelizmus“, kde sa problém rieši tým, že sa komponenty fyzicky rozmnožia.

2. Rozparalelnenie výpočtového procesu

Prednáška ukazuje jednoduchý cyklický program, v ktorom sa v každej iterácii vypočíta podiel, potom súčin a nakoniec akumulovaná hodnota. Zmysel príkladu je ukázať, že ten istý výpočtový problém sa môže rozparalelniť rôznymi spôsobmi. V architektúre SIMD by sa rovnaká operácia vykonávala nad viacerými prvkami dát naraz. V architektúre MIMD by sa zasa jednotlivé časti výpočtu alebo iterácie mohli rozdeľovať medzi samostatné procesory alebo vlákna, ktoré si riadia vlastný tok inštrukcií. Prednáška tým hneď na začiatku naznačuje dôležitú vec: paralelizmus nie je jediná technika, ale celý súbor možností, ako rozdeliť výpočet. Súbežné vykonávanie sa potom v širšom zmysle prejavuje ako multiprogramovanie, viacprocesorové spracovanie a viacpočítačové spracovanie. Z toho ďalej vyrastajú konkrétne formy paralelizácie úloh, programov, inštrukcií a dát.

Prednáška zároveň vymenúva veľa foriem, v ktorých sa paralelizácia môže objaviť. Patria sem dopredné prehľadávanie, prúdové spracovanie, vektorizácia, údajový paralelizmus, segmentovanie, vkladanie, prekrývanie, rozmnožovanie, prideľovanie času, prideľovanie priestoru, viacúrovňové spracovanie úloh, multivláknové spracovanie aj distribuované výpočty. Zmysel tohto zoznamu nie je naučiť sa izolované pojmy bez súvislosti, ale pochopiť, že paralelizmus sa objavuje na viacerých úrovniach a v rôznych formách organizácie systému.

3. Od čoho závisí, či sa program dá vykonávať paralelne

Prednáška potom prechádza k najdôležitejšej otázke: kedy sa vlastne môžu inštrukcie alebo časti programu vykonávať súbežne bez toho, aby sa pokazil výsledok. Stupeň paralelizácie programu je podľa nej určovaný troma základnými triedami závislostí: údajovými

závislosťami, štrukturálnymi alebo zdrojovými závislosťami a riadiacimi závislosťami. To je veľmi dôležitý rámec, pretože prakticky všetko ďalšie v prednáške sa odvíja práve od toho, ako tieto závislosti rozpoznať a ako ich obísť, minimalizovať alebo efektívne zvládnuť.

4. Údajové závislosti: RAW, WAR, WAW

Najväčšia pozornosť sa venuje údajovým závislostiam. Základné tri typy sú RAW, WAR a WAW. RAW znamená „read after write“ – neskoršia operácia chce niečo čítať až po tom, čo to skoršia operácia zapísala. To je klasická pravá závislosť a z pohľadu korektnosti programu býva najkritickejšia. WAR znamená „write after read“ – neskoršia operácia zapisuje na miesto, ktoré skoršia operácia ešte potrebuje prečítať. WAW znamená „write after write“ – dve operácie zapisujú na to isté miesto a musí sa zachovať správne poradie zápisov. Prednáška okrem toho spomína aj osobitný prípad V/V závislosti, ale v ďalšej analýze pracuje hlavne s trojicou RAW, WAR a WAW.

Tieto závislosti sú mimoriadne dôležité najmä pri prúdovom spracovaní inštrukcií, teda v pipeline. Prednáška vysvetľuje, že pri jednoduchom pipeline spracovaní typicky vzniká najmä hazard RAW, lebo nasledujúca inštrukcia môže potrebovať výsledok ešte skôr, ako bol korektne zapísaný do registra alebo pamäte. WAR a WAW sa pri jednoduchších pipeline objavujú len v osobitných prípadoch, ale pri zložitejších architektúrach už zohrávajú reálnu úlohu. To je dôležitý rozdiel: nie každá závislosť je rovnako častá v každej architektúre, ale pri analýze paralelizmu sa všetky sledujú.

5. Ako sa odstraňuje RAW hazard

Prednáška veľmi prakticky ukazuje, že RAW hazard sa dá odstrániť buď softvérovo, alebo hardvérovo. Softvérový prístup je jednoduchší na pochopenie: vloží sa prázdna inštrukcia, teda oneskorenie, alebo sa preusporiada poradie inštrukcií tak, aby medzi producentom výsledku a jeho použitím vznikol dostatočný odstup. Hardvérový prístup je výkonnejší: pipeline sa na chvíľu zablokuje a vygenerujú sa prázdne cykly, alebo sa použije dopredné generovanie výsledku, teda forwarding, pri ktorom sa výsledok predchádzajúcej inštrukcie neposiela najprv do registra a až potom späť, ale rovno sa privedie ako operand nasledujúcej inštrukcii. Prednáška tým ukazuje zásadný kompromis: jednoduchší softvér býva pomalší, sofistikovanejší hardvér vie hazardy zmierniť, ale za cenu vyššej zložitosti architektúry.

6. Riadiace závislosti a hazardy vetvenia

Druhou veľkou skupinou sú riadiace závislosti. Tie vznikajú pri skokoch a vetveniach, teda vtedy, keď ďalšia vykonávaná inštrukcia závisí od výsledku podmienky. Prednáška ukazuje rozdiel medzi nezávislým riadením cyklov a závislým riadením cyklov. Pri nezávislom riadení je rozhodnutie v každej iterácii nezávislé od predchádzajúcej iterácie. Pri závislom riadení je aktuálna iterácia závislá od výsledku predchádzajúcej. To má zásadný vplyv na možnosti paralelizácie, pretože závislé vetvenie obmedzuje, koľko práce sa dá pripraviť dopredu. V pipeline sa takýto problém prejaví ako riadiaci hazard: kým sa neurčí cieľ vetvenia, ďalšie stupne pipeline môžu čakať alebo vykonávať špekulatívnu prácu, ktorá sa neskôr možno

zahodí. Prednáška to ilustruje aj príkladom, kde odstránenie hazardu vetvenia znamená niekoľko prázdnych strojových cyklov.

7. Zdrojové závislosti

Tretia skupina sú zdrojové alebo štrukturálne závislosti. Tie nevznikajú preto, že by si inštrukcie logicky prekážali na dátach, ale preto, že chcú v rovnakom čase použiť tú istú funkčnú jednotku, register alebo iný zdroj. Ak má napríklad architektúra len jednu sčítačku a dve operácie ju chcú použiť naraz, obe sa jednoducho súčasne vykonať nedajú. Prednáška preto uvádza dve základné možnosti riešenia: buď sa vykonanie jednej operácie oneskorí vložením prázdneho cyklu, alebo sa architektúra vybaví viacerými zdrojmi, teda viacerými funkčnými jednotkami, oddelenými registrami a podobne. Toto je veľmi dôležitá myšlienka, lebo ukazuje rozdiel medzi „program by sa paralelizovať dal“ a „hardvér to naozaj dovolí“.

8. Graf závislostí

Na systematickú analýzu paralelizácie sa používa graf závislostí. Ide o orientovaný graf, v ktorom uzly predstavujú programové entity – inštrukcie, príkazy alebo procesy – a hrany vyjadrujú typ závislosti medzi nimi. Prednáška ukazuje príklad s inštrukciami typu load, add, move a store a potom z neho zostavuje graf, v ktorom sú osobitne vyznačené RAW, WAR a WAW väzby. Zmysel takého grafu je veľmi praktický: okamžite vidno, ktoré operácie musia zostať v poradí a ktoré by sa teoreticky dali vykonať súbežne. Práve tento grafický pohľad je základom pre ďalšie rozhodovanie o plánovaní výpočtu a o návrhu architektúry.

9. Bernsteinove podmienky

Prednáška potom formalizuje možnosť paralelného vykonania pomocou Bernsteinových podmienok. Dva procesy alebo inštrukcie PiP_iPi a PjP_jPj sa môžu vykonávať paralelne len vtedy, keď neexistuje konflikt medzi ich vstupmi a výstupmi. Formálne to znamená, že musí platiť:
Oi∩Ij=ØO_i \cap I_j = \varnothingOi∩Ij=Ø,
Ii∩Oj=ØI_i \cap O_j = \varnothingIi∩Oj=Ø,
Oi∩Oj=ØO_i \cap O_j = \varnothingOi∩Oj=Ø.

Prvá podmienka zodpovedá RAW konfliktu, druhá WAR konfliktu a tretia WAW konfliktu. Prednáška zároveň zdôrazňuje, že riadiace a zdrojové závislosti sa analyzujú osobitne, teda samotné Bernsteinove podmienky nestačia na úplné rozhodnutie o vykonateľnosti na konkrétnom hardvéri. To je veľmi dôležité: Bernstein vie povedať, či je niečo teoreticky nezávislé z pohľadu dát, ale ešte nehovorí, či na to má stroj dosť sčítačiek, násobičiek alebo pamäťových portov.

Prednáška pridáva aj konkrétny príklad s piatimi príkazmi P1P_1P1 až P5P_5P5, kde sa ukáže, že sekvenčné vykonanie zaberie päť krokov, ale pri vhodnom rozdelení a s dostatočnými zdrojmi sa dá znížiť na tri kroky. To je presne pointa Bernsteinových podmienok v praxi: nehovoria len „áno/nie“, ale pomáhajú nájsť, kde sa v programe skutočne skrýva použiteľný paralelizmus.

10. Návrh štruktúrnej organizácie funkčných jednotiek

V nadväznosti na závislostné grafy prednáška ukazuje, že analýza paralelizmu neslúži len na optimalizáciu programu, ale aj na návrh samotných funkčných jednotiek architektúry. Ak z grafu vyplýva, že sa často súbežne vyskytujú dve sčítania a jedno násobenie, architekt môže navrhnúť takú organizáciu funkčných jednotiek, ktorá to podporí. Ak naopak v programe dominujú lineárne závislosti, pridanie veľkého počtu jednotiek neprinesie primeraný efekt. Prednáška tu teda prepája softvérový pohľad s hardvérovým: správna architektúra má zodpovedať skutočnému profilu paralelizmu v úlohách, ktoré má riešiť.

11. Hardvérový a softvérový paralelizmus

Veľmi dôležité je rozlíšenie medzi hardvérovým a softvérovým paralelizmom. Hardvérový paralelizmus je daný tým, aké zdroje architektúra reálne ponúka. Prednáška ho charakterizuje najmä počtom inštrukcií, ktoré sa dajú vykonať za jeden inštrukčný cyklus. Podľa toho rozlišuje jednooperačné procesory, viacoperačné procesory a viacprúdové procesory. Jednooperačný procesor vykoná jednu inštrukciu za jeden alebo niekoľko strojových cyklov. Viacoperačný procesor vie vykonať aspoň dve inštrukcie za jeden strojový cyklus. Viacprúdový procesor už predstavuje multiprocesorový model.

Softvérový paralelizmus je naopak daný samotným programom. Závisí od algoritmu, štýlu programovania a optimalizujúceho kompilátora. Prednáška hovorí, že sa prejavuje v profile programu alebo vo vývojovom grafe programu. To znamená, že aj na veľmi silnom hardvéri môže bežať program zle, ak je napísaný sekvenčne a plný závislostí. Naopak, dobre navrhnutý algoritmus môže odhaliť paralelizmus, ktorý by sa inak nevyužil. V tom je zásadný rozdiel: hardvér hovorí, čo je možné, softvér hovorí, čo sa naozaj využije.

Prednáška uvádza aj jednoduché ilustračné pomery. Pri jednom príklade vychádza priemerný softvérový paralelizmus PSWP=Ni/NSC=8/3=2,67PSWP = N_i / N_{SC} = 8/3 = 2{,}67PSWP=Ni/NSC=8/3=2,67 inštrukcie na strojový cyklus. Na slabšom hardvéri s jednou load/store jednotkou a jednou ALJ vyjde hardvérový paralelizmus iba 9/7=1,289/7 = 1{,}289/7=1,28 inštrukcie na cyklus. Na bohatšom multiprocesorovom usporiadaní s dvoma load/store jednotkami a dvoma ALJ narastie na 11/6=1,8311/6 = 1{,}8311/6=1,83. Tieto čísla pekne ukazujú, že softvérový paralelizmus môže byť väčší než to, čo vie hardvér reálne využiť.

12. Rozklad programu: granularita a komunikačná latentnosť

Rozklad programu na paralelné segmenty podľa prednášky vždy závisí od dvoch základných atribútov: od granularity a od komunikačnej latentnosti. Granularita je miera množstva výpočtu v segmente. Čím menšie segmenty, tým jemnejšia granularita a tým viac potenciálneho paralelizmu. Lenže jemná granularita zároveň znamená viac komunikácie medzi segmentmi. Komunikačná latentnosť je časová miera tejto komunikačnej réžie medzi procesormi alebo komponentmi systému. To znamená, že pri rozklade programu treba neustále hľadať kompromis: príliš jemné delenie síce zvýši počet paralelných vetiev, ale môže spôsobiť, že väčšinu času stratíme koordináciou a prenosmi dát.

Prednáška navyše zdôrazňuje, že na rozklade programu sa môže podieľať viac „vrstiev“: tvorca algoritmu, programátor, kompilátor aj operačný systém. To je dôležité, pretože paralelizácia nevzniká len na jednej úrovni. Niekedy sa paralelné vetvy navrhnú už v algoritme, niekedy ich odhalí kompilátor a niekedy výsledné plánovanie ovplyvní až runtime alebo OS.

13. Úrovne granularity

Prednáška rozdeľuje granularitu na jemnú, strednú a hrubú, pričom hovorí o piatich konkrétnych úrovniach paralelizácie. Pri jemnej granularite ide o inštrukčnú úroveň a o úroveň operácie cyklu. Inštrukčná úroveň má typický rozmer 2 až 1000 inštrukcií, pričom priemerne ide približne o sedem inštrukcií. Úroveň operácie cyklu sa viaže na nerekurzívne cykly a rozvinuté slučky, ktoré môžu obsahovať aj stovky inštrukcií. Pri strednej granularite ide o úroveň procedúr a podprogramov. Procedúry, subrutiny, úlohy a korutiny majú typicky rádovo tisíce inštrukcií. Podprogramová úroveň sa využíva napríklad v režimoch SPMD a MPMD alebo pri multiprogramovaní. Hrubá granularita sa týka nezávislých programov a prác, kde segment môže mať desaťtisíce a viac inštrukcií. Typicky sem patrí multicomputing a veľké MIMD prostredia. Pointa je jednoduchá: čím hrubšie segmenty, tým menej komunikácie, ale aj menej jemného rozdelenia práce.

14. Kľúčové problémy pri rozklade programu

Pri rozklade programu treba podľa prednášky riešiť tri hlavné problémy: optimalizovať rozmer granularity, minimalizovať komunikačnú latentnosť a zabrániť uviaznutiu programu. To znamená, že nestačí len „rozrezať program na kúsky“. Treba nájsť také rozdelenie, pri ktorom sa paralelné vetvy navzájom neblokujú, neprenášajú priveľa dát a pritom je ich dostatok na využitie dostupných procesorov. Prednáška to ilustruje aj príkladom programu s postupnými násobeniami, kde sa ukazuje, že po zhustení granularity vznikajú iné komunikačné pomery než pri jemnozrnnom rozklade.

15. Grafy paralelného programu (GPP)

Na vyjadrenie rozkladu sa používajú grafy paralelného programu. V uzloch grafu sa zapisuje meno segmentu a rozmer jeho granularity. Na hranách sa uvádza prenášaná premenná a komunikačný čas medzi zdrojovým a cieľovým uzlom. Prednáška pri konkrétnom príklade rozlišuje uzly 1 až 6 ako pamäťové referencie, kde samotné adresovanie trvá 1 strojový cyklus a výber z pamäte má komunikačný čas 6 cyklov. Uzly 7 až 17 reprezentujú operácie CPU, kde výpočtová časť zaberá 2 cykly. Niektoré výstupné hrany majú komunikačný čas 4 cykly a pri vnútorných väzbách v hrubozrnných segmentoch iba 3 cykly. To je veľmi dôležité, pretože práve tieto časy rozhodujú, či sa jemnejší rozklad oplatí alebo nie.

Prednáška potom vysvetľuje kompakciu, teda zhustenie granularity. Pri jemnozrnnom paralelizme je viac medziprocesorových komunikácií, ale jednotlivé uzly vykonávajú menšiu prácu. Pri hrubozrnnom paralelizme je uzlov menej, trvajú dlhšie, ale výrazne sa obmedzí komunikácia medzi nimi. Preto kompakcia granularity v praxi hľadá kompromis medzi stupňom paralelizmu a komunikačnou réžiou. Veľmi dôležitá myšlienka je, že operácie

zlúčené do jedného hrubšieho uzla zväčša vykonáva jeden procesor, a tak sa komunikácia vnútri takého uzla stáva zanedbateľnou v porovnaní s komunikáciou medzi uzlami.

16. Plánovanie spracovania GPP

Prednáška uvádza štvorstupňový postup plánovania. Najprv sa navrhne jemnozrnný GPP podľa údajových závislostí. Potom sa naplánuje jeho paralelné vykonanie. Následne sa vykoná kompakcia GPP, aby sa zväčšila granularita a znížila komunikačná réžia. Nakoniec sa opäť naplánuje vykonanie už zhusteného, hrubozrnného grafu. Toto poradie je dôležité: prednáška nechce, aby sa granularita volila „od oka“, ale aby sa najprv odhalil maximálny paralelizmus a až potom sa rozumne zlučoval do väčších segmentov.

17. Škálovateľnosť paralelných systémov

Druhá veľká časť prednášky je o škálovateľnosti. Zavádzajú sa tri základné pojmy. Pracovná záťaž PPP je množstvo výpočtov vykonaných v paralelnom prostredí. Rozmer problému sss kvantifikuje veľkosť úlohy. Strojový rozmer nnn je počet procesných elementov, teda veľkosť paralelného stroja. Škálovateľnosť sa potom chápe ako schopnosť systému zachovať si účinnosť aj pri raste počtu procesorov. Inými slovami, ak pridávam procesory a účinnosť ostáva približne konštantná, systém je škálovateľný. Ak účinnosť prudko padá, architektúra alebo algoritmus sa neškáluje dobre.

Prednáška zároveň zdôrazňuje, že škálovateľnosť nie je len vlastnosť hardvéru, ale vlastnosť dvojice algoritmus–architektúra. Do hry vstupuje determinizmus algoritmu, granularita výpočtu, profil paralelizmu, komunikačná réžia, uniformita operácií a vplyv údajových štruktúr. Napríklad statické algoritmy sa hodia skôr pre prúdové architektúry alebo SIMD, kým dynamické algoritmy sú prirodzenejšie pre MIMD. Aj to je dôležitá myšlienka: neexistuje univerzálne „najlepšia“ paralelná architektúra, všetko závisí od charakteru úlohy.

18. Vzory pracovnej záťaže

Prednáška rozlišuje viaceré typické vzory rastu pracovnej záťaže vzhľadom na rast stroja. Konštantná pracovná záťaž zodpovedá modelu fixed-load. Lineárne rastúca záťaž zodpovedá modelu fixed-time, kde chceme pri väčšom stroji riešiť väčší problém v tom istom čase. Sublineárny rast predstavuje prechodové prípady. Exponenciálny rast zodpovedá modelu fixed-memory, kde problém rastie podľa toho, koľko pamäte poskytuje väčší stroj. Tieto modely sú dôležité, lebo hovoria, čo vlastne od škálovania očakávame. Pri jednom type úloh sa snažíme skrátiť čas fixnej úlohy, pri inom chceme pri rovnakom čase spracovať väčší problém a pri ďalšom nás limituje hlavne pamäť.

19. Izoúčinnosť a asymptotické zrýchlenie

Na formálnu analýzu škálovateľnosti prednáška používa izoúčinnosť. Účinnosť je definovaná vzťahom

E(s,n)=P(s)P(s)+h(s,n),E(s,n)=\frac{P(s)}{P(s)+h(s,n)},E(s,n)=P(s)+h(s,n)P(s),

kde P(s)P(s)P(s) je pracovná záťaž a h(s,n)h(s,n)h(s,n) je súhrnná komunikačná réžia. Z toho vyplýva, že ak chceme pri rastúcom počte procesorov zachovať rovnakú účinnosť, musí pracovná záťaž rásť primerane rýchlo ako komunikačná réžia. Po úprave dostávame vzťah

P(s)=E1−E h(s,n)=K·h(s,n),P(s)=\frac{E}{1-E}\,h(s,n)=K\cdot h(s,n),P(s)=1−EEh(s,n)=K·h(s,n),

kde KKK je konštanta pre zvolenú účinnosť. To je presne význam izoúčinnosti: hovorí, ako rýchlo musí rásť problém, aby sa oplatilo pridávať ďalšie procesory.

Asymptotické zrýchlenie sa v prednáške definuje ako

S(s,n)=T(s,1)T(s,n)+h(s,n),S(s,n)=\frac{T(s,1)}{T(s,n)+h(s,n)},S(s,n)=T(s,n)+h(s,n)T(s,1),

teda pomer sekvenčného času k paralelnému času vrátane komunikačnej réžie. V ideálnom prostredí PRAM, kde sa komunikácia zanedbá, sa tento vzťah zjednoduší na

SI(s,n)=T(s,1)TI(s,n).S_I(s,n)=\frac{T(s,1)}{T_I(s,n)}.SI(s,n)=TI(s,n)T(s,1).

Prednáška tým zdôrazňuje, že teoretické zrýchlenie bez komunikácie a reálne zrýchlenie na skutočnom stroji sú dve odlišné veci. Skutočné prostredia majú réžiu, a preto sa k ideálu iba približujú.

20. Výkonnosť paralelného prostredia

Prednáška potom zavádza viacero spôsobov, ako merať výkonnosť paralelného prostredia na sade programov. Aritmetická priemerná výkonnosť RAR_ARA je obyčajný priemer rýchlostí jednotlivých programov. Ak majú programy rôznu dôležitosť alebo výskyt, používa sa vážená aritmetická priemerná výkonnosť RA∗R_A^*RA∗, kde sa jednotlivé výkony násobia váhami fif_ifi a platí ∑fi=1\sum f_i = 1∑fi=1. Prednáška však upozorňuje, že v praxi sa často lepšie pracuje s časmi než s rýchlosťami, a preto sa používa harmonická priemerná výkonnosť RHR_HRH a jej vážená verzia RH∗R_H^*RH∗. Harmonický priemer lepšie vystihuje situáciu, keď viaceré benchmarky reprezentujú reálne používané programy a jednoduchšie sa merajú ich časy vykonania než ich okamžité rýchlosti.

Okrem toho sa zavádza harmonické stredné zrýchlenie výkonnosti, ktoré porovnáva výkon paralelného prostredia s uniprocesorovým prostredím cez viac vykonávacích módov a ich váhy. Pointa je, že pri hodnotení paralelného systému nestačí jedno číslo namerané na jednom programe. Rozumné hodnotenie musí zachytiť správanie na celej triede úloh.

21. Metriky výkonnosti a ich význam

V závere prednáška ešte raz zhrňuje základné praktické metriky. Ak T(n)T(n)T(n) označuje čas vykonania na nnn-procesorovom systéme, potom činiteľ zrýchlenia je

S(n)=T(1)T(n).S(n)=\frac{T(1)}{T(n)}.S(n)=T(n)T(1).

Účinnosť je

E(n)=S(n)n.E(n)=\frac{S(n)}{n}.E(n)=nS(n).

Redundancia R(n)R(n)R(n) vyjadruje, ako rastie celkový počet operácií alebo práce pri paralelnom spracovaní oproti sekvenčnému. Využiteľnosť U(n)U(n)U(n) vyjadruje mieru využitia zdrojov pri vykonávaní programu. Kvalita je v prednáške chápaná ako súhrnnejší ukazovateľ využitia a primeranosti použitého paralelného riešenia. Zmysel týchto metrík je odlíšiť dve veci: rýchly výsledok a efektívne využitý systém nie sú to isté. Môžem mať zrýchlenie, ale pritom zle využívať procesory, alebo môžem mať vysoký počet procesorov a nízku kvalitu riešenia, ak väčšinu času čakajú.

22. Amdahlov zákon

Amdahlov zákon v prednáške vystupuje ako základné obmedzenie paralelizácie. Myšlienka je jednoduchá: ak časť programu ostáva sekvenčná, táto časť začne pri veľkom počte procesorov dominovať a obmedzí celkové zrýchlenie. Pri modeli, kde sekvenčná časť zaberá podiel α\alphaα, vychádza horná hranica zrýchlenia pri neobmedzenom raste počtu procesorov ako

Smax⁡=1α.S_{\max}=\frac{1}{\alpha}.Smax=α1.

To znamená, že ak je 10 % programu sekvenčných, maximálne zrýchlenie je 10 bez ohľadu na to, či použijeme 4, 16 alebo 1024 procesorov. Prednáška výslovne zdôrazňuje, že Amdahl rieši problém s konštantnou veľkosťou: chceme ten istý problém vyriešiť čo najrýchlejšie. Preto je jeho záver dôležitý, ale platí len pre tento konkrétny pohľad na škálovanie.

23. Gustafsonov zákon

Gustafsonov zákon mení perspektívu. Namiesto toho, aby pri väčšom počte procesorov držal konštantnú veľkosť problému, drží konštantný čas behu. Predpokladá, že pri väčšom stroji budeme riešiť väčší alebo presnejší problém. Ak α∗\alpha^*α∗ označuje sekvenčnú časť programu vykonanú na paralelnom stroji, zrýchlenie sa zapíše ako

S=N+(1−N)α∗,S = N + (1-N)\alpha^*,S=N+(1−N)α∗,

čo je ekvivalent tvaru uvádzaného na slidoch. Podstatný záver je, že zrýchlenie tu nie je zhora tak striktne obmedzené ako pri Amdahlovi. Ak problém rastie s veľkosťou stroja, môžeme dosahovať stále väčší úžitok z ďalších procesorov. Prednáška výslovne hovorí, že Gustafson Amdahla nepopiera – len odpovedá na inú otázku. Amdahl rieši fixný problém, Gustafson fixný čas.

Veľmi názorná je analógia s autom medzi bodmi A a B vzdialenými 60 km. Amdahlov pohľad hovorí, že ak auto za prvú hodinu prešlo 30 km, priemerná rýchlosť za celú 60 km cestu už nikdy neprekročí 60 km/h. Gustafsonov pohľad hovorí, že ak dovolíme, aby cesta bola dlhšia, auto môže priemernú rýchlosť zvyšovať takmer ľubovoľne – stačí, aby po počiatočnej pomalej časti pokračovalo dosť dlho dostatočne vysokou rýchlosťou. Táto analógia veľmi pekne vysvetľuje rozdiel medzi „zrýchliť fixnú úlohu“ a „spracovať väčšiu úlohu v rovnakom čase“.

24. Profil paralelizmu programu

Úplný záver prednášky patrí profilu paralelizmu programu. Stupeň paralelizmu SP(ti)SP(t_i)SP(ti) vyjadruje, koľko procesorov sa v danom časovom intervale podieľa na vykonávaní programu. Profil paralelizmu je potom časový diagram tejto funkcie počas celého behu programu. Prednáška výslovne uvádza, že ho ovplyvňuje štruktúra algoritmu, optimalizácia programu, využitie zdrojov a konkrétne podmienky behu. To je veľmi dôležité: profil paralelizmu nie je len vlastnosť zdrojového kódu, ale výsledok interakcie programu, prekladu aj architektúry.

Priemerný paralelizmus sa definuje ako časovo vážený priemer stupňa paralelizmu:

P=∑i=1mi Δti∑i=1mΔti,P=\frac{\sum_{i=1}^{m} i\,\Delta t_i}{\sum_{i=1}^{m}\Delta t_i},P=∑i=1mΔti ∑i=1miΔti,

kde Δti\Delta t_iΔti je celkový čas, počas ktorého bol stupeň paralelizmu rovný iii, a mmm je maximálny stupeň paralelizmu. Prednáška zároveň definuje vykonanú prácu

W=∑i=1mwi=∑i=1mr i Δti,W=\sum_{i=1}^{m} w_i=\sum_{i=1}^{m} r\,i\,\Delta t_i,W=i=1∑mwi=i=1∑m riΔti,

kde rrr je operačná rýchlosť jedného procesora. Z týchto vzťahov sa potom odvodzuje asymptotické zrýchlenie výpočtového procesu pri neobmedzenom počte procesorov. V ideálnom prípade vyjde

S∞=P,S_\infty = P,S∞=P,

teda asymptotické zrýchlenie sa rovná priemernému paralelizmu programu. V reálnom prípade však platí iba

S∞≤P,S_\infty \le P,S∞≤P,

pretože do hry vstupuje komunikačná a systémová réžia. To je jeden z najdôležitejších záverov celej prednášky: priemerný paralelizmus hovorí, koľko súbežnosti program v sebe skrýva, ale skutočné zrýchlenie je vždy menšie alebo nanajvýš rovné tomuto potenciálu.

Prednáška 5: Prúdové spracovanie, stavový graf inicializácií, vektorizácia

1. Podstata prúdového spracovania

Prúdové spracovanie, teda zreťazenie, je všeobecný princíp spracovania informácií, pri ktorom sa výpočtový proces nerozoberá ako jedna nedeliteľná činnosť, ale rozkladá sa na viac oddelených krokov. Tieto kroky sa realizujú samostatnými funkčnými modulmi, nazývanými stupne alebo segmenty zreťazenia. Kľúčová myšlienka je v tom, že jednotlivé

kroky sa nevykonávajú striktne po sebe pre jednu úlohu a až potom pre ďalšiu, ale prekrývajú sa. Kým prvá úloha je už v druhom alebo treťom stupni, ďalšia úloha môže medzitým vstúpiť do prvého stupňa. Výsledkom nie je skrátenie „vnútornej“ logiky samotnej operácie, ale zvýšenie počtu dokončených operácií za jednotku času.

Preto je dôležité rozlišovať dve veci: **latenciu jednej úlohy** a **priepustnosť celého systému**. Pri zreťazení sa často stáva, že spracovanie jednej jedinej úlohy netrvá kratšie než pri sekvenčnom riešení, dokonca môže trvať aj dlhšie, lebo medzi stupňami sú vložené registračné alebo synchronizačné mechanizmy. Zisk sa prejaví až pri dlhšom prúde úloh, keď sa stupne pipeline zaplnia a systém začne produkovať výsledky pravidelne. Vtedy sa zrýchlenie približuje počtu stupňov zreťazenia, nie však absolútne, ale ako teoretická horná hranica pri ideálnych podmienkach.

2. Podmienky, aby sa prúdové spracovanie vôbec oplatilo

Aby sa dalo hovoriť o prúdovom spracovaní, výpočtový proces musí byť rozložiteľný na viac fáz. Pri inštrukčnom spracovaní to znamená, že inštrukčný cyklus sa rozdelí na niekoľko krokov, ktoré sa môžu vykonávať konkurenčne, teda prekrývane s inými krokmi predchádzajúcich alebo nasledujúcich inštrukcií. Každá takáto fáza pritom trvá jeden alebo viac strojových cyklov a na úrovni obvodov jej zodpovedá konkrétny segment prúdovej funkčnej jednotky.

Zmysel prúdového spracovania teda nespočíva v tom, že sa jedna operácia „magicky“ zrýchli, ale v tom, že sa viac operácií vhodne rozloží a ich jednotlivé fázy sa časovo prekryjú. Je to typický príklad časového paralelizmu. Ak by sa nedali prekryť alebo by medzi nimi boli príliš silné závislosti, pipeline by sa často zastavovala a jej prínos by sa výrazne znížil. Preto sa zreťazenie najlepšie uplatní tam, kde sa opakujú rovnaké alebo podobné operácie nad dlhou sériou údajov.

3. Sekvenčné a prúdové spracovanie – základný rozdiel

Pri sekvenčnom spracovaní funkčná jednotka dokončí jednu úlohu ako celok a až potom začne ďalšiu. Ak máme n úloh a každá trvá určitý čas, celkový čas narastá lineárne s počtom úloh. Pri prúdovom spracovaní sa tá istá logika rozdelí na k stupňov. Po počiatočnom „naplnení“ pipeline sa potom výsledky začnú objavovať pravidelne po takte zreťazenia. To znamená, že prvá úloha čaká, kým prejde všetkými stupňami, ale ďalšie úlohy už využívajú to, že predchádzajúca úloha uvoľnila prvý stupeň.

Z tohto pohľadu je veľmi dôležité pochopiť, že zreťazenie zvyšuje hlavne **priepustnosť**, nie nutne dobu vybavenia jednej konkrétnej úlohy. Ak máme dlhý sled operácií, pipeline pracuje efektívne. Ak máme iba veľmi krátku sériu operácií, zapĺňanie a vyprázdňovanie pipeline môže zisk výrazne zmenšiť. Aj preto sa v teórii uvádza, že pri veľkom počte úloh sa zrýchlenie asymptoticky blíži počtu stupňov k. V praxi však túto hranicu limituje nerovnaká dĺžka stupňov, oneskorenie registrov a konflikty v pipeline.

4. Typické stupne inštrukčného zreťazenia

Pri procesoroch sa prúdové spracovanie najčastejšie vysvetľuje na inštrukčnom cykle. Prednáška uvádza základné štyri stupne. **Fetch (F)** načíta inštrukciu z pamäte. **Decode (D)** ju dekóduje, teda určí, čo má procesor vykonať. **Execute (E)** vykoná definovanú operáciu. **Store alebo Write Back (S/WB)** zapíše výsledok späť do registra alebo pamäte. Takéto členenie je základom klasických RISC pipeline.

Podstatné je, že kým jedna inštrukcia sa dekóduje, ďalšia sa už môže načítavať a ešte iná sa môže vykonávať. Tak vzniká prekrývanie. V časovom diagrame preto nevidíme jednu inštrukciu od začiatku po koniec izolovane, ale viac inštrukcií naraz v rôznych stupňoch spracovania. Táto predstava je úplne kľúčová pre pochopenie pipeline v procesoroch. Bez nej sa ľahko zamieňa sekvenčná logika programu s fyzickým spôsobom, akým ju vykonáva hardvér.

5. Doba vykonania prúdu inštrukcií

Prednáška pri dobe vykonania zavádza niekoľko parametrov. Perióda strojového cyklu je TSCT_{SC}TSC. Počet fáz dekomponovanej inštrukcie je kkk. Počet spracúvaných inštrukcií je nnn. Pri rozšírení na širšie prúdy sa uvažuje aj parameter mmm, teda šírka prúdu, čiže koľko inštrukcií sa môže začínať alebo spracúvať súčasne. Pre bežné skalárne zreťazenie platí m=p=1m = p = 1m=p=1, pri superskalárnom zreťazení už platí m>1m > 1m>1 a systém vie do pipeline zavádzať viac než jednu inštrukciu za takt.

Z praktického hľadiska je dôležité, že celkový čas prúdu inštrukcií nepozostáva len z čistého „počtu inštrukcií krát čas jednej inštrukcie“. Najprv treba pipeline zaplniť, potom určitý čas beží v ustálenom režime a na konci ju treba „vyprázdniť“. Preto sa celkový čas prúdového spracovania skladá z úvodnej fázy plnenia a následnej fázy, keď už výsledky prichádzajú pravidelne. Toto je presne dôvod, prečo pipeline najviac vyhovuje dlhým prúdom operácií.

6. Zrýchlenie pipeline a jeho skutočný význam

Prednáška uvádza, že efektívnosť, teda zrýchlenie prúdového spracovania, sa pri veľkom počte úloh blíži počtu stupňov kkk. To je teoretický ideál. Význam tejto vety je však potrebné chápať správne. Neznamená to, že vždy dostaneme presne k-násobný výkon. Znamená to, že ak sú stupne vyvážené, ak medzi nimi nie sú konflikty a ak dokážeme stále dodávať nové vstupy, pipeline môže produkovať výsledky tempom približne jeden výsledok za takt, a tým sa dlhodobá priepustnosť blíži ideálu.

V praxi sa však objavujú tri prirodzené obmedzenia. Po prvé, pôvodný proces sa nedá vždy rozdeliť na presne rovnako dlhé fázy. Po druhé, medzi stupňami sú registračné a synchronizačné oneskorenia. Po tretie, pipeline musí zostať plná, čo kladie vysoké nároky na prísun dát aj na absenciu konfliktov. Preto je pipeline veľmi silný, ale nie bezpodmienečný mechanizmus zrýchlenia. Ak sa stupne často zastavujú, pipeline síce stále môže byť užitočná, ale jej skutočný prínos je nižší, než by vyplývalo z jednoduchého počtu stupňov.

7. Kde sa prúdové spracovanie používa

Prednáška zdôrazňuje, že zreťazenie sa neuplatňuje iba na úrovni inštrukcií. Používa sa aj pri **zreťazení operácií**, napríklad v aritmeticko-logických jednotkách, najmä vo floating-point jednotkách. Ďalej sa využíva pri **zreťazení inštrukcií**, čo je typické pre procesory s RISC architektúrou. A napokon aj pri **zreťazení procesov a prístupu do pamäte**, teda na systémovej úrovni. To je veľmi dôležité, lebo ukazuje, že pipeline nie je len technika „vnútri CPU“, ale všeobecný princíp organizácie výpočtu.

8. Syntéza prúdového systému

Pri návrhu prúdového systému nestačí len povedať, že rozdelíme proces na viac krokov. Prednáška rozlišuje tri základné úlohy syntézy. Prvou je **analýza procesu zreťazenia**. To znamená rozložiť úlohu na čiastkové časti, určiť počet a štruktúru stupňov a zistiť, aký čas údajmi v týchto stupňoch preteká. Druhou je **stratégia riadenia procesu zreťazenia**, teda ako budeme vstupné údaje prideľovať segmentom a ako ich budeme optimálne prekrývať, aby sa dosiahla čo najvyššia vyťaženosť. Treťou je **návrh technických prostriedkov**, čiže konkrétne obvodové riešenie, interakcie medzi segmentmi, synchronizácia a riadenie.

Toto členenie je dôležité, pretože pekne oddeľuje tri rôzne roviny problému. Najprv treba vedieť, **čo** sa má rozdeliť. Potom treba určiť, **ako** sa bude tok riadiť. A až napokon sa rozhoduje, **čím** sa to fyzicky implementuje. Mnohé chyby pri uvažovaní o pipeline vznikajú práve vtedy, keď sa tieto tri roviny miešajú dokopy.

9. Prúdová funkčná jednotka a rezervačná tabuľka

Cieľom syntézy je návrh prúdovej funkčnej jednotky, teda PFJ. Tá môže mať lineárne usporiadanie stupňov alebo aj spätnoväzobné prepojenia. Správanie takejto jednotky sa popisuje pomocou **rezervačnej tabuľky (RTB)**. Rezervačná tabuľka ukazuje, v ktorých krokoch sú jednotlivé stupne zreťazenia aktivované pri spracovaní danej funkcie. Je to veľmi praktický nástroj, lebo presne zachytáva, kde vznikajú konflikty a kde sú voľné miesta na ďalšiu inicializáciu novej úlohy.

Inými slovami, RTB je časovo-priestorový opis využitia pipeline. Keď sa na ňu pozeráš, nevidíš len „aká je funkcia“, ale hlavne „kedy ktorý stupeň pracuje“. Práve z toho sa potom odvodzujú prípustné latentnosti a neskôr aj stavový graf inicializácií.

10. Lineárne a nelineárne systémy zreťazenia

Prednáška rozlišuje dva základné typy zreťazených systémov. **Lineárny systém zreťazenia** je taký, v ktorom vstupný prúd údajov postupne prechádza všetkými stupňami v pevne určenej postupnosti. Takýto systém je najjednoduchší na pochopenie aj na riadenie. **Nelineárny systém zreťazenia** má spätnoväzobné väzby. To znamená, že údaje nemusia prechádzať stupňami vždy v jednej jednoznačnej ceste a jedna funkcia môže byť opísaná viacerými rezervačnými tabuľkami. Takéto systémy sa označujú ako dynamické.

Pri lineárnom systéme je preto analýza jednoduchšia: všetko ide dopredu po pevnej trase. Pri nelineárnom systéme sa už musí sledovať, ktoré kombinácie inicializácií sa navzájom neblokujú. A práve tu sa stavový graf inicializácií stáva nevyhnutným nástrojom.

11. Celkový čas zreťazenia

Prednáška zavádza aj celkový čas zreťazenia TpT_pTp, ktorý sa pri lineárnom systéme skladá z časov jednotlivých stupňov. Každý stupeň môže zaberať jednu alebo viac periód základného taktu zreťazenia. Preto sa celkový čas vyjadruje ako súčet časov všetkých stupňov, prípadne ako súčet násobkov základnej taktovacej periódy. Význam tejto formulácie je jednoduchý: nie všetky stupne musia byť rovnako dlhé a nie každý krok pipeline je nutne jednocyklový.

To je veľmi dôležité pre návrh. Ak je jeden stupeň výrazne pomalší než ostatné, pipeline sa musí prispôsobiť jemu. V takom prípade sa zreťazenie síce formálne vytvorilo, ale jeho reálna priepustnosť môže byť výrazne horšia, než by sa čakalo od počtu stupňov. Úzky profil jedného stupňa teda môže obmedziť celý systém podobne, ako najslabší článok obmedzí reťaz.

12. Inicializácia a latentnosť

Veľmi dôležitým pojmom je **inicializácia zreťazenia**, teda okamih, keď sa do pipeline zavádza nová úloha. Medzi dvoma po sebe idúcimi inicializáciami je určitý počet synchronizačných krokov. Tento počet sa nazýva **latentnosť LLL**. Ak sa určitá postupnosť prípustných latentností cyklicky opakuje, hovoríme o **cykle latentností**. Pre takýto cyklus sa dá určiť priemerná latentnosť a najmenšia možná latentnosť LminL_{min}Lmin.

Latentnosť je kľúčová preto, že priamo vyjadruje, ako často môžeme do systému púšťať nové vstupy bez konfliktu. Ak je latentnosť malá, pipeline vie prijímať nové úlohy často a má vysokú priepustnosť. Ak je latentnosť veľká, pipeline síce možno formálne existuje, ale prijíma nové úlohy príliš pomaly, a tým sa stráca jej hlavná výhoda.

13. Stavový graf inicializácií (SGI)

Stavový graf inicializácií je nástroj, ktorý opisuje všetky prípustné prechody medzi nasledujúcimi inicializáciami. Každý vrchol grafu predstavuje určitý stav pipeline a hrany hovoria, po koľkých krokoch možno bez konfliktu urobiť ďalšiu inicializáciu. Hlavným cieľom SGI je umožniť určiť najmenšiu prípustnú latentnosť a nájsť optimálny spôsob synchronizácie vstupného prúdu údajov.

Vrchol SGI je označený **vektorom konfliktov**. Tento vektor hovorí, v ktorých budúcich krokoch je nová inicializácia prípustná a v ktorých nie. Hodnota 1 znamená, že inicializácia v danom kroku je neprípustná, hodnota 0 znamená, že prípustná je. Týmto sa veľmi elegantne formalizuje problém konfliktov v pipeline. Namiesto intuitívneho skúšania „čo sa ešte zmestí“ dostávame presný stavový model.

14. Vektor konfliktov a jeho význam

Začiatočný vektor konfliktov C0C_0C0 reprezentuje všetky predtým začaté inicializácie vzhľadom na prvú inicializáciu procesu. Ak chceme zistiť ďalší stav, pracujeme s posunutým vektorom konfliktov a podľa pravidiel aktualizácie dostávame nový vektor CiC_iCi. Logika je v zásade taká, že sledujeme, ako sa konfliktové okná posúvajú v čase a či sa pri novej inicializácii neprekryjú zakázané obsadenia tých istých stupňov.

Praktický význam je veľmi veľký. SGI neukazuje iba jeden „náhodný“ priebeh, ale všetky možné kombinácie bezkonfliktných inicializácií. Vďaka tomu sa dá vybrať optimálna synchronizácia prúdového vstupu. Pri syntéze zreťazeného systému je to zásadné, lebo od toho priamo závisí maximálna priepustnosť a využitie stupňov.

15. Riadenie zreťazeného systému

Riadenie pipeline zabezpečuje **riadiaca jednotka zreťazenia (RJZ)**. Jej hlavná funkcia je dvojitá. Po prvé riadi a synchronizuje vykonávanie funkcií jednotlivých stupňov zreťazenia. Po druhé riadi a synchronizuje samotné začiatky inicializácií na vstupe prúdovej funkčnej jednotky. Teda nestará sa len o „život vo vnútri“ stupňov, ale aj o to, kedy smie do pipeline vstúpiť ďalšia úloha.

Prednáška ďalej uvádza, že základným komponentom RJZ je posuvný register nastavený na začiatočnú hodnotu vektora konfliktov C0C_0C0. Tým je riadenie veľmi priamo previazané so stavovým grafom inicializácií. Teoretická analýza konfliktov sa tak mení na konkrétny riadiaci mechanizmus. To je jedna z najdôležitejších myšlienok tejto časti prednášky: SGI nie je len kresba na papieri, ale návod, ako pipeline reálne riadiť.

16. Riadenie jednotlivých stupňov

Riadenie každého stupňa zreťazenia vychádza zo všeobecnej schémy, v ktorej sa stupeň rozkladá na operačné obvody a pamäťové obvody. Vnútrostupňové riadenie zahŕňa najmä výber funkcie stupňa, aktiváciu logiky pamäťových obvodov, ich synchronizáciu a sledovanie stavu stupňa počas aktivácie. Prednáška tieto prvky označuje ako F,A,H,SF, A, H, SF,A,H,S.

Tým sa ukazuje, že pipeline nie je len „rad funkčných blokov“, ale aj precízne riadený synchronizačný systém. Každý stupeň musí vedieť, akú funkciu má vykonať, kedy sa má aktivovať a kedy má bezpečne odovzdať výsledok ďalej. Ak by sa to neriadilo dôsledne, stupne by si buď prekážali, alebo by vznikali neplatné výsledky.

17. Výkonnostné parametre zreťazeného systému

Základným parametrom je **priepustnosť zreťazenia PPP**. Tá vyjadruje priemerný počet inicializácií pripadajúci na jeden cyklus zreťazenia. Prednáška ju uvádza aj v tvare P=N/n=1/LminP = N/n = 1/L_{min}P=N/n=1/Lmin. Zmysel tohto vzťahu je veľmi intuitívny: čím menšia minimálna latentnosť, tým častejšie možno pipeline inicializovať, a tým väčšia je priepustnosť.

Druhým parametrom je **vyťažiteľnosť stupňov VVV**. Tá hovorí, aká časť celkového času a priestoru pipeline je reálne využitá prácou. Prednáška ju definuje pomerom počtu využití stupňov k celkovému počtu dostupných cyklov všetkých stupňov. Prakticky ide o odpoveď na otázku, či pipeline naozaj pracuje naplno, alebo len stojí a čaká. Dôležité je, že opatrenia, ktoré zvyšujú priepustnosť, zvyčajne zvyšujú aj vyťažiteľnosť, ale nie automaticky za každých okolností.

18. Vektorizácia ako pokračovanie myšlienky prúdového spracovania

Druhá veľká časť prednášky je venovaná vektorizácii. Vektorizácia znamená, že namiesto spracovania jedného skalárneho prvku po druhom sa operácia formuluje nad celým vektorom dát. Prednáška rozlišuje dve možnosti: buď ide o **prúdové spracovanie skalárnych inštrukcií programu**, alebo o **paralelné spracovanie vektorových inštrukcií programu**. Inými slovami, buď necháme pipeline bežať nad dlhým tokom jednoduchých inštrukcií, alebo zdvihneme úroveň abstrakcie a povieme priamo „načítaj vektor“, „vynásob vektor“, „pripočítaj vektor“, „ulož vektor“.

Vektorizácia je mimoriadne silná pri pravidelných numerických algoritmoch, kde sa rovnaká operácia opakuje nad veľkým počtom prvkov polí alebo matíc. Tam je možné využiť vysokú regularitu dát aj výpočtu. Práve preto boli vektorové architektúry historicky mimoriadne úspešné v superpočítačoch a numerických aplikáciách.

19. Príklad cyklu so skalárnymi inštrukciami

Prednáška používa cyklus

FOR i=1 to N:Z(i)=Z(i)+X(i)·Y(i)\text{FOR } i = 1 \text{ to } N:\quad Z(i)=Z(i)+X(i)\cdot Y(i)FOR i=1 to N:Z(i)=Z(i)+X(i)·Y(i)

V skalárnej verzii sa každá iterácia realizuje postupnosťou piatich inštrukcií: načítanie hodnoty, násobenie, sčítanie, uloženie výsledku a vetvenie späť na začiatok cyklu. Ak vykonanie jednej skalárnej inštrukcie trvá 111 strojový cyklus, potom sa celý program v jednoprocesorovom systéme vykoná za 5N·TSC5N \cdot T_{SC}5N·TSC. Tento spôsob je typický pre architektúry typu SISD, teda klasické sekvenčné počítače.

Dôležité je uvedomiť si, že väčšina času sa tu nestráca na „zložitosti matematiky“, ale na opakovaní tej istej krátkej sekvencie riadiacich a dátových operácií pre každý prvok zvlášť. A práve to je dôvod, prečo sa takýto kód výborne hodí na vektorizáciu.

20. Tá istá úloha vo vektorovej verzii

Vo vektorovej verzii sa namiesto päťinštrukčnej slučky používa postupnosť vektorových inštrukcií, napríklad **VLD**, **VMUL**, **VADD** a **VST**. Po rozvinutí cyklu už nie je potrebná podmienená skoková inštrukcia pri každom prvku. Každá vektorová inštrukcia reprezentuje celú sériu skalárnych operácií nad všetkými prvkami vektora. Prednáška priamo ukazuje, že

napríklad pri N=8N=8N=8 jedna vektorová inštrukcia zodpovedá ôsmim skalárnym operáciám rovnakého typu.

Ak vykonanie operácie vektorovej inštrukcie po elementoch trvá 111 strojový cyklus a máme k dispozícii NNN-procesorový systém, potom sa uvedený program vykoná za 4·TSC4 \cdot T_{SC}4·TSC. Vetvenie netreba, lebo práca je už vyjadrená priamo na úrovni vektora. Prednáška uvádza, že takýto postup je charakteristický pre architektúry typu SIMD, prípadne MIMD, ak sú vektorové operácie riadené nezávislými inštrukciami.

21. Prečo je vektorizácia taká účinná

Hlavný prínos vektorizácie je v tom, že odstraňuje značnú časť riadiacej režijnosti. Namiesto opakovaného načítania inštrukcií a opakovaného testovania konca cyklu sa celý rad podobných operácií spojí do jednej vektorovej operácie. To znižuje počet inštrukcií, obmedzuje vetvenie a zvyšuje možnosť dlhého súvislého toku dát do pipeline. Inak povedané, vektorizácia premieňa množstvo drobných skalárnych krokov na menší počet dlhých a pravidelných operácií, ktoré pipeline miluje.

Druhý dôležitý dôvod je údajová nezávislosť medzi prvkami vektora. Ak sa jednotlivé elementy neovplyvňujú, pipeline môže spracúvať jeden prvok za druhým bez zastavení a hazardov. Práve táto vlastnosť robí z vektorových výpočtov ideálny prípad pre prúdové spracovanie.

22. Koeficient vektorizácie

Prednáška zavádza **koeficient vektorizácie rrr**. Ten vyjadruje, aká časť pôvodného skalárneho programu je vhodná na prevedenie do vektorovej formy. Ak je rrr malé, väčšina programu ostáva skalárna a zisk z vektorizácie bude obmedzený. Ak je rrr veľké, väčšia časť času sa dá urýchliť vektorovým spracovaním a výsledný program sa výrazne zrýchli. Prednáška pritom rozkladá pôvodný čas TsT_sTs na dve časti: nevektorizovanú časť (1−r)Ts(1-r)T_s(1−r)Ts a vektorizovanú časť rTsrT_srTs. Výsledný vektorový čas TvT_vTv potom vzniká tak, že nevektorizovaná časť ostáva prakticky nezmenená, zatiaľ čo vektorizovaná časť sa skracuje vďaka vektorovému vykonaniu.

Zmysel tejto myšlienky je rovnaký ako pri Amdahlovom zákone: čokoľvek sa nepodarí vektorizovať, zostáva limitom výsledného zrýchlenia. Preto je pri praktickej optimalizácii dôležité nielen mať výkonný vektorový hardvér, ale najmä dosiahnuť vysokú mieru vektorizácie samotného kódu.

23. Prúdové spracovanie inštrukcií cyklu

V závere prednáška rozlišuje **vnútorné zreťazenie** a **vonkajšie zreťazenie** pri spracovaní cyklu. Vnútorné zreťazenie znamená, že pipeline pracuje vo vnútri jednej iterácie – prekrývajú sa fázy inštrukcií ako fetch, decode, execute. Vonkajšie zreťazenie znamená, že sa prekrývajú celé iterácie cyklu medzi sebou. To je dôležité pri dlhých slučkách, lebo paralelizmus nevzniká len „v rámci jednej inštrukcie“, ale aj medzi po sebe idúcimi iteráciami.

Prednáška pri tom zavádza periódu hodinového taktu THT_HTH, periódu sériového vykonania jednej iterácie TST_STS a periódu prúdového vykonania jednej iterácie TPT_PTP. Myšlienka je jednoduchá: ak sa iterácie dajú bezpečne prekrývať, nová iterácia nemusí čakať na úplné dokončenie predchádzajúcej. Práve preto sa pri cykloch často kombinujú techniky zreťazenia, rozvinutia cyklu a vektorizácie.

Prednáška 6: Prepojovacie siete

1. Čo sú prepojovacie siete a prečo sú dôležité

Prepojovacia sieť je hardvérový prostriedok, ktorý zabezpečuje komunikáciu medzi komponentmi paralelného systému. V praxi rieši najmä dva základné modely komunikácie: **procesor – pamäť (P–M)** a **procesor – procesor (P–P)**. Zmysel prepojovacej siete je v tom, že pri väčšom počte procesorov už nestačí jednoduché pripojenie všetkých prvkov na jednu spoločnú zbernicu. Ak má systém rásť a zároveň si zachovať výkon, musí mať organizovanú a škálovateľnú komunikačnú štruktúru.

Prednáška rozlišuje tri základné spôsoby prepájania komponentov:

- **jednoduchá zdieľaná zbernica** – používa sa v menších a menej náročných systémoch, kde sa nevyžaduje veľa súbežných prenosov,
- **viacnásobná zbernica** – typická pre klasické multiprocesorové systémy, najmä v modeli P–M,
- **prepojovacia sieť** – používa sa vo výkonných paralelných systémoch s veľkým počtom procesorových elementov.

---

2. Základná definícia prepojovacej siete

Prepojovacia sieť sa označuje ako **PS [N × M]**, kde:

- **N** je počet vstupov,
- **M** je počet výstupov.

To znamená, že sieť prepája množinu vstupných komponentov s množinou výstupných komponentov. Ak prepája procesory s pamäťami, ide o komunikáciu typu P–M. Ak prepája navzájom procesory alebo výpočtové elementy, ide o komunikáciu typu P–P. Formálne sa správanie siete vyjadruje prepojovacou funkciou:

F: X → Y

kde XXX je množina vstupov a YYY je množina výstupov. Dôležitá nie je len existencia tejto funkcie, ale aj to, aké prepojenia sieť dovolí, aké nepovolí a za akú cenu ich vie vytvoriť.

---

3. Základné delenie prepojovacích sietí

Prepojovacie siete sa v prednáške delia podľa dvoch hľadísk.

Podľa typu prepojenia

- **jednostranné siete**,
- **obojstranné siete**.

Podľa smeru komunikácie

- **jednosmerné siete**,
- **obojsmerné siete**.

Z kombinácie týchto vlastností vznikajú základné typy, napríklad:

- jednostranná obojsmerná sieť,
- jednostranná jednosmerná sieť,
- obojstranná obojsmerná sieť,
- obojstranná jednosmerná sieť.

Zmysel tohto delenia je praktický. Nie každá architektúra potrebuje rovnaký druh prenosu. Niekde stačí jednosmerný tok dát, inde treba plne obojsmernú komunikáciu medzi rovnocennými uzlami.

---

4. Prepínací prvok ako základný stavebný blok siete

Veľké prepojovacie siete sa neskladajú z jedného obrovského „prepínača“, ale z množstva malých prepínacích prvkov. Prednáška ukazuje najmä prepínač typu **[2 × 2]**, ktorý sa často používa vo viacstupňových sieťach.

Takýto prepínač môže podľa riadiaceho signálu realizovať rôzne stavy:

- priamy prechod,
- výmenu prechodov,
- výber hornej alebo dolnej vetvy,
- v niektorých prípadoch aj kopírovanie na viac výstupov.

To znamená, že sieť je určená dvoma vecami:

1. **topológiou**, teda tým, ako sú prepínače fyzicky prepojené,
2. **riadením**, teda tým, ako sú prepínače v danom okamihu nastavené.

Práve kombinácia topológie a riadenia určuje, aké cesty medzi vstupmi a výstupmi sa dajú vytvoriť.

---

5. Štvorcové prepojovacie siete

Osobitný význam majú **štvorcové prepojovacie siete**, teda siete, pre ktoré platí:

N = M

Pri nich sa zavádzajú dva dôležité pojmy:

Vyžadovaný stav siete

Je to množina dvojíc vstup–výstup, ktoré chceme v sieti prepojiť. Inými slovami, opisuje, čo od siete práve požadujeme.

Skutočný stav siete

Je to reálne nastavenie siete v danom okamihu. Nemusí vždy zodpovedať požadovanému stavu, pretože topológia siete môže niektoré požadované prepojenia blokovať.

Na základe vzťahu medzi vyžadovaným a skutočným stavom sa siete delia na:

- **blokujúce**,
- **neblokujúce**,
- **prestaviteľné**.

---

6. Blokujúce, neblokujúce a prestaviteľné siete

Blokujúca sieť

Blokujúca prepojovacia sieť je taká, v ktorej existuje aspoň jedno požadované prepojenie medzi vstupom a výstupom, ktoré nemožno realizovať, pretože mu bráni iné už vytvorené alebo súčasne požadované spojenie. To znamená, že sieť nemá dostatočnú flexibilitu na ľubovoľnú kombináciu spojení.

Neblokujúca sieť

Neblokujúca sieť je taká, v ktorej možno vytvoriť spojenie medzi ľubovoľnými dvojicami vstupov a výstupov. Je flexibilnejšia, ale zvyčajne aj zložitejšia a drahšia. Výhodou je, že pri vytváraní nových spojení nehrozí blokovanie inými cestami.

Prestaviteľná sieť

Prestaviteľná sieť je špeciálny prípad blokujúcej siete. Pri aktuálnom nastavení môže byť požadované spojenie zablokované, ale ak sa niektoré existujúce cesty presmerujú inak,

zablokované spojenie sa dá nakoniec vytvoriť. To znamená, že sieť síce nie je úplne neblokujúca, ale vie sa preusporiadať tak, aby vyriešila konflikt.

---

7. Údajový manipulátor

Prednáška zavádza aj pojem **údajový manipulátor**. Ide o osobitnú skupinu prepojovacích sietí, ktoré sa používajú najmä v architektúrach typu MIMD. Ich dôležitou vlastnosťou je, že vedia **jeden vstup prepojiť s viacerými výstupmi**. To znamená, že neslúžia len na jednoduché presmerovanie dát, ale aj na ich kopírovanie alebo rozmiestňovanie.

Toto je zásadný rozdiel oproti bežnej permutačnej sieti, kde sa zvyčajne rieši hlavne prestavenie poradia alebo výmena vstupov a výstupov. Údajový manipulátor je teda vhodný tam, kde potrebujeme nielen prepájať, ale aj aktívne manipulovať s dátovým tokom.

---

8. Permutačné siete

Prednáška delí permutačné siete na:

- **jednostupňové**,
- **viacstupňové**.

Ich základom je **prepojovacia funkcia**, ktorá opisuje, ako sa binárna adresa vstupu mení na adresu výstupu. V tomto kontexte sa adresy nechápu len ako čísla portov, ale ako bitové reťazce. Práve nad týmito bitmi sa definujú jednotlivé permutačné operácie. To je dôležité, pretože mnohé siete sa konštruujú systematicky ako opakovanie určitých operácií nad bitovou reprezentáciou adries.

---

9. Základné permutačné operácie

Prednáška uvádza tieto základné permutačné operácie:

Dokonalé premiešanie (shuffle)

Pri tejto operácii sa bity adresy cyklicky presunú. Najvyšší bit sa presunie na koniec. Výsledkom je pravidelné premiešanie výstupov, ktoré sa často používa medzi stupňami viacstupňových sietí.

Inverzné premiešanie (shuffle^{-1})

Je to opačná operácia k shuffle. Bity sa posunú späť opačným smerom. Aj táto operácia sa používa ako medzistupňová permutácia v niektorých typoch sietí.

Výmena (exchange)

Ide o jednoduchú permutáciu, ktorá typicky mení dvojice liniek alebo pracuje ako elementárna lokálna výmena. V prednáške sa spája s funkciou typu cube_0.

Preklopenie (bit reversal)

Pri tejto operácii sa obráti poradie bitov v adrese. To je známa operácia napríklad z FFT algoritmov a z niektorých paralelných prepojovacích štruktúr.

Motýliková operácia (butterfly)

Butterfly operácia presúva konkrétny bit na inú pozíciu a tým vytvára štruktúru typickú pre motýlikové siete. Ide o veľmi dôležitú operáciu najmä pri viacstupňových topológiách.

---

10. Jednostupňové siete

Jednostupňové siete realizujú komunikáciu v rámci jednej prepájacej vrstvy alebo jednej pravidelnej topológie.

N-dimenzionálna kocka

Pri hyperkocke sa uzly reprezentujú binárnymi adresami a dve adresy sú susedné vtedy, keď sa líšia v jednom bite. Funkcia cubeicube_icubei teda znamená preklopenie i-teho bitu adresy. Výhodou hyperkocky je pravidelnosť a pomerne krátke komunikačné cesty vzhľadom na počet uzlov. Je to veľmi elegantný príklad toho, ako sa sieť dá odvodiť priamo z práce s bitmi adresy.

Štvorcová mriežka

Mriežková topológia prepája uzly so susedmi v dvoch rozmeroch. Používajú sa funkcie typu posun o jeden krok v riadku alebo o jeden krok v stĺpci. Táto topológia je veľmi prirodzená pri fyzickom rozložení procesorov alebo spracovacích elementov, ale komunikácia medzi vzdialenými uzlami býva dlhšia než v niektorých iných topológiách.

---

11. Viacstupňové prepojovacie siete

Viacstupňové siete vznikajú tak, že sa použije viac vrstiev jednoduchých prepínačov a medzi nimi sa vodiče pravidelne prepájajú pomocou permutačných operácií. Ich hlavnou výhodou

je to, že dokážu poskytovať bohaté možnosti komunikácie bez potreby jednej obrovskej a drahej plne prepojenej siete.

Prednáška ich opisuje ako kombináciu:

- **permutačných funkcií**,
- **prepínačov typu E**,
- a opakovania týchto prvkov v niekoľkých stupňoch.

To znamená, že viacstupňová sieť je vlastne pravidelná skladba malých prepínacích blokov a medzistupňových permutácií.

---

12. Omega sieť

Omega sieť patrí medzi najznámejšie viacstupňové permutačné siete. Je vytvorená tak, že medzi jednotlivými stupňami prepínačov sa používa **dokonalé premiešanie (shuffle)**. Výhodou je veľmi pravidelná štruktúra a jednoduché smerovanie podľa bitov cieľovej adresy. Nevýhodou je, že ide o sieť, v ktorej môže dochádzať k blokovaniu. To znamená, že nie všetky kombinácie spojení sa dajú vytvoriť súčasne.

Z učebného hľadiska je Omega dôležitá preto, že je typickým príkladom pravidelnej viacstupňovej siete: jednoduchá, škálovateľná, dobre analyzovateľná, ale nie úplne bezkonfliktná.

---

13. Kubická sieť, základná sieť, preklápacia sieť a R-sieť

Prednáška uvádza aj ďalšie typy viacstupňových sietí.

Kubická sieť

Táto sieť využíva postupnosť motýlikových operácií a je odvodená z práce s bitovými reprezentáciami adries. Dôležité je pochopiť, že vzniká z iného typu medzistupňových transformácií než Omega sieť.

Základná sieť ZNZ_NZN

Je to jedna z pravidelných viacstupňových sietí definovaných cez postupnosť prepínačov a permutačných operácií. Slúži ako základ pre ďalšie odvodené štruktúry.

Preklápacia sieť FNF_NFN

Používa inverzné dokonalé premiešanie medzi stupňami. Je to ďalší príklad toho, ako zmena medzistupňovej permutácie vedie k odlišnej topológii.

R-sieť

Je odvodená od základnej siete ZNZ_NZN. Význam tejto siete v prednáške je najmä klasifikačný: ukazuje, že existuje viac príbuzných rodín viacstupňových sietí, ktoré sa dajú systematicky odvodiť.

---

14. Benešova sieť

Benešova permutačná sieť je v prednáške dôležitá tým, že sa výslovne uvádza ako **neblokujúca sieť**. Má:

- **2log(N)−12\log(N)-12log(N)−1 stupňov**,
- v každom stupni **N/2N/2N/2** prepínačov typu **[2×2]**.

Z každého vstupu Benešovej siete možno definovať viac možných ciest, a práve to jej umožňuje vysokú flexibilitu pri prepájaní. Je to veľmi dôležitý príklad, pretože ukazuje, že aj viacstupňová sieť môže byť neblokujúca, ak je vhodne navrhnutá. Z pohľadu učiva si ju treba pamätať ako kľúčový vzor neblokujúcej viacstupňovej architektúry.

---

15. SIMD vs. MIMD pohľad na prepojovanie

Prednáška zdôrazňuje, že požiadavky architektúr SIMD a MIMD nie sú rovnaké.

SIMD architektúry

Sú synchrónne a často potrebujú pravidelné manipulačné operácie nad dátami. Preto sa v nich prirodzene uplatňujú **permutačné siete**. Tie dobre podporujú presuny, premiešanie alebo systematické preusporiadanie údajov.

MIMD architektúry

Tu sa kladie dôraz na čo najlepšie všeobecné prepojenie medzi nezávisle pracujúcimi prvkami. Práve preto sa v tejto súvislosti uvádzajú **údajové manipulátory**, ktoré vedia lepšie podporovať flexibilnú komunikáciu a kopírovanie údajov.

---

16. Manipulačné funkcie

Pri údajových manipulátoroch sa zavádzajú tzv. **manipulačné funkcie**. Prednáška ich delí na štyri základné triedy:

- **permutácia**,
- **kopírovanie**,
- **rozmiestnenie**,
- **maskovanie**.

To znamená, že údaje sa v sieti nemusia iba presúvať z jedného miesta na druhé. Sieť môže zabezpečiť aj to, že sa:

- poradie údajov zmení,
- jedna hodnota rozmnoží na viac miest,
- údaje sa rozdelia podľa určitého vzoru,
- niektoré prenosy sa potlačia alebo vyfiltrujú.

Takýto pohľad je veľmi dôležitý, lebo ukazuje, že prepojovacia sieť môže byť aj nástrojom aktívnej dátovej manipulácie, nielen pasívnym prenosovým kanálom.

---

17. Štruktúra údajového manipulátora

Údajový manipulátor je opísaný ako viacstupňová sieť M[N×N]M[N \times N]M[N×N], vytvorená na báze prepínacích elementov s tromi výstupnými linkami. V každom stupni sa údaje môžu smerovať:

- na „nižší“ uzol,
- na identický uzol,
- alebo na „vyšší“ uzol.

Prednáška to zapisuje pomocou funkcií typu:

- M−(j)M^-(j)M−(j),
- I(j)I(j)I(j),
- M+(j)M^+(j)M+(j),

teda posun doľava, identita a posun doprava modulo veľkosti siete. To vytvára pravidelnú manipulačnú štruktúru, v ktorej možno realizovať presuny aj kopírovanie údajov.

Hlavná myšlienka je, že prepínač si nevyberá len jednu výstupnú cestu, ale podľa nastavenej funkcie môže pripojiť vstup na jednu, dve alebo tri výstupné linky. Tým sa vysvetľuje, prečo údajový manipulátor podporuje kopírovanie a rozmiestňovanie dát.

Prednáška 7: ILP architektúry

1. Čo znamená ILP

ILP je **Instruction-Level Parallelism**, teda paralelizmus na úrovni inštrukcií. Ide o snahu zvýšiť výkon procesora tak, aby sa počas krátkeho časového intervalu vykonalo viac práce, aj keď program na úrovni zdrojového kódu vyzerá sekvenčne. Prednáška vychádza z toho, že výkon sa dá zvyšovať dvoma hlavnými spôsobmi: buď sa zvýši pracovná frekvencia, alebo sa funkčne vylepší architektúra tak, aby vedela spracovať viac inštrukcií súbežne. Pri ILP nás zaujíma najmä druhá možnosť.

Z pohľadu vývoja prednáška ukazuje postupný prechod od konvenčných procesorov k:

- skalárnym ILP procesorom,
- superskalárnym procesorom,
- a VLIW procesorom.

Teda nejde o úplne odlišné svety, ale o vývoj jednej línie architektúr smerom k väčšiemu paralelizmu vo vnútri jedného procesora.

---

2. Dva základné spôsoby paralelizmu v ILP procesoroch

Prednáška rozlišuje dve základné cesty, ako sa v ILP procesoroch dosahuje paralelizmus:

Časový paralelizmus

Dosahuje sa **prúdovým spracovaním**, teda pipeline. Výpočet sa rozdelí na fázy a tieto fázy sa prekrývajú v čase. Znamená to, že jedna inštrukcia sa môže dekódovať, zatiaľ čo iná sa už vykonáva a ďalšia sa načítava.

Priestorový paralelizmus

Dosahuje sa **zvyšovaním počtu funkčných jednotiek**. Namiesto jednej ALU alebo jednej vykonávacej vetvy má procesor viac jednotiek, ktoré môžu pracovať naraz. To umožňuje vykonať viac inštrukcií v tom istom takte, ak medzi nimi nie sú konflikty.

Prednáška tým zdôrazňuje, že moderné ILP architektúry nie sú len „hlbšie pipeline“, ale kombinujú časový aj priestorový paralelizmus. Práve podľa toho sa potom odlišujú skalárne, superskalárne a VLIW procesory.

---

3. Základné triedy ILP procesorov

Prednáška rozdeľuje ILP procesory na tri hlavné triedy.

Skalárne procesory

Skalárny procesor spravidla vykoná jednu jednoduchú inštrukciu za jeden strojový cyklus pipeline. To neznamená, že je úplne sekvenčný bez prekrývania, ale že jeho pipeline typicky spracúva jednu inštrukciu na jednu „výstupnú vetvu“.

Superskalárne procesory

Superskalárny procesor vie v jednom strojovom cykle pipeline spracovať **niekoľko inštrukcií naraz**. Charakteristické je, že má viac prúdov spracovania a preto sa o ňom hovorí ako o multiprúdovom procesore. Tu už nestačí len pipeline, ale treba riešiť aj paralelné dekódovanie, výber inštrukcií, konflikty, premenovávanie registrov a podobne.

VLIW procesory

VLIW znamená **Very Long Instruction Word**. Tento prístup kombinuje horizontálne mikroprogramovanie a myšlienku paralelného vykonávania viacerých operácií. Rozdiel oproti superskaláru je v tom, že veľká časť plánovania paralelizmu sa presúva z hardvéru na kompilátor.

---

4. Spoločný pohľad na VLIW a superskalárne procesory

Prednáška ukazuje, že VLIW aj superskalárny procesor sa snažia o podobný cieľ: vykonať v krátkom čase viac operácií. Rozdiel nie je v tom, že jeden je paralelný a druhý nie, ale **kde sa robí plánovanie paralelizmu**.

- Pri **superskalárnom procesore** hľadá paralelizmus najmä hardvér počas behu programu.
- Pri **VLIW procesore** pripravuje paralelizmus vopred kompilátor a hardvér je jednoduchší v oblasti dynamického plánovania.

To je jedna z najdôležitejších myšlienok celej prednášky. Obe architektúry chcú využívať viac funkčných jednotiek, ale úplne inak si delia úlohu medzi prekladač a hardvér.

---

5. Paralelizmus v skalárnych procesoroch

Prednáška najprv rozoberá skalárne procesory, pretože práve na nich sa najľahšie vysvetľuje základná pipeline logika. Predpoklad je, že inštrukčný cyklus sa dá rozdeliť na dielčie fázy a tieto fázy sa dajú prekrývať s fázami predchádzajúcich a nasledujúcich inštrukcií. To je klasické prúdové vykonávanie inštrukcií.

Základné stupne pipeline sú:

- **F** – fetch, načítanie inštrukcie z pamäte,
- **D** – decode, dekódovanie inštrukcie,
- **E** – execute, vykonanie operácie,

- **WB** – write back, zápis výsledku do registra alebo pamäte.

Prednáška tým pripomína, že aj obyčajný skalárny procesor už obsahuje určitý paralelizmus, len je to hlavne časový paralelizmus cez pipeline. ILP sa teda nezačína až pri superskalároch – začína už pri samotnom prúdovom spracovaní inštrukcií.

---

6. Architektonické typy skalárnych procesorov

Prednáška rozlišuje tri základné línie skalárnych procesorov:

CISC

CISC architektúry používajú bohatú a komplexnú inštrukčnú sadu. Tradične mali funkčné jednotky pre celočíselné operácie priamo v procesore a operácie s pohyblivou rádovou čiarkou často riešili cez koprocesor. Príkladom vývoja v tejto vetve sú staršie procesory Intel, DEC či VAX systémy.

RISC

RISC architektúry sa vyvíjali ako prostriedok pre efektívne vykonávanie programov vyšších jazykov. Typické črty sú:

- jednoduchšie inštrukcie,
- viac registrov,
- dôraz na pipeline,
- a presun zložitosti na kompilátor.

Prednáška ako príklady uvádza RISC I a RISC II z Berkeley, MIPS zo Stanfordu a SPARC od Sun Microsystems. Pri MIPS je vyslovene zdôraznené, že kompilátor rieši otázku hazardov, čo je veľmi typická filozofia klasických RISC návrhov.

Zásobníkovo orientované architektúry

Sem patria najmä JavaChips, teda architektúry optimalizované pre JVM. Ide o zaujímavý smer, pretože ukazuje, že ILP princípy sa dajú aplikovať aj na zásobníkové modely, nie iba na klasické registrové architektúry.

---

7. RISC procesory v prednáške

Prednáška venuje väčšiu pozornosť práve RISC vetve, pretože odtiaľ prirodzene vyrastajú moderné ILP techniky.

RISC I a RISC II

Pri týchto procesoroch sa zdôrazňuje:

- použitie **registrových okien**,
- 3-stupňová pipeline,
- malé a jednoduché inštrukcie,
- vyrovnávacia pamäť pre skokové inštrukcie,
- oneskorovanie realizácie vetvenia.

MIPS

Pri MIPS sú kľúčové tieto znaky:

- 5-stupňová pipeline,
- 32-bitové inštrukcie,
- tri formáty inštrukcií: R, I, J,
- obmedzený počet registrov,
- kompilátor rieši hazardy.

SPARC

Pri SPARC sa zdôrazňuje:

- nahradenie viaccyklových inštrukcií jednocyklovými,
- použitie registrového okna,
- a neskorší vývoj smerom k viacjadrovým a viacvláknovým procesorom.

Tieto príklady majú v prednáške význam preto, že ukazujú, ako sa pipeline, práca s registrami a zjednodušenie inštrukčnej sady stali základom pre ďalší rast ILP.

---

8. Zásobníkové ILP architektúry – JavaChips

Prednáška uvádza ako zaujímavú vetvu procesory **picoJava-I, picoJava-II, microJava 701 a UltraJava**. Tieto procesory sú orientované na prostredie JVM a teda na zásobníkový spôsob vykonávania. To je dôležité, pretože ukazujú, že ILP sa nemusí viazať iba na klasické registre všeobecného účelu.

picoJava-I

Charakteristické črty:

- 4-stupňová pipeline,
- stupne F – D – E – WB.

picoJava-II

Charakteristické črty:

- optimalizované hardvérové prostredie pre JVM,
- hardvérová implementácia niektorých JVM inštrukcií,
- podpora aj pre iné vyššie jazyky,
- zásobníková architektúra,
- približne 300 inštrukcií.

microJava 701

Prednáška pri ňom uvádza viac detailov:

- 32-bitová implementácia picoJava-II,
- 200 MHz,
- 2,8 milióna tranzistorov,
- 6-stupňová pipeline,
- logika prekrývania štyroch inštrukcií,
- I-cache a D-cache,
- viac paralelných zberníc medzi jednotkami.

Zaujímavá je najmä tzv. **Instruction Folding Unit**, teda logika prekrývania inštrukcií. Jej zmysel je zredukovať režijnosť zásobníkových operácií a spojiť viac jednoduchých krokov do efektívnejšieho vnútorného vykonania. Tým sa zlepšuje využitie pipeline aj pri zásobníkovej architektúre.

---

9. Superskalárne procesory – základná myšlienka

Superskalárny procesor je ďalší krok za obyčajnú pipeline. Nestačí mu, že jedna inštrukcia je v každom stupni pipeline. On sa snaží dosiahnuť, aby sa **v jednom takte pipeline spracovalo viac inštrukcií naraz**. Na to potrebuje:

- viac prúdových funkčných jednotiek,
- viac vykonávacích jednotiek,
- paralelné dekódovanie,
- výber vhodných inštrukcií,
- a mechanizmy na zachovanie korektnosti pri konfliktoch.

Prednáška ako kľúčové aspekty superskalárneho spracovania uvádza:

- paralelné dekódovanie,
- superskalárny výber,
- paralelné vykonávanie,
- konzistenciu vykonávania,
- konzistenciu spracovania výnimiek.

To je veľmi dôležité, lebo ukazuje, že superskalárnosť nie je „iba viac ALU“. Je to celá sada mechanizmov, ktoré musia spolupracovať.

---

10. Paralelné dekódovanie

Ak má procesor vyberať viac inštrukcií za takt, musí ich aj viac naraz načítať a dekódovať. Preto prednáška vysvetľuje, prečo je potrebné **paralelné dekódovanie**. Dôvody sú dva:

- procesor má viac pipeline vetiev,
- a treba priebežne kontrolovať výskyt hazardov medzi inštrukciami.

S tým súvisí aj **preddekódovanie** inštrukcií v I-cache. Pri niektorých procesoroch sa ku každej inštrukcii pridávajú pomocné príznakové bity, aby bolo neskoršie dekódovanie a výber rýchlejšie. Prednáška uvádza príklad AMD-K5, kde sa každý bajt rozširoval o príznakové bity. Zmysel je znížiť záťaž kritickej dekódovacej cesty.

---

11. Superskalárny výber inštrukcií

Výber inštrukcií je jadro superskalárneho procesora. Nestačí mať viac pripravených inštrukcií – treba z nich vybrať také, ktoré sa dajú naozaj vykonať súbežne. Prednáška rozoberá štyri hlavné aspekty:

a) Riadenie fázy výberu

Tu ide o to, akými mechanizmami sa procesor snaží odstrániť alebo obísť prekážky pri výbere. Patrí sem:

- riešenie riadiacich hazardov,
- riešenie WAR a WAW hazardov,
- potlačenie blokovania výberu,
- spôsob plánovania výberu.

b) Politika riadenia výberu

Prednáška uvádza konkrétne nástroje:

- **NOP cykly**,
- **špekulatívne vetvenie**,
- **premenovávanie registrov**,
- priame a nepriame plánovanie výberu.

c) Plánovanie výberu

Výber môže byť:

- **v zhodnom poradí**,
- alebo **v nezhodnom, teda preusporiadanom poradí**.

S tým súvisí aj **prehľadávacie okno**, ktoré môže byť:

- fixované,
- alebo pohyblivé.

d) Počet súčasne spracovaných inštrukcií

Prednáška ukazuje, že reálne procesory tejto éry spracúvali približne 2, 3, 4 alebo 6 inštrukcií superskalárne. To pekne ilustruje, že šírka superskalárneho spracovania je praktický návrhový parameter a nie neobmedzená hodnota.

---

12. Zhodné a nezhodné poradie výberu

Toto je jedna z najdôležitejších častí prednášky.

Výber v zhodnom poradí

Procesor číta a vydáva inštrukcie v rovnakom poradí, v akom sú v programe. Je to jednoduchšie na riadenie, ale ak sa v ceste objaví závislá inštrukcia, môže blokovať aj ďalšie za ňou, hoci by tie mohli byť nezávislé.

Výber v nezhodnom poradí

Procesor si vie z prehľadávacieho okna vybrať neskoršiu bezkonfliktovú inštrukciu a vydať ju skôr než staršiu, ktorá je práve zablokovaná závislosťou. To vedie k lepšiemu využitiu funkčných jednotiek, ale zároveň výrazne komplikuje riadenie a neskoršie dokončovanie inštrukcií.

Prednáška tým vysvetľuje hlavnú motiváciu dynamického plánovania: ak procesor vie preskočiť momentálne problémové inštrukcie a vykonať zatiaľ iné, získa vyšší výkon. Cena za to je však zložitejšia architektúra.

---

13. Fixované a pohyblivé prehľadávacie okno

Prehľadávacie okno je oblasť inštrukcií, v ktorej procesor hľadá kandidátov na vykonanie.

Fixované okno

Má pevný rozsah a výber sa robí v rámci tohto pevného bloku. Je jednoduchšie, ale menej flexibilné.

Pohyblivé okno

Vie sa dynamicky posúvať a lepšie hľadať bezkonfliktové inštrukcie. Je výkonnejšie, ale zložitejšie. Moderné superskalárne procesory smerovali práve k pohyblivým oknám, k špekulatívnemu vetveniu a k premenovávaniu registrov.

Prednáška tu v podstate ukazuje evolúciu od jednoduchých blokovaných výberov až po „moderný“ superskalárny výber s dynamickým plánovaním.

---

14. Premenovávanie registrov

Premenovávanie registrov je mechanizmus, ktorý odstraňuje najmä **WAR a WAW hazardy**. Tieto hazardy nevznikajú preto, že by si inštrukcie logicky odovzdávali hodnoty, ale preto, že používajú rovnaké architektonické mená registrov. Ak vie procesor priradiť týmto zápisom odlišné fyzické registre, zdanlivý konflikt zmizne.

Prednáška premenovávanie zaraďuje medzi kľúčové prvky modernej fázy výberu. Je to dôležitá technika, pretože bez nej by sa veľká časť potenciálneho paralelizmu zbytočne stratila na umelých menových konfliktoch.

---

15. Oddelenie fázy výberu

Prednáška samostatne vysvetľuje, prečo sa oplatí **oddeliť fázu výberu inštrukcií** od ostatných častí spracovania. Hlavným dôvodom je eliminácia blokovania výberu, ktoré by inak vznikalo v dôsledku údajových a zdrojových hazardov.

Pri tomto oddelení sa sledujú tri hlavné aspekty:

- trieda inštrukcií, pre ktoré sa plánovací zásobník používa,
- spôsob realizácie plánovacieho zásobníka,
- politika načítania operandov.

To je dôležité, pretože ukazuje, že issue fáza v superskaláre nie je len „jedna pipeline etapa“, ale pomerne komplexný podsystém plánovania.

---

16. Plánovací zásobník a ROB

Prednáška používa pojem **P-zásobník**, často priamo spojený s **ROB (Reorder Buffer)**.

Na čo slúži ROB

ROB je pamäť preusporiadania inštrukcií. Jeho úlohou je:

- držať informácie o inštrukciách, ktoré už boli vydané,
- umožniť preusporiadané vykonávanie,
- ale pritom zabezpečiť správne usporiadané dokončovanie, ak to architektúra vyžaduje.

Prednáška uvádza, že ROB môže mať funkciu:

- len preusporiadania,
- preusporiadania a premenovávania,
- alebo preusporiadania, premenovávania a blokovania.

Typy realizácie P-zásobníka

- **špecializovaný (decentralizovaný)**,
- **skupinový**,
- **centralizovaný**,
- prípadne kombinovaný zásobník typu **DRIS**.

Kapacita

Prednáška uvádza historické aj modernejšie veľkosti:

- špecializovaný: napr. 1,
- skupinový: napr. 16,
- centralizovaný: napr. 20,
- a v novších procesoroch rádovo stovky záznamov.

Z hľadiska pochopenia je najdôležitejšie vedieť, že ROB je nástroj, ktorý umožňuje spojiť výhodu nezhodného vykonávania s potrebou zachovať korektný architektonický stav.

---

17. Načítanie operandov: issue vs. dispatch

Prednáška rozlišuje dve politiky načítania operandov.

Načítanie operandov vo fáze výberu (issue)

Rezervačná stanica už obsahuje hodnoty zdrojových registrov. To znamená, že pri vydaní inštrukcie sa so sebou nesie aj samotný operand. Tento prístup používali napríklad PowerPC 620, AMD K5 alebo Pentium Pro.

Načítanie operandov vo fáze odoslania (dispatch)

Rezervačná stanica neobsahuje priamo hodnoty, ale len indexy či odkazy na zdrojové registre. Skutočné načítanie hodnôt sa deje neskôr. Tento prístup sa spája napríklad s CDC 6600, PA 8000 alebo R10000.

Toto delenie je dôležité, lebo ukazuje ďalší architektonický kompromis: buď nesiem hodnoty skôr a zaťažím issue logiku, alebo nesiem len odkazy a načítam operandy neskôr, čo kladie iné nároky na plánovanie.

---

18. Paralelné vykonávanie a kompletizácia

Superskalárna architektúra má viac prúdových a vykonávacích jednotiek, takže viaceré inštrukcie môžu byť vykonávané paralelne. Prednáška však zdôrazňuje, že vykonanie nie je to isté ako **kompletizácia**. Inštrukcia môže byť dokončená výpočtovo, ale ešte nemusí byť architektonicky „commitnutá“ do konečného stavu.

Rozlišuje sa:

- **usporiadaná kompletizácia**,
- **preusporiadaná kompletizácia**.

Toto rozlíšenie je kľúčové pre ďalší pojem konzistencie. Prakticky ide o otázku: v akom poradí sa výsledky vykonaných inštrukcií stanú oficiálnou súčasťou stavu procesora.

---

19. Konzistencia vykonávania

Prednáška zavádza dva druhy konzistencie:

Procesorová konzistencia

Týka sa poradia kompletizácie inštrukcií.

- **Slabá procesorová konzistencia** znamená, že dochádza k preusporiadanému poradiu a preusporiadanej kompletizácii bezkonfliktného prúdu inštrukcií. To zvyšuje výkon, ale vyžaduje detekciu hazardov. Ako príklady sa uvádzajú Power1, Power2 a R8000.
- **Silná procesorová konzistencia** znamená usporiadané poradie a usporiadanú kompletizáciu inštrukcií. To spravidla vyžaduje ROB alebo DRIS. Ako príklady sa uvádzajú Pentium Pro, AMD 29000 a R10000.

Pamäťová konzistencia

Týka sa poradia prístupov do pamäte.

- **Slabá pamäťová konzistencia** povoľuje preusporiadaný prístup do pamäte pri bezkonfliktnom prúde. Vyžaduje detekciu hazardov. Príkladmi sú PPC 602–620, UltraSPARC a R10000.
- **Silná pamäťová konzistencia** zachováva usporiadaný prístup do pamäte a typicky vyžaduje ROB. Ako príklad sa uvádza PPC 601.

Tieto pojmy sú veľmi dôležité, pretože ukazujú, že vyšší výkon často vzniká práve tým, že procesor porušuje intuitívne poradie vykonávania – ale musí to robiť tak, aby to navonok ostalo korektné.

---

20. Konzistencia spracovania výnimiek

Pri výnimkách a prerušeniach je dôležité, či je ich poradie jednoznačne definované.

Slabá konzistencia výnimiek

Poradie spracovania výnimiek nie je jasne definované. Príklady: Power1 a Alpha procesory.

Silná konzistencia výnimiek

Poradie spracovania výnimiek je definované. Príklady: Pentium a R10000.

Táto časť nadväzuje na predchádzajúcu tému. Procesor môže vykonávať inštrukcie veľmi agresívne a paralelne, ale pri chybách, prerušení alebo výnimkách musí vedieť poskytnúť konzistentný a zmysluplný stav. Práve preto je otázka commitovania a ROB taká dôležitá.

---

21. VLIW architektúry

VLIW procesory predstavujú druhú veľkú vetvu ILP. Ich podstata je v tom, že jedno veľmi dlhé inštrukčné slovo obsahuje viacero operácií, ktoré sa majú vykonať paralelne. Tým sa veľká časť plánovania presúva z hardvéru na kompilátor.

Prednáška uvádza príklady formátov:

- ELI 512 – 512 bitov,
- TRACE 7/200 – 256 bitov,
- TRACE 28/200 – 4×256 bitov,
- Intel Itanium – 128 bitov.

Dôležité vlastnosti VLIW:

- veľký počet vykonávacích jednotiek,
- dlhé inštrukčné formáty,
- vyššie nároky na pamäť a zbernicu,
- preferencia **statického plánovania** vykonávania.

Prednáška priamo zdôrazňuje, že VLIW nie je manuálne programovateľné na úrovni assembleru v bežnom zmysle. Potrebuje silný kompilátor, ktorý nájde paralelizmus a správne ho zabuduje do dlhých inštrukčných slov.

---

22. VLIW vs. superskalárny procesor

Najdôležitejší rozdiel možno zhrnúť takto:

Superskalárny procesor

- plánovanie paralelizmu robí najmä hardvér,
- typicky používa dynamické plánovanie,
- potrebuje komplikovanejšiu logiku výberu, hazardov a preusporiadania.

VLIW procesor

- paralelizmus plánuje kompilátor,
- hardvér je odľahčený od zložitého dynamického plánovania,
- ale cena sa prenáša na prekladač a často aj na väčší objem inštrukčného kódu.

Teda opäť nejde o to, že jeden prístup je „paralelný“ a druhý nie. Rozdiel je v tom, **kde sídli inteligencia plánovania**.

---

23. Intel Itanium ako príklad VLIW/EPIC

Prednáška používa Itanium ako hlavný príklad modernejšieho VLIW smeru, presnejšie filozofie **EPIC – Explicit Parallel Instruction Computing**. Pri EPIC má kompilátor explicitne rozhodovať, ktoré inštrukcie možno vykonať paralelne. Hardvér tak nemusí robiť také zložité dynamické plánovanie ako superskalárne procesory.

Prednáška uvádza pri Itaniu napríklad:

- 128 celočíselných registrov,
- 128 registrov pre operácie v pohyblivej rádovej čiarke,
- 64 jednobitových prediktorov,
- 8 branch registrov.

Ďalej uvádza veľký počet funkčných jednotiek, napríklad:

- viac ALU,
- celočíselné jednotky,
- posuvné jednotky,
- multimediálne jednotky,
- násobičky,
- jednotky vetvenia.

Z pamäťového hľadiska sa spomína:

- L1 I-cache a D-cache,
- spoločná L2 cache,
- spoločná L3 cache,
- a široká zbernicová architektúra.

Tento príklad má ukázať, že VLIW/EPIC nie je len teória. Je to reálny pokus postaviť veľmi široký paralelný procesor, ktorého paralelizmus je explicitne naplánovaný prekladačom.

Prednáška 8: TLP architektúry

1. Čo sú TLP architektúry

TLP znamená **Thread-Level Parallelism**, teda paralelizmus na úrovni **vlákien a procesov**. Kým v predchádzajúcej téme o ILP išlo hlavne o jemnozrnný paralelizmus vo vnútri jedného inštrukčného prúdu, pri TLP už ide o strednozrnný a hrubozrnný paralelizmus, teda o to, ako spracovávať viac vlákien alebo viac procesov naraz. Inými slovami: ILP sa pýta, ako vyťažiť viac z jedného vlákna, TLP sa pýta, ako efektívne spracovať viac vlákien súčasne.

Prednáška hneď na úvode zdôrazňuje, že hrubozrnný paralelizmus sa dnes využíva najmä dvoma hlavnými spôsobmi. Prvým sú **multiprocesorové čipy**, teda čipy, na ktorých je integrovaných viac procesorových jadier. Druhým sú **multivláknové procesory**, ktoré prekrývajú vykonávanie inštrukcií z viacerých vlákien, často aj v rámci jedného jadra. TLP teda nie je jedna konkrétna technika, ale širšia trieda architektúr, ktoré sa snažia využiť viacero vlákien alebo procesov naraz.

---

2. Prečo TLP vôbec vzniklo

Hlavná motivácia pre vznik TLP architektúr bola veľmi praktická: **latencia pamäte**. Procesory sa zrýchľovali rýchlejšie než pamäťový systém, a preto sa čoraz častejšie stávalo, že procesor musel čakať na dáta. Ak architektúra vie počas čakania jedného vlákna spustiť alebo pokračovať v inom vlákne, vie lepšie využiť výpočtové jednotky a znížiť nečinnosť procesora. Práve preto prednáška hovorí, že viacvláknový prístup ponúkol stratégiu tolerancie odozvy, hlavne v multiprocesorových systémoch.

To je veľmi dôležitý rozdiel oproti ILP. Pri ILP sa snažíš nájsť paralelizmus medzi inštrukciami jedného vlákna. Pri TLP sa zmieruješ s tým, že jedno vlákno môže byť často blokované, a preto si pomáhaš tým, že máš pripravené ďalšie vlákna. TLP je teda do veľkej miery odpoveď na to, že v reálnych systémoch sa veľmi často čaká na pamäť, na komunikáciu alebo na synchronizáciu.

---

3. Základné smery TLP architektúr

Prednáška rozlišuje dve veľké vetvy využitia TLP:

a) Multiprocesorové čipy

Tu ide o architektúry, kde je na jednom čipe viac procesorov alebo jadier. Prednáška sem zaraďuje:

- **SMS** – symetrické multiprocesory, ktoré zodpovedajú modelu **UMA**,
- **DSM** – distribuované multiprocesory so zdieľanou pamäťou, teda model **NUMA**,
- **MPP** – multiprocesory s odovzdávaním správ.

b) Multivláknové procesory

Tu ide o architektúry, kde sa viaceré vlákna vykonávajú v tom istom procesore alebo jadre. Prednáška sem zaraďuje viac modelov:

- po cykloch prekrývaný model,
- blokovo prekrývaný model,
- simultánny multivláknový model,
- nanovláknový a mikrovláknový model,
- model VLIW,
- model multiprocesorového čipu.

Najlepšie je zapamätať si to takto: **jedna vetva TLP pridáva jadrá, druhá vetva pridáva vlákna na jadro**. V praxi sa tieto dva prístupy často kombinujú.

---

4. Výpočtové modely TLP architektúr

Prednáška sa nesústredí iba na hardvér, ale aj na to, **ako si výpočet konceptuálne predstavujeme**. Rozlišuje tri základné modely:

- **CF model** – Control Flow, teda klasický Neumannov model,
- **DF model** – Data Flow, teda model toku dát,
- **hybridný CF/DF model** – kombinácia oboch prístupov.

Toto delenie je dôležité, lebo vysvetľuje, ako môže architektúra „premýšľať“ o programe: buď je riadená hlavne poradím inštrukcií, alebo dostupnosťou dát, alebo kombináciou oboch.

---

5. CF model – klasický riadiaci tok

CF model je klasický model sekvenčného riadenia. Program sa vykonáva podľa poradia inštrukcií, ktoré určuje programové počítadlo. Inštrukcie sú uložené v inštrukčnej pamäti a

údaje v pamäti alebo v registroch. Výpočet je teda riadený najmä tým, **ktorá inštrukcia nasleduje po ktorej**.

Tento model je prirodzený a intuitívny, pretože zodpovedá tomu, ako sa bežne rozmýšľa o programe: najprv urob toto, potom tamto. Je to základ klasických procesorov. Pri TLP je však dôležité uvedomiť si, že ak chceš mať viac vlákien, CF model stále funguje, ale naraz môže existovať viac riadiacich tokov. Teda nejde o to, že CF zmizne, ale že sa **rozmnoží počet súčasne existujúcich riadiacich sekvencií**.

---

6. DF model – riadenie tokom dát

DF model funguje inak. Tu už nie je rozhodujúce poradie inštrukcií v programe, ale to, **či sú pripravené všetky potrebné operandy**. Inštrukcia sa vykoná vtedy a len vtedy, keď má dostupné všetky vstupy. Výpočet sa preto popisuje cez **graf toku dát (DFG)**.

Prednáška zároveň zdôrazňuje, že v čistom DF modeli neexistujú spoločné prepisovateľné pamäťové elementy tak, ako v klasickom von Neumannovom modeli. To je zásadná vlastnosť. Znamená to, že výpočet nie je organizovaný okolo jedného spoločného stavu pamäte, ale okolo závislostí medzi operáciami. Praktická výhoda je v tom, že paralelizmus je v takomto modeli často prirodzenejší. Nevýhoda je, že takýto model je ťažší na priamu implementáciu a programovanie.

Prednáška tiež spomína, že uzly DFG sa môžu zoskupovať do podgrafov s nízkym stupňom paralelizmu a tieto podgrafy sa potom vykonávajú sekvenčne. To je dôležitá praktická myšlienka: ani v dataflow svete sa nemusí všetko vykonávať naraz. To, čo nemá dostatočný paralelizmus, sa často zabalí do sekvenčnej časti.

---

7. Hybridný CF/DF model

Hybridný model predstavuje prienik klasického riadiaceho modelu a modelu toku dát. Prednáška rozlišuje dve verzie:

- hybridný model založený na **riadiacich operátoroch (RO)**,
- hybridný model založený na **riadiacich tokenoch (RT)**.

Zmysel hybridného modelu je v tom, že sa snaží zobrať výhody oboch svetov. Z CF modelu si berie prirodzené sekvenčné riadenie a jednoduchší spôsob rozmýšľania o programe. Z DF modelu si berie schopnosť aktivovať časti výpočtu podľa dostupnosti potrebných informácií. Hybridný prístup je preto pokusom spojiť praktickosť von Neumanna s paralelizmom dataflow sveta.

---

8. Hybridný model s riadiacimi operátormi (RO)

V modeli RO majú paralelné riadiace operátory podobnú úlohu ako riadiace inštrukcie v klasickom CF modeli, ale sú silnejšie. Kým bežná riadiaca inštrukcia len určuje ďalší krok programu, riadiaci operátor vie **rozdeliť sekvenčné vykonávanie do viacerých vlákien**.

Prednáška uvádza dve kľúčové inštrukcie:

- **FORK** – vytvorí nové vlákno a určí jeho prvú inštrukciu,
- **JOIN** – spojí dve vlákna do jedného pokračujúceho toku.

Tento model je veľmi dôležitý na pochopenie hrubozrnného paralelizmu. Program sa správa ako klasický program, ale v určitých bodoch sa rozdelí na viac vlákien a neskôr sa opäť spojí. Vnútri každého vlákna sa pritom stále vykonáva klasický CF model. Komunikácia medzi vláknami prebieha cez spoločné registre alebo spoločnú pamäť, a preto treba explicitnú synchronizáciu, napríklad semafory.

Prednáška ďalej upozorňuje, že pri viacerých vláknach už viac rôznych inštrukcií môže naraz pristupovať k tým istým dátam. Preto vznikajú synchronizačné problémy a treba ich riešiť explicitne. Pri hrubozrnných architektúrach sa dajú registre optimalizovať zvlášť pre každé vlákno. Pri jemnozrnných architektúrach, kde môže súčasne existovať veľké množstvo vlákien, by to vyžadovalo veľmi veľké registrové súbory.

---

9. Hybridný model s riadiacimi tokenmi (RT)

V modeli RT sa tok vykonávania riadi pomocou **grafu údajových závislostí (DDG)**. Na hranách grafu sa nepohybujú klasické dáta, ale **riadiace tokeny**. Inštrukcia je pripravená na vykonanie vtedy, keď má na všetkých vstupných hranách potrebné tokeny.

Prednáška výslovne hovorí, že tento prístup má silnú analógiu s dataflow modelom, ale riadenie toku inštrukcií tu prebieha nezávisle od samotného toku dát. To je veľmi dôležitý rozdiel. Dáta a riadenie sú v tomto modeli oddelené ešte výraznejšie než v modeli RO.

Inštrukčný rámec klasického CF modelu sa tu rozširuje o dva nové rámce:

- rámec riadiacich tokenov,
- rámec adries operátorov, kam sa po vykonaní posielajú ďalšie tokeny.

Dá sa to chápať tak, že v tomto modeli už programové počítadlo nehrá hlavnú úlohu. Namiesto neho sa tok riadi tým, **kde sa objavia správne tokeny**. To dáva väčšiu flexibilitu pri paralelnom vykonávaní, ale robí architektúru koncepčne zložitejšou.

---

10. Multivláknové architektúry – hlavná myšlienka

Prednáška potom prechádza k praktickým multivláknovým architektúram. Tie sú definované ako architektúry, kde sa inštrukcie vykonávajú vo vláknach skalárne, superskalárne alebo podľa princípu VLIW. Hlavná motivácia týchto architektúr je opäť tolerancia latencie, najmä latencie pamäte.

Treba si zapamätať, že multivláknovanie neznamená automaticky to isté čo viacjadrovosť. Viacjadrovosť znamená viac fyzických jadier. Multivláknovanie znamená, že jedno jadro alebo procesor vie **udržiavať a obsluhovať viac vlákien**. Obe techniky sa často kombinujú, ale nie sú totožné.

---

11. Aspekty návrhu multivláknových architektúr

Prednáška uvádza štyri hlavné návrhové aspekty:

- **výpočtový model**,
- **granularita**,
- **pamäťová hierarchia**,
- **počet vlákien na procesor**.

Výpočtový model

Architektúra môže byť viac neumannovská alebo viac hybridná CF/DF. To ovplyvňuje spôsob riadenia výpočtu a prácu s vláknami.

Granularita

Prednáška rozlišuje jemnozrnnú a hrubozrnnú granularitu. Pri jemnozrnnom prístupe sa medzi vláknami prepína veľmi často, niekedy po každom cykle. Pri hrubozrnnom sa prepína menej často, typicky pri dlhšom čakaní alebo po väčších blokoch práce.

Pamäťová hierarchia

Uvádzajú sa tri varianty:

- spoločná pamäť,
- zdieľaná spoločná pamäť,
- zdieľaná koherentná spoločná rýchla asociatívna pamäť.

Počet vlákien na procesor

Prednáška rozlišuje:

- malý počet vlákien: približne 4–10,

- stredný: 10–100,
- veľký: 100 a viac.

Toto delenie je dôležité, pretože priamo ovplyvňuje návrh registrov, plánovania aj pamäťového systému. Čím viac vlákien chce procesor držať aktívnych, tým väčšia je režijnosť ich správy, ale zároveň tým väčšia je šanca, že sa podarí prekryť latencie.

---

12. Hardvérové modely multivláknových architektúr

Prednáška uvádza tieto základné modely:

- po cykloch prekrývaný model,
- blokovo prekrývaný model,
- simultánny multivláknový model,
- nanovláknový a mikrovláknový model,
- model VLIW,
- model multiprocesorového čipu.

Aby sa to ľahšie učilo, dá sa to chápať ako odpoveď na otázku: **kedy a ako často prepínam medzi vláknami a či ich viem spracovávať aj súbežne v tom istom cykle**.

---

13. Po cykloch prekrývaný model

Pri po cykloch prekrývanom modeli sa vlákna striedajú **pravidelne po jednotlivých cykloch**. V jednom cykle sa vykoná inštrukcia z jedného vlákna, v ďalšom cykle z ďalšieho, a tak ďalej cyklicky. Prednáška to zapisuje ako pravidelné striedanie inštrukcií I11,I21,...,In1,I12,I22,...I_{11}, I_{21}, ..., I_{n1}, I_{12}, I_{22}, ...I11,I21,...,In1,I12,I22,....

Tento model je vhodný na jemnozrnnú toleranciu latencie. Ak jedno vlákno čaká, procesor už má pripravené ďalšie. Nevýhodou je, že jedno konkrétne vlákno dostane procesor len každých niekoľko cyklov, takže jeho individuálny progres môže byť pomalší. Výhodou je vysoká schopnosť prekryť dlhé latencie.

Dá sa to zapamätať jednoducho: **cyklický model = pravidelné striedanie vlákien po taktoch**.

---

14. Blokovo prekrývaný model

Pri blokovo prekrývanom modeli sa nevykonáva jedno vlákno po jednom takte. Namiesto toho sa spracúva určitý **blok inštrukcií jedného vlákna**, potom blok inštrukcií ďalšieho vlákna

a podobne. Prednáška to ilustruje zápisom typu I11,I12,...,I1n1,I21,...I_{11}, I_{12}, ..., I_{1n_1}, I_{21}, ...I11,I12,...,I1n1,I21,....

Tento model je hrubozrnnejší než cyklické prekrývanie. Jeho výhoda je menšia réžia prepínania. Nehodí sa však tak dobre na veľmi jemné skrývanie krátkych latencií. Najväčší zmysel má vtedy, keď sa prepína medzi vláknami pri dlhších udalostiach, napríklad pri čakaní na pamäť alebo pri inej výraznejšej prekážke.

Dá sa to zapamätať ako: **blokový model = chvíľu ide jedno vlákno, potom dlhší kus ďalšie**.

---

15. Simultánny multithreading (SMT)

SMT je jeden z najdôležitejších modelov TLP architektúr. Jeho podstata je v tom, že procesor vie v tom istom čase využívať zdroje pre **viac vlákien naraz**. To znamená, že v jednom takte nemusí len striedať vlákna, ale vie dokonca vydať inštrukcie z viacerých vlákien do rôznych vykonávacích jednotiek.

Z pohľadu učenia je dôležité odlíšiť SMT od obyčajného prepínania vlákien:

- pri jednoduchom prepínaní vykonávaš v jednom okamihu jedno vlákno a len sa rýchlo prepínaš,
- pri SMT môžeš mať **skutočnú súbežnosť viacerých vlákien v tom istom jadre**.

To je dôvod, prečo je SMT veľmi silné v kombinácii so superskalárnym procesorom. Ak má jadro viac vykonávacích jednotiek a jedno vlákno ich nevie všetky zaplniť, druhé vlákno môže využiť voľné miesto. Cena za to je zložitejšie plánovanie a súťaženie o zdieľané zdroje.

---

16. Nanovláknový a mikrovláknový model

Prednáška tieto modely zaraďuje medzi pokročilejšie varianty multivláknových architektúr a opisuje ich ako:

- blokovo prekrývanú superskalárnu verziu,
- simultánnu superskalárnu verziu.

Hlavná idea je, že sa kombinujú vlastnosti superskalárneho spracovania a viacvláknového prekrývania na ešte jemnejšej alebo špecializovanejšej úrovni. Pre študijné účely si ich stačí zaradiť ako **špeciálne hybridné formy multivláknovania**, ktoré idú ešte ďalej než klasický coarse-grained model alebo čisté SMT. Prednáška ich nespresňuje tak detailne ako CF/DF alebo SMT, preto pri učení dáva zmysel pamätať si skôr ich miesto v klasifikácii než sa snažiť naspamäť reprodukovať každý detail.

---

17. Model VLIW v TLP kontexte

Prednáška uvádza aj **model VLIW** ako jednu z možností TLP architektúr. Zmysel je v tom, že vláknová úroveň paralelizmu sa môže kombinovať s architektúrou, ktorá už sama o sebe využíva viacnásobné paralelné vykonávanie v rámci jedného inštrukčného slova.

Z praktického hľadiska si to môžeš predstaviť tak, že VLIW rieši paralelizmus vnútri inštrukčného slova a TLP rieši paralelizmus medzi vláknami. Tieto dve vrstvy sa teda nevylučujú, ale môžu sa kombinovať.

---

18. Model multiprocesorového čipu

Tento model predstavuje integráciu viacerých procesorov alebo jadier na jednom čipe. Prednáška ho uvádza ako súčasť klasifikácie TLP architektúr, pretože ide o veľmi dôležitý spôsob využitia hrubozrnného paralelizmu.

Najjednoduchšie sa to dá zapamätať takto:

- **multivláknovanie** pridáva viac vlákien na jedno jadro,
- **multiprocesorový čip** pridáva viac jadier na jeden čip,
- moderné procesory často robia oboje naraz.

---

19. Príklady architektúr spomenutých v prednáške

Prednáška uvádza viaceré historické aj praktické príklady. Ich význam je hlavne ilustračný – ukazujú, že TLP nie je len teória.

Výpočtové a hybridné modely

- **HEP (1982)** – uvádza sa ako prvý komerčný MIMD systém,
- **Tera**,
- **MIT Alewife a Sparcle**,
- **P-RISC, USC, MIT Hybrid Machine**,
- **McGill MGDA & SAM**,
- **EM-4**.

SMT a multivláknové procesory

- **Clearwater Networks CNP810SP (2000)** – sieť procesorov s podporou SMT, 8 vlákien,
- **MLX1 (2002)** – spomínaný ako start-up projekt,
- **Alpha 21464 (EV8)** – podpora SMT, jedno jadro, 4 vlákna,
- **Intel HyperThreading (2002)** – 2 vlákna,

- **Intel Atom (2008)** – 2-cestný SMT,
- **Intel Itanium Tukwila** – 2-cestný SMT,
- **Intel Xeon Phi (2012)** – 4-cestný SMT,
- **SPARC M8 (2017)** – 32 jadier, 8 vlákien na jadro.

Tieto príklady si netreba pamätať ako holý zoznam rokov. Dôležitejšie je pochopiť, že:

- SMT sa objavuje v rôznych líniách procesorov,
- počet vlákien na jadro sa líši,
- a TLP sa v praxi kombinuje s viacjadrovosťou.

---

20. Intel mikroarchitektúry v prednáške

Prednáška pridáva aj stručný prehľad Intel rodín:

- **Atom** – nízka spotreba, notebooky, vložené systémy, IoT,
- **Core** – pracovné stanice a bežné výkonné PC,
- **Xeon** – servery,
- **Itanium** – podnikové servery a vysokovýkonné systémy.

Spomína sa aj známy model vývoja **Tick-Tock**:

- **TICK** – zmena výrobného procesu pri rovnakej architektúre,
- **TOCK** – nová architektúra pri tom istom výrobnom procese.

Toto nie je jadro teórie TLP, ale pomáha pochopiť, v akom priemyselnom kontexte sa viacjadrovosť a multivláknovanie vyvíjali.

---

21. Intel Core i7 Nehalem ako konkrétny príklad TLP

Prednáška používa **Intel Core i7 Nehalem** ako konkrétny príklad viacjadrovej a viacvláknovej architektúry. Pri modeli i7 920 sa uvádza:

- 1. generácia Nehalem,
- 4 jadrá,
- 45 nm výrobný proces,
- veľkosť čipu 263 mm²,
- 731 miliónov tranzistorov.

Architektonicky je dôležité najmä toto:

- procesor je **4-jadrový**,

- každé jadro podporuje **2 vlákna**,
- plánovanie vlákien je **dynamické** cez SMT/HyperThreading,
- niektoré zdroje sú zdieľané – napríklad **ROB**, vykonávacie jednotky a cache.

Prednáška zároveň zdôrazňuje, že i7 má „širokú“ prúdovú funkčnú jednotku:

- 6 vykonávacích jednotiek,
- z toho 3 pre aritmeticko-logické operácie a 3 pre pamäťové operácie.

Pri cache sa uvádza:

- **L1**: 32 KB I-cache a 32 KB D-cache na jadro, latencia 4 cykly,
- **L2**: 256 KB unifikovaná na jadro, latencia 10 cyklov,
- **L3**: zdieľaná, latencia 35 cyklov.

Tento príklad je veľmi dobrý na pochopenie modernej reality: TLP v praxi neznamená jeden „čistý“ model, ale kombináciu:

- viac jadier,
- viac vlákien na jadro,
- zdieľaných a nezdieľaných zdrojov,
- a dynamického plánovania.

---

22. Ako si zapamätať rozdiel medzi hlavnými TLP modelmi

Na učenie sa oplatí mať v hlave jednoduché rozlíšenie:

- **CF model**: vykonávanie riadi poradie inštrukcií.
- **DF model**: vykonávanie riadi dostupnosť operandov.
- **Hybridný RO model**: CF vo vláknach, ale vieš vlákna rozdeľovať a spájať cez FORK/JOIN.
- **Hybridný RT model**: tok vykonávania určujú riadiace tokeny v grafe závislostí.
- **Cyklické prekrývanie**: vlákna sa striedajú po cykloch.
- **Blokové prekrývanie**: vykonáva sa blok inštrukcií jedného vlákna, potom ďalšieho.
- **SMT**: viac vlákien využíva to isté jadro súčasne v tom istom období.
- **CMP / multiprocesorový čip**: na čipe je viac jadier.

Toto je asi najpraktickejší „kostrový“ prehľad celej prednášky. Ak ho chápeš, detaily sa potom ukladajú ľahšie.

---

23. Hlavná myšlienka celej prednášky

Prednáška 8 chce ukázať, že keď už nestačí paralelizmus na úrovni inštrukcií, výkon sa ďalej zvyšuje cez **paralelizmus vlákien a procesov**. TLP architektúry preto pracujú s viacerými vláknami, aby:

- lepšie využili procesor,
- prekryli pamäťové latencie,
- a zvýšili celkovú priepustnosť systému.

Najdôležitejšie veci, ktoré si z tejto prednášky treba odniesť, sú:

- TLP je paralelizmus na úrovni vlákien a procesov, nie inštrukcií,
- vznikol hlavne ako reakcia na dlhé latencie pamäte,
- základné výpočtové modely sú CF, DF a hybridný CF/DF,
- kľúčové multivláknové modely sú cyklické prekrývanie, blokové prekrývanie a SMT,
- moderné procesory často kombinujú viacjadrovosť aj multivláknovanie,
- TLP je v praxi kompromis medzi počtom vlákien, granularitou, pamäťovou hierarchiou a zdieľaním zdrojov.

Prednáška 9: Pthreads

1. Čo je cieľom prednášky

Prednáška 9 sa venuje programovaniu paralelných programov v prostredí **zdieľanej pamäte** pomocou **Pthreads**. Ide teda o situáciu, keď viac vlákien beží v rámci jedného procesu a tieto vlákna môžu pristupovať k spoločným dátam. To je výkonné, ale aj nebezpečné, pretože ak viac vlákien naraz mení rovnakú premennú alebo dátovú štruktúru, môžu vzniknúť chyby, ktoré sa objavujú nepravidelne a ťažko sa hľadajú.

Hlavné témy prednášky sú:

- procesy a vlákna,
- POSIX threads, teda Pthreads,
- vytváranie a ukončovanie vlákien,
- globálne a lokálne premenné vo vláknach,
- násobenie matice a vektora pomocou vlákien,
- kritické sekcie,
- race condition,
- busy-waiting,
- mutexy,
- semafory,
- bariéry,
- condition variables,
- read-write locks,
- cache coherence, false sharing,
- thread-safety.

---

2. Zdieľaná pamäť a vlákna

V systéme so zdieľanou pamäťou majú viaceré výpočtové jednotky alebo jadrá prístup k tej istej hlavnej pamäti. Programátor preto nemusí explicitne posielať správy medzi procesormi, ako je to napríklad pri distribuovanej pamäti. Namiesto toho môžu vlákna komunikovať cez spoločné premenné.

To je pohodlné, ale prináša to hlavný problém celej prednášky: **ak viaceré vlákna pristupujú k tej istej zdieľanej premennej, treba presne riadiť, kto a kedy ju číta alebo mení**.

Proces je jedna bežiaca alebo pozastavená inštancia programu. Vlákno je odľahčená forma procesu. V rámci jedného procesu môže existovať viacero vlákien riadenia. Tieto vlákna zdieľajú globálne premenné, haldu a ďalšie časti adresného priestoru procesu, ale každé vlákno má vlastný zásobník a vlastné lokálne premenné vo funkciách, ktoré práve vykonáva.

Jednoducho:

- **proces** = bežiaci program s vlastným adresným priestorom,
- **vlákno** = samostatná vetva vykonávania vo vnútri procesu,
- **globálne premenné** = spoločné pre všetky vlákna,
- **lokálne premenné funkcie** = typicky súkromné pre konkrétne vlákno.

---

3. Čo sú POSIX Threads / Pthreads

Pthreads je skratka pre **POSIX Threads**. Je to štandardné API pre viacvláknové programovanie v systémoch podobných Unixu. Používa sa hlavne v jazyku C a umožňuje explicitne vytvárať, riadiť a synchronizovať vlákna.

Pthreads je dostupné najmä na systémoch ako:

- Linux,
- macOS,
- Solaris,
- HPUX,
- ďalšie POSIX-kompatibilné systémy.

V programe treba použiť hlavičkový súbor:

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

Pri preklade treba pripojiť knižnicu pthread, napríklad:

```
gcc -g -Wall -o pth_hello pth_hello.c -lpthread
```

alebo v niektorých systémoch:

```
gcc -g -Wall -pthread -o pth_hello pth_hello.c
```

Dôležité je, že Pthreads nie je automatický paralelizmus. Programátor musí v kóde explicitne povedať, koľko vlákien chce vytvoriť, akú funkciu majú vlákna vykonávať a ako sa majú synchronizovať.

---

4. Globálne premenné vo viacvláknovom programe

Globálne premenné sú vo vláknach zdieľané. To znamená, že každé vlákno k nim môže pristupovať. Toto je výhodné, keď chceme, aby vlákna spolupracovali nad spoločnými dátami, ale zároveň je to častý zdroj chýb.

Ak dve vlákna súčasne zapisujú do tej istej globálnej premennej alebo jedno vlákno číta hodnotu, ktorú druhé práve mení, výsledok nemusí byť správny. Preto prednáška upozorňuje, že globálne premenné treba používať opatrne a iba tam, kde sú naozaj potrebné.

Príklad problému:

```
x = x + y;
```

Táto operácia vyzerá ako jeden jednoduchý krok, ale v skutočnosti sa skladá z viacerých krokov:

1. načítaj hodnotu `x`,
2. pripočítaj `y`,
3. zapíš výsledok späť do `x`.

Ak toto robia dve vlákna naraz, môže sa stať, že obe načítajú rovnakú pôvodnú hodnotu `x`, obe vypočítajú výsledok a jedno prepíše výsledok druhého. Tak vzniká chyba.

---

5. Vytváranie vlákien pomocou `pthread_create`

Na vytvorenie vlákna sa používa funkcia:

```
int pthread_create(
   pthread_t* thread_p,
   const pthread_attr_t* attr_p,
   void* (*start_routine)(void*),
   void* arg_p
);
```

Význam parametrov:

pthread_t* thread_p

Sem sa uloží identifikátor vytvoreného vlákna. Pre každé vlákno potrebujeme jeden objekt typu pthread_t.

Typ pthread_t je nepriehľadný typ. To znamená, že programátor nemá pracovať priamo s jeho vnútornou štruktúrou. Štandard iba zaručuje, že tento objekt obsahuje dosť informácií na jednoznačnú identifikáciu vlákna.

const pthread_attr_t* attr_p

Tento parameter určuje atribúty vlákna. Ak nepotrebujeme špeciálne nastavenia, dávame sem NULL.

void* (*start_routine)(void*)

Toto je funkcia, ktorú bude nové vlákno vykonávať. Každé vlákno po vytvorení začne práve v tejto funkcii.

Funkcia vlákna musí mať tvar:

```
void* thread_function(void* args_p);
```

void* arg_p

Toto je argument, ktorý sa odovzdá funkcii vlákna. Keďže ide o void*, môžeme cez tento ukazovateľ odovzdať rôzne typy dát. Často sa takto odovzdáva číslo vlákna, ukazovateľ na štruktúru s viacerými údajmi alebo rozsah dát, ktoré má vlákno spracovať.

---

6. Ukončenie vlákien pomocou **pthread_join**

Po vytvorení vlákien musí hlavné vlákno často počkať, kým pracovné vlákna skončia. Na to sa používa:

```
pthread_join(thread, NULL);
```

Funkcia pthread_join spôsobí, že volajúce vlákno čaká na dokončenie konkrétneho vlákna. Typicky ju volá hlavné vlákno pre každé vytvorené pracovné vlákno.

Schéma je:

1. hlavné vlákno vytvorí pracovné vlákna,
2. pracovné vlákna vykonávajú svoju časť práce,
3. hlavné vlákno zavolá pthread_join pre každé z nich,
4. až keď všetky vlákna skončia, program pokračuje ďalej alebo sa ukončí.

Bez pthread_join by sa mohlo stať, že hlavné vlákno skončí skôr, než pracovné vlákna dokončia svoju prácu.

---

7. Základný model programu s Pthreads

Typický Pthreads program má takúto štruktúru:

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

void* Thread_work(void* rank);

int main(int argc, char* argv[]) {
   long thread;
   int thread_count;
   pthread_t* thread_handles;

   thread_count = strtol(argv[1], NULL, 10);
   thread_handles = malloc(thread_count * sizeof(pthread_t));

   for (thread = 0; thread < thread_count; thread++) {
      pthread_create(&thread_handles[thread], NULL, Thread_work, (void*) thread);
   }

   for (thread = 0; thread < thread_count; thread++) {
      pthread_join(thread_handles[thread], NULL);
   }

   free(thread_handles);
   return 0;
}

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

Dôležitá myšlienka je, že hlavné vlákno explicitne vytvorí pracovné vlákna. Každému vláknu môže odovzdať identifikátor alebo iné dáta. Potom hlavné vlákno čaká, kým všetky pracovné vlákna skončia.

---

8. Rozdelenie práce medzi vlákna

Pri paralelnom programe nestačí iba vytvoriť viac vlákien. Treba im aj zmysluplne rozdeliť prácu. Prednáška ukazuje príklad násobenia matice a vektora.

Sekvenčne by sme pre každý riadok matice vypočítali jeden prvok výsledného vektora. Pri použití vlákien môžeme rozdeliť riadky matice medzi vlákna.

Napríklad ak máme 3 vlákna:

- vlákno 0 spracuje prvú časť riadkov,
- vlákno 1 spracuje druhú časť riadkov,
- vlákno 2 spracuje tretiu časť riadkov.

Každé vlákno teda počíta inú časť výsledku. Toto je vhodný príklad, pretože jednotlivé riadky matice sa dajú často spracovať nezávisle. Ak každý prvok výsledného vektora zapisuje iba jedno vlákno, nevzniká konflikt pri zápise.

Hlavná zásada pri rozdeľovaní práce:

- hľadať časti výpočtu, ktoré sú nezávislé,
- minimalizovať spoločné zápisy,
- rozdeliť prácu rovnomerne,
- dávať pozor na zdieľané dáta.

---

9. Kritická sekcia

Kritická sekcia je časť programu, v ktorej sa pristupuje k zdieľanému zdroju tak, že súčasný prístup viacerých vlákien by mohol spôsobiť chybu.

Zdieľaným zdrojom môže byť napríklad:

- globálna premenná,
- zdieľané pole,
- zdieľaný súbor,
- zreťazený zoznam,
- front,
- počítadlo,
- výstup na obrazovku.

Príklad kritickej sekcie:

```
global_sum += local_sum;
```

Ak túto operáciu vykoná viac vlákien naraz, môže vzniknúť nesprávny výsledok. Preto treba zabezpečiť, aby do tejto časti v jednom okamihu vstúpilo iba jedno vlákno.

---

10. Race condition

Race condition vzniká vtedy, keď výsledok programu závisí od nepredvídateľného poradia vykonávania vlákien. Typicky ide o situáciu, keď viac vlákien pristupuje k rovnakému zdieľanému zdroju a aspoň jedno z nich tento zdroj mení.

Príklad:

```
y = Compute(my_rank);
x = x + y;
```

Premenná `y` je súkromná pre každé vlákno, ale `x` je spoločná. Ak dve vlákna naraz vykonávajú `x = x + y`, výsledok môže byť zlý.

Prečo?

Vlákno 0 môže načítať `x = 0`.

Vlákno 1 môže tiež načítať `x = 0`.

Vlákno 0 vypočíta `x + y0` a zapíše výsledok.

Vlákno 1 vypočíta `x + y1` z pôvodnej hodnoty a prepíše výsledok vlákna 0.

Výsledkom je, že jedna aktualizácia sa stratí.

Race condition je nebezpečná najmä preto, že sa nemusí prejaviť vždy. Program môže niekedy fungovať správne a inokedy nie, podľa toho, ako operačný systém naplánuje vlákna.

---

11. Busy-waiting

Busy-waiting znamená, že vlákno opakovane kontroluje určitú podmienku, ale medzitým nerobí žiadnu užitočnú prácu.

Príklad:

```
while (flag != my_rank);
```

Vlákno v tomto cykle čaká, kým premenná `flag` nebude mať požadovanú hodnotu. Problém je, že počas čakania stále spotrebúva CPU čas.

Busy-waiting teda môže zabezpečiť poradie prístupu ku kritickej sekcii, ale je neefektívne. Prednáška ukazuje, že pri výpočte π môže busy-waiting viesť k horšiemu výkonu než sekvenčný program, pretože vlákna veľa času iba čakajú a zaťažujú procesor.

Ďalší problém je optimalizujúci kompilátor. Ak kompilátor zistí, že telo cyklu je prázdne, môže sa pokúsiť kód optimalizovať spôsobom, ktorý naruší očakávané správanie. Preto je busy-waiting nielen neefektívny, ale môže byť aj nespoľahlivý.

---

12. Lepšia organizácia kritickej sekcie

Pri paralelnom výpočte je veľmi dôležité, aby kritická sekcia bola čo najkratšia. Ak vlákna vstupujú do kritickej sekcie často, veľká časť programu sa vykonáva sériovo a paralelizmus sa stráca.

Zlý prístup:

```
for (...) {
   global_sum += nieco;
}
```

Ak je global_sum zdieľaná premenná, každá iterácia potrebuje synchronizáciu.

Lepší prístup:

```
local_sum = 0;

for (...) {
   local_sum += nieco;
}

global_sum += local_sum;
```

Tu každé vlákno najprv počíta do svojej lokálnej premennej. Až na konci raz vstúpi do kritickej sekcie a pripočíta svoj výsledok do globálnej sumy.

Toto je veľmi dôležitý princíp: **najprv počítať lokálne, potom krátko synchronizovať globálny výsledok**.

---

13. Mutex

Mutex znamená **mutual exclusion**, teda vzájomné vylúčenie. Je to špeciálny synchronizačný objekt, ktorý zabezpečí, že kritickú sekciu môže naraz vykonávať iba jedno vlákno.

V Pthreads sa používa typ:

```
pthread_mutex_t
```

Základné operácie sú:

```
pthread_mutex_init(&mutex, NULL);
pthread_mutex_lock(&mutex);
pthread_mutex_unlock(&mutex);
pthread_mutex_destroy(&mutex);
```

Typické použitie:

```
pthread_mutex_lock(&mutex);

global_sum += local_sum;

pthread_mutex_unlock(&mutex);
```

Keď jedno vlákno zavolá `pthread_mutex_lock`, získa zámok. Ak iné vlákno skúsi získať ten istý zámok, musí čakať. Keď prvé vlákno skončí kritickú sekciu, zavolá `pthread_mutex_unlock` a tým zámok uvoľní.

Mutex je lepší než busy-waiting, pretože vlákno, ktoré nemôže vstúpiť do kritickej sekcie, nemusí aktívne páliť CPU cykly prázdnym čakaním.

---

14. Kedy mutex pomáha a kedy škodí výkonu

Mutex je potrebný na korektnosť, ale nesmie sa používať zbytočne často. Ak celý veľký výpočet obalíme jedným mutexom, program bude síce správny, ale prakticky sériový.

Dobré použitie mutexu:

- iba okolo krátkej kritickej sekcie,
- pri zápise do spoločného výsledku,
- pri aktualizácii spoločnej dátovej štruktúry,
- pri ochrane zdieľaného počítadla.

Zlé použitie mutexu:

- okolo celého výpočtu,
- v každej iterácii veľkého cyklu, ak to nie je nevyhnutné,
- okolo čítania dát, ktoré sa nemenia,
- tam, kde každé vlákno môže pracovať s vlastnou lokálnou kópiou.

Treba si zapamätať: **mutex zvyšuje korektnosť, ale príliš veľa mutexov alebo príliš veľké kritické sekcie znižujú paralelný výkon**.

---

15. Producent–konzument a potreba poradia

Nie vždy nám stačí iba zabezpečiť, že do kritickej sekcie vstúpi naraz jedno vlákno. Niekedy potrebujeme riadiť aj **poradie**, v akom vlákna pokračujú.

Typický príklad je problém **producent–konzument**:

- producent vytvorí dáta,
- konzument ich spracuje,
- konzument nesmie čítať dáta skôr, než ich producent pripraví.

Mutex sám o sebe zaručuje vzájomné vylúčenie, ale neurčuje presné poradie udalostí. Preto prednáška zavádza semafory ako nástroj, ktorý vie okrem ochrany kritickej sekcie pomôcť aj pri riadení poradia medzi vláknami.

---

16. Semafory

Semafor je synchronizačný objekt, ktorý si môžeme predstaviť ako celočíselný počítadlový zámok. Má hodnotu a dve základné operácie:

- sem_wait,
- sem_post.

Semafory nie sú priamo súčasťou Pthreads, preto treba pridať:

```
#include <semaphore.h>
```

Základné funkcie:

```
sem_init(&sem, 0, initial_value);
sem_wait(&sem);
sem_post(&sem);
sem_destroy(&sem);
```

sem_wait

Zníži hodnotu semafora. Ak je hodnota semafora 0, vlákno čaká.

sem_post

Zvýši hodnotu semafora a prípadne zobudí čakajúce vlákno.

Semafor je všeobecnejší ako mutex, pretože mutex má logiku „zamknuté/odomknuté“, zatiaľ čo semafor môže mať ľubovoľnú nezápornú hodnotu. Preto sa dá použiť nielen na vzájomné vylúčenie, ale aj na signalizáciu medzi vláknami.

---

17. Rozdiel medzi mutexom a semaforom

Mutex je najvhodnejší vtedy, keď chceme ochrániť kritickú sekciu:

```
pthread_mutex_lock(&mutex);
/* kritická sekcia */
pthread_mutex_unlock(&mutex);
```

Semafor je vhodnejší vtedy, keď chceme riadiť poradie alebo počet povolených vstupov:

```
sem_wait(&sem);
/* pokračuj až po povolení */
sem_post(&sem);
```

Jednoduché zapamätanie:

- **mutex** = do miestnosti môže vstúpiť iba jedno vlákno,
- **semafor** = počítadlo povolení, môže riadiť aj poradie udalostí,
- **mutex chráni zdroj**,
- **semafor často signalizuje, že niečo nastalo**.

---

18. Bariéra

Bariéra je miesto v programe, kde musia počkať všetky vlákna. Žiadne vlákno nemôže pokračovať za bariéru, kým sa k nej nedostanú všetky ostatné vlákna.

Použitie bariéry:

- rozdelenie výpočtu na fázy,
- zabezpečenie, že všetky vlákna dokončili jednu časť pred začatím ďalšej,
- meranie času najpomalšieho vlákna,
- ladenie paralelných programov.

Predstav si to ako štart ďalšej etapy. Ak niektoré vlákno skončí svoju časť skôr, musí počkať na ostatné. Až keď všetci dorazia, pokračuje sa ďalej.

---

19. Implementácia bariéry

Prednáška ukazuje viac možností implementácie bariéry.

Bariéra pomocou busy-waitingu a mutexu

Použije sa zdieľané počítadlo chránené mutexom. Každé vlákno po príchode k bariére zvýši počítadlo. Keď počítadlo ukáže, že prišli všetky vlákna, môžu pokračovať.

Problém je, že ak sa používa busy-waiting, vlákna môžu zbytočne spotrebúvať CPU čas.

Bariéra pomocou semaforov

Semafory umožňujú uspať vlákna a zobudiť ich, keď sú splnené podmienky. To je efektívnejšie než prázdne čakanie.

Bariéra pomocou condition variables

Condition variables sú prirodzený nástroj na situácie, keď vlákno čaká na splnenie určitej podmienky. Preto sa dajú použiť aj na implementáciu bariéry.

---

20. Condition variables

Condition variable je synchronizačný objekt, ktorý umožňuje vláknu zaspať, kým nenastane určitá udalosť alebo podmienka. Keď podmienka nastane, iné vlákno môže čakajúce vlákno zobudiť.

Dôležité vlastnosti:

- condition variable je vždy spojená s mutexom,
- vlákno môže čakať na podmienku,
- iné vlákno môže signalizovať, že podmienka sa zmenila,
- čakajúce vlákno sa potom môže zobudiť a pokračovať.

Typické funkcie:

```
pthread_cond_init(&cond, NULL);
pthread_cond_wait(&cond, &mutex);
pthread_cond_signal(&cond);
pthread_cond_broadcast(&cond);
pthread_cond_destroy(&cond);
```

pthread_cond_wait

Táto funkcia spôsobí, že vlákno čaká na condition variable. Dôležité je, že pri čakaní uvoľní mutex, aby iné vlákna mohli zmeniť stav programu. Keď je vlákno zobudené, mutex získa späť.

pthread_cond_signal

Zobudí jedno čakajúce vlákno.

pthread_cond_broadcast

Zobudí všetky čakajúce vlákna.

Condition variables sú veľmi dôležité pri situáciách typu „čakaj, kým niečo nebude pripravené“. Sú efektívnejšie a čistejšie ako ručné busy-waiting cykly.

---

21. Read-write locks

Ďalšia časť prednášky rieši situáciu, keď máme veľkú zdieľanú dátovú štruktúru, napríklad utriedený zreťazený zoznam celých čísel. Nad zoznamom môžeme robiť operácie:

- Member – zisťuje, či sa hodnota nachádza v zozname,
- Insert – vkladá novú hodnotu,
- Delete – odstraňuje hodnotu.

Operácia Member iba číta. Operácie Insert a Delete menia štruktúru zoznamu.

Ak použijeme obyčajný mutex na celý zoznam, všetky operácie sa serializujú. To znamená, že aj viacero operácií Member, ktoré by sa mohli bezpečne vykonávať súčasne, musí čakať jedna na druhú.

To je zbytočné, najmä ak väčšina operácií len číta.

---

22. Hrubozrnný a jemnozrnný zámok pri zozname

Prednáška ukazuje dve možnosti.

Riešenie 1: jeden mutex pre celý zoznam

Výhoda:

- jednoduchá implementácia,
- ľahko sa chápe,
- korektné správanie.

Nevýhoda:

- celý zoznam môže používať naraz iba jedno vlákno,
- aj čítania sa navzájom blokujú,
- slabé využitie paralelizmu pri veľkom počte operácií Member.

Toto riešenie môže byť vhodné, ak väčšina operácií mení zoznam, teda ak prevládajú Insert a Delete.

Riešenie 2: mutex pre každý uzol

Toto je jemnozrnnejšie riešenie. Namiesto zamknutia celého zoznamu sa zamykajú jednotlivé uzly.

Výhoda:

- teoreticky väčší paralelizmus,
- vlákna môžu pracovať v rôznych častiach zoznamu.

Nevýhoda:

- implementácia je oveľa zložitejšia,
- každé prechádzanie zoznamu vyžaduje veľa operácií lock/unlock,
- každý uzol musí obsahovať mutex, čo zvyšuje pamäťovú náročnosť,
- program môže byť pomalší kvôli režijným nákladom synchronizácie.

---

23. Prečo sú read-write locks užitočné

Read-write lock je zámok, ktorý rozlišuje medzi čítaním a zápisom.

Má dva režimy:

- zámok na čítanie,
- zámok na zápis.

Viacero vlákien môže naraz získať zámok na čítanie. To je bezpečné, pretože čítanie nemení dátovú štruktúru.

Iba jedno vlákno môže získať zámok na zápis. Počas zápisu nesmie žiadne iné vlákno ani čítať, ani zapisovať, pretože štruktúra sa môže meniť.

Použitie:

```
pthread_rwlock_rdlock(&rwlock);
/* čítanie */
pthread_rwlock_unlock(&rwlock);

pthread_rwlock_wrlock(&rwlock);
/* zápis alebo zmena štruktúry */
pthread_rwlock_unlock(&rwlock);
```

Read-write lock je výhodný najmä tam, kde je veľa čítaní a málo zápisov. Napríklad ak 99 % operácií je Member a iba malé percento je Insert alebo Delete, read-write lock môže výrazne zlepšiť paralelizmus.

---

24. Pravidlá read-write locku

Treba si zapamätať tieto pravidlá:

- Ak viac vlákien iba číta, môžu čítať súčasne.

- Ak jedno vlákno zapisuje, nikto iný nesmie čítať ani zapisovať.
- Ak niekto číta, zapisujúce vlákno musí čakať.
- Ak niekto zapisuje, čítajúce aj zapisujúce vlákna musia čakať.

Jednoduchá predstava:

- čitáreň môže mať naraz veľa čitateľov,
- ale ak príde niekto prestavať knižnicu, všetci musia odísť,
- počas prestavby nesmie nikto čítať ani robiť ďalšie zmeny.

---

25. Cache a jej vplyv na zdieľanú pamäť

Prednáška ďalej upozorňuje, že výkon paralelných programov nezávisí iba od počtu vlákien a zámkov. Veľmi dôležitá je aj cache pamäť.

Moderné procesory majú viac úrovní cache:

- L1 cache,
- L2 cache,
- L3 cache,
- niekedy aj L4 cache.

Cache slúži na to, aby procesor nemusel stále pristupovať do pomalej hlavnej pamäte. Ak sa hľadané dáta nachádzajú v cache, ide o **cache hit**. Ak tam nie sú, ide o **cache miss** a dáta treba načítať z nižšej úrovne cache alebo z hlavnej pamäte.

Cache využíva dva princípy:

Temporal locality

Ak program práve použil nejakú hodnotu, je pravdepodobné, že ju čoskoro použije znova.

Spatial locality

Ak program použil nejakú adresu v pamäti, je pravdepodobné, že čoskoro použije aj okolité adresy.

Preto sa z pamäte do cache neprenáša iba jedna premenná, ale celý blok pamäte nazývaný **cache line** alebo **cache block**.

---

26. Cache coherence

V paralelnom systéme môže mať viac jadier vo svojej cache kópiu tej istej pamäťovej hodnoty. Ak jedno jadro túto hodnotu zmení, ostatné kópie musia byť aktualizované alebo označené ako neplatné.

Tento problém sa nazýva **cache coherence**.

Príklad:

- jadro 0 má v cache premennú x,
- jadro 1 má tiež v cache premennú x,
- jadro 0 zmení x,
- jadro 1 nesmie ďalej používať starú hodnotu.

Systém koherencie cache sa stará o to, aby jadrá nepracovali s neaktuálnymi hodnotami. Tento mechanizmus je potrebný pre správnosť, ale môže výrazne ovplyvniť výkon paralelných programov.

---

27. False sharing

False sharing je situácia, keď dve vlákna pracujú s rôznymi premennými, ale tieto premenné sa nachádzajú v tej istej cache line. Hoci logicky nezdieľajú tú istú premennú, hardvér ich vidí ako súčasť toho istého bloku cache.

Príklad:

```
double x, y;
```

Ak x a y ležia v tej istej cache line a:

- jadro 0 často mení x,
- jadro 1 často mení y,

tak cache line sa môže stále presúvať medzi jadrami. Výsledkom je veľká réžia koherencie cache, aj keď vlákna v skutočnosti nemenia tú istú premennú.

False sharing je zákerný, pretože program je logicky správny, ale výkon môže byť zlý.

Možné riešenia:

- oddeliť často menené premenné do rôznych cache lines,
- používať padding,
- usporiadať dáta tak, aby každé vlákno pracovalo s vlastným blokom pamäte,
- minimalizovať časté zápisy do susedných zdieľaných dát.

---

28. Thread-safety

Thread-safe kód je taký kód, ktorý môžu bezpečne vykonávať viaceré vlákna naraz bez toho, aby vznikli chyby.

Nie každá funkcia je thread-safe. Prednáška ukazuje príklad funkcie strtok, ktorá sa používa na rozdelenie textu na tokeny.

Pri prvom volaní sa strtok zavolá s reťazcom. Pri ďalších volaniach sa ako prvý argument dáva NULL, aby funkcia pokračovala v tokenizácii toho istého reťazca.

Problém je, že strtok si medzi volaniami ukladá stav do statickej premennej. Táto statická premenná je zdieľaná medzi vláknami. Ak teda dve vlákna volajú strtok súčasne, môžu si navzájom prepísať vnútorný stav a výsledok bude nesprávny.

Preto strtok nie je thread-safe.

---

29. Statické premenné a problém thread-safety

Funkcia, ktorá používa statickú premennú na uchovanie stavu medzi volaniami, môže byť nebezpečná vo viacvláknovom programe.

Príčina:

- statická premenná existuje iba raz,
- zdieľajú ju všetky vlákna,
- ak ju viaceré vlákna menia, vzniká konflikt.

Prednáška spomína aj ďalšie knižničné funkcie, ktoré môžu byť problematické:

- random zo stdlib.h,
- localtime z time.h.

V niektorých prípadoch existujú alternatívne reentrantné alebo thread-safe verzie funkcií. Tie sú navrhnuté tak, aby si stav neukladali do spoločnej statickej premennej, ale aby pracovali s dátami odovzdanými cez argumenty.

---

30. Reentrantné funkcie

Reentrantná funkcia je taká funkcia, ktorú možno bezpečne volať súčasne z viacerých vlákien, pretože nepoužíva zdieľaný vnútorný stav alebo si ho necháva explicitne odovzdať cez argumenty.

Rozdiel:

strtok(...)

nie je thread-safe, pretože si pamätá stav interne.

Bezpečnejší variant býva napríklad:

strtok_r(...)

Ten používa dodatočný argument na uchovanie stavu tokenizácie. Každé vlákno tak môže mať vlastný stav.

Zásada:

- vyhýbať sa funkciám, ktoré interne používajú zdieľaný statický stav,
- používať reentrantné varianty,
- pri práci s knižničnými funkciami overiť, či sú thread-safe.

---

31. Najdôležitejšie synchronizačné nástroje v Pthreads

pthread_create

Vytvorí nové vlákno.

pthread_create(&thread, NULL, function, argument);

pthread_join

Počká na dokončenie vlákna.

pthread_join(thread, NULL);

Mutex

Chráni kritickú sekciu.

```
pthread_mutex_lock(&mutex);
/* kritická sekcia */
pthread_mutex_unlock(&mutex);
```

Semafor

Riadi povolenia alebo poradie udalostí.

```
sem_wait(&sem);
/* čakaj na povolenie */
sem_post(&sem);
```

Condition variable

Uspí vlákno, kým nenastane podmienka.

```
pthread_cond_wait(&cond, &mutex);
pthread_cond_signal(&cond);
```

Read-write lock

Umožní viac súčasných čitateľov, ale iba jedného zapisovateľa.

```
pthread_rwlock_rdlock(&rwlock);
/* čítanie */
pthread_rwlock_unlock(&rwlock);

pthread_rwlock_wrlock(&rwlock);
/* zápis */
pthread_rwlock_unlock(&rwlock);
```

---

32. Čo si treba dávať pozor pri Pthreads

Pri Pthreads sa najčastejšie chyby objavujú v týchto oblastiach:

Zdieľané premenné

Ak viac vlákien používa spoločnú premennú, treba vedieť, či ju iba čítajú alebo aj zapisujú.

Kritické sekcie

Každá časť kódu, ktorá mení zdieľané dáta, musí byť chránená.

Príliš veľké kritické sekcie

Ak zamkneme príliš veľkú časť programu, stratíme paralelizmus.

Busy-waiting

Môže síce zabezpečiť poradie, ale plytvá CPU časom.

Nesprávne použitie mutexov

Ak vlákno zabudne uvoľniť mutex, ostatné vlákna môžu uviaznuť.

Deadlock

Deadlock vznikne, keď vlákna navzájom čakajú na zámky, ktoré si držia. Napríklad vlákno A čaká na zámok, ktorý drží vlákno B, a vlákno B čaká na zámok, ktorý drží vlákno A.

Thread-unsafe funkcie

Niektoré knižničné funkcie nie sú bezpečné pri súčasnom volaní z viacerých vlákien.

False sharing

Program môže byť korektný, ale pomalý, ak vlákna menia rôzne premenné v tej istej cache line.

---

33. Ako rozmýšľať pri návrhu Pthreads programu

Pri návrhu viacvláknového programu sa oplatí postupovať takto:

1. Určiť, čo sa dá rozdeliť medzi vlákna

Najprv treba nájsť nezávislé časti výpočtu. Napríklad riadky matice, intervaly cyklu alebo samostatné bloky dát.

2. Určiť, ktoré dáta sú súkromné a ktoré zdieľané

Súkromné dáta by malo mať každé vlákno vlastné. Zdieľané dáta treba chrániť.

3. Minimalizovať zdieľané zápisy

Čím menej vlákna zapisujú do spoločných premenných, tým menej synchronizácie treba.

4. Kritické sekcie robiť čo najkratšie

Vlákna by mali čo najviac počítať nezávisle a synchronizovať sa iba na krátky čas.

5. Vybrať správny synchronizačný mechanizmus

- mutex na jednoduchú kritickú sekciu,
- semafor na riadenie poradia alebo povolení,
- condition variable na čakanie na udalosť,
- bariéru na synchronizáciu všetkých vlákien v jednom bode,
- read-write lock na dátové štruktúry s veľa čítaniami a málo zápismi.

6. Myslieť aj na výkon cache

Aj správny program môže byť pomalý, ak zle pracuje s pamäťou a cache.

---

34. Najdôležitejšie pojmy na zapamätanie

Pthreads je POSIX API pre viacvláknové programovanie v zdieľanej pamäti.

Vlákno je ľahšia jednotka vykonávania než proces. Vlákna v jednom procese zdieľajú globálne dáta.

pthread_create vytvára nové vlákno.

pthread_join čaká na dokončenie vlákna.

Globálne premenné sú zdieľané medzi vláknami a môžu spôsobovať chyby.

Race condition vzniká, keď výsledok závisí od náhodného poradia vykonávania vlákien.

Kritická sekcia je časť programu, kde sa mení zdieľaný zdroj a musí tam byť naraz iba jedno vlákno.

Busy-waiting je aktívne čakanie v cykle. Je jednoduché, ale často veľmi neefektívne.

Mutex zabezpečuje vzájomné vylúčenie pri prístupe ku kritickej sekcii.

Semafor je počítadlový synchronizačný mechanizmus, ktorý sa hodí aj na signalizáciu medzi vláknami.

Bariéra je bod, kde čakajú všetky vlákna, kým sa k nemu nedostanú aj ostatné.

Condition variable umožňuje vláknu spať, kým nenastane určitá podmienka.

Read-write lock umožňuje súčasné čítanie viacerými vláknami, ale zápis povoľuje iba jednému vláknu.

Cache coherence zabezpečuje, aby viaceré cache kópie tej istej hodnoty boli konzistentné.

False sharing vzniká, keď rôzne premenné používané rôznymi vláknami ležia v tej istej cache line.

Thread-safe funkcia je funkcia, ktorú môžu bezpečne volať viaceré vlákna naraz.

---

35. Hlavná myšlienka celej prednášky

Prednáška 9 ukazuje, že programovanie v zdieľanej pamäti pomocou Pthreads je silný, ale citlivý nástroj. Vlákna sú ľahšie než procesy a vedia efektívne rozdeliť výpočet medzi viac

jadier. Keďže však zdieľajú globálne dáta, programátor musí veľmi presne riadiť prístup k spoločným premenným a dátovým štruktúram.

Najdôležitejšie je pochopiť, že paralelný program nie je správny len preto, že sa spustí vo viacerých vláknach. Musí byť správne rozdelená práca, správne určené zdieľané dáta a správne použitá synchronizácia.

Základná logika celej prednášky je:

- vlákna umožňujú paralelný výpočet,
- zdieľaná pamäť umožňuje jednoduchú komunikáciu,
- zdieľané dáta vytvárajú riziko race conditions,
- kritické sekcie treba chrániť,
- mutexy, semafory, bariéry, condition variables a read-write locks sú nástroje na synchronizáciu,
- výkon ovplyvňuje nielen počet vlákien, ale aj veľkosť kritických sekcií, spôsob čakania, cache coherence a false sharing,
- nie všetky funkcie knižnice C sú bezpečné pre viacvláknové použitie.

Ak si z tejto prednášky zapamätáš jednu vec, tak túto: **Pthreads dávajú programátorovi veľkú kontrolu nad vláknami, ale zároveň mu dávajú aj veľkú zodpovednosť za synchronizáciu a správnosť prístupu k zdieľaným dátam.**

Prednáška 10: OpenMP

1. Čo je OpenMP

OpenMP je API určené na paralelné programovanie v systémoch so **zdieľanou pamäťou**. To znamená, že všetky vlákna programu môžu potenciálne pristupovať k tej istej hlavnej pamäti. OpenMP je navrhnuté hlavne tak, aby sa existujúce sekvenčné programy dali pomerne jednoducho upraviť na paralelné – často stačí doplniť niekoľko direktív nad cykly alebo bloky kódu.

Na rozdiel od Pthreads, kde programátor ručne vytvára vlákna pomocou `pthread_create`, v OpenMP zvyčajne nepíšeme explicitný kód na vytváranie a rušenie vlákien. Namiesto toho používame **direktívy prekladača**, ktoré hovoria: „tento blok vykonaj paralelne“ alebo „tento cyklus rozdeľ medzi vlákna“.

OpenMP teda stojí medzi úplne ručným prístupom a automatickou paralelizáciou. Programátor stále rozhoduje, čo sa má paralelizovať, ale veľa technických detailov, napríklad vytvorenie tímu vlákien, rozdelenie iterácií cyklu alebo synchronizáciu na konci paralelného bloku, rieši runtime systém OpenMP.

---

2. Zdieľaná pamäť v OpenMP

OpenMP predpokladá model, v ktorom máme viac jadier alebo procesorov a všetky majú prístup k hlavnej pamäti. Program beží ako jeden proces, ale v určitých miestach sa jeho vykonávanie rozdelí na viac vlákien.

To znamená:

- vlákna môžu zdieľať premenné,
- každé vlákno môže mať aj vlastné súkromné premenné,
- komunikácia medzi vláknami často prebieha cez spoločné premenné,
- treba dávať pozor na race conditions, podobne ako pri Pthreads.

Hlavná výhoda OpenMP je jednoduchosť. Kým v Pthreads musí programátor vytvárať vlákna a spravovať ich ručne, v OpenMP väčšinou stačí pridať direktívu typu:

```
#pragma omp parallel
```

alebo

```
#pragma omp parallel for
```

a runtime systém sa postará o vytvorenie a riadenie vlákien.

---

3. Pragmy v OpenMP

OpenMP používa tzv. **pragmy**. Pragma je špeciálna inštrukcia pre prekladač. V jazyku C/C++ má tvar:

```
#pragma ...
```

V OpenMP sa používa napríklad:

```
#pragma omp parallel
```

Pragma hovorí prekladaču, že nasledujúci blok kódu má byť spracovaný špeciálnym spôsobom. Ak prekladač OpenMP nepodporuje, neznáme pragmy ignoruje. To je užitočné, pretože program môže zostať syntakticky platný aj bez OpenMP podpory, aj keď sa potom vykoná sekvenčne.

OpenMP programy sa preto často skladajú z obyčajného C kódu, ktorý je doplnený o OpenMP direktívy. To je dôvod, prečo sa OpenMP považuje za relatívne jednoduchý spôsob paralelizácie existujúceho programu.

---

4. Preklad OpenMP programu

Aby prekladač OpenMP direktívy naozaj použil, treba pri preklade zapnúť OpenMP podporu. Pri GCC sa používa prepínač:

```
gcc -g -Wall -fopenmp -o omp_hello omp_hello.c
```

Prepínač `-fopenmp` zabezpečí, že prekladač rozpozná OpenMP direktívy, pripojí potrebnú runtime podporu a umožní používať funkcie z hlavičkového súboru:

```
#include <omp.h>
```

Bez `-fopenmp` by sa OpenMP direktívy ignorovali alebo by program nemusel vedieť preložiť OpenMP funkcie ako `omp_get_thread_num()`.

---

5. Základná paralelná direktíva

Najjednoduchšia OpenMP direktíva je:

```
#pragma omp parallel
{
   /* kód vykonávaný viacerými vláknami */
}
```

Táto direktíva spôsobí, že runtime systém vytvorí skupinu vlákien a každé vlákno vykoná ten istý blok kódu. V OpenMP sa táto skupina vlákien nazýva **team**. Pôvodné vlákno programu sa označuje ako hlavné/master vlákno a ostatné vlákna sú dodatočne vytvorené pracovné vlákna.

Ak napríklad každé vlákno vypíše svoj identifikátor, poradie výpisu nemusí byť rovnaké pri každom spustení. To je normálne, pretože vlákna bežia súbežne a ich presné naplánovanie závisí od operačného systému a runtime systému.

---

6. Počet vlákien: `num_threads`

Počet vlákien možno určiť pomocou klauzuly `num_threads`:

```
#pragma omp parallel num_threads(thread_count)
{
   /* paralelný blok */
}
```

Klauzula je doplnok k direktíve, ktorý mení jej správanie. V tomto prípade hovorí, koľko vlákien by malo vykonávať nasledujúci blok. OpenMP štandard však nezaručuje absolútne, že runtime systém vždy vytvorí presne požadovaný počet vlákien, pretože systém môže mať vlastné obmedzenia. V bežných prípadoch však dostaneme požadovaný počet, ak nejde o extrémne veľa vlákien.

Vo vnútri paralelného bloku sa často používajú funkcie:

```
int my_rank = omp_get_thread_num();
int thread_count = omp_get_num_threads();
```

omp_get_thread_num() vráti číslo aktuálneho vlákna v tíme. omp_get_num_threads() vráti počet vlákien v aktuálnom paralelnom bloku.

---

7. Ochrana kódu pre prekladače bez OpenMP

Ak chceme, aby program vedel fungovať aj s prekladačom bez OpenMP, môžeme použiť makro _OPENMP.

Príklad:

```
#ifdef _OPENMP
#include <omp.h>
#endif
```

A vo vnútri programu:

```
#ifdef _OPENMP
int my_rank = omp_get_thread_num();
int thread_count = omp_get_num_threads();
#else
int my_rank = 0;
int thread_count = 1;
#endif
```

Zmysel je jednoduchý: ak sa program prekladá s OpenMP podporou, použijú sa OpenMP funkcie. Ak nie, program sa správa ako jednovláknový program.

---

8. OpenMP vs. Pthreads

OpenMP a Pthreads riešia podobný typ problému – paralelné programovanie v zdieľanej pamäti – ale úplne iným štýlom.

Pthreads

Pri Pthreads programátor explicitne:

- vytvára vlákna,
- odovzdáva im argumenty,
- čaká na ich dokončenie,
- používa mutexy, semafory a ďalšie synchronizačné nástroje ručne.

OpenMP

Pri OpenMP programátor väčšinou:

- označí paralelné oblasti pomocou direktív,
- nechá runtime systém vytvoriť vlákna,
- používa klauzuly na riadenie rozsahu premenných, počtu vlákien a rozdelenia práce,
- používa jednoduchšie direktívy ako `parallel for`, `critical`, `atomic`, `barrier`.

OpenMP je preto často vhodné na rýchlu paralelizáciu cyklov a numerických výpočtov, zatiaľ čo Pthreads dáva nižšiu úroveň kontroly, ale za cenu väčšej zložitosti.

---

9. Príklad: lichobežníkové pravidlo

Prednáška používa ako príklad **lichobežníkové pravidlo** na numerickú aproximáciu integrálu. Ide o výpočet, kde interval rozdelíme na veľa malých častí a nad každou časťou vypočítame plochu lichobežníka. Celkový výsledok je súčet týchto čiastkových plôch.

Tento príklad je vhodný na paralelizáciu, pretože:

- výpočet jednotlivých lichobežníkov je navzájom nezávislý,
- počet lichobežníkov býva oveľa väčší než počet jadier,
- prácu možno rozdeliť medzi vlákna po blokoch,
- problém vzniká až pri sčítaní lokálnych výsledkov do spoločného výsledku.

Rozdelíme teda úlohu na dve časti:

1. Každé vlákno vypočíta plochy svojho bloku lichobežníkov.
2. Lokálny výsledok každého vlákna sa pripočíta do spoločnej premennej `global_result`.

Prvá časť je prirodzene paralelná. Druhá časť je problémová, pretože viac vlákien chce meniť tú istú zdieľanú premennú.

---

10. Problém so spoločným výsledkom

Ak viac vlákien vykoná naraz:

```
global_result += my_result;
```

môže vzniknúť race condition. Táto operácia totiž nie je v skutočnosti jeden nedeliteľný krok. Procesor musí najprv načítať `global_result`, pripočítať `my_result` a potom výsledok zapísať späť. Ak to urobia dve vlákna súčasne, jedno vlákno môže prepísať výsledok druhého.

Preto treba zabezpečiť vzájomné vylúčenie, teda aby túto aktualizáciu robilo naraz iba jedno vlákno.

V OpenMP sa to dá spraviť napríklad takto:

```
#pragma omp critical
global_result += my_result;
```

Direktíva critical zabezpečí, že daný blok alebo príkaz vykonáva naraz iba jedno vlákno. Je to jednoduché a správne riešenie, ale nie vždy najvýkonnejšie, pretože kritická sekcia serializuje časť programu.

---

11. Rozsah premenných: shared a private

V sekvenčnom programe rozsah premennej znamená, v ktorej časti programu možno premennú používať. V OpenMP má pojem rozsahu ešte ďalší význam: hovorí, **ktoré vlákna majú k premennej prístup**.

OpenMP rozlišuje najmä:

Shared premenné

Premenná so zdieľaným rozsahom je spoločná pre všetky vlákna v tíme. Ak ju jedno vlákno zmení, ostatné vlákna môžu túto zmenu vidieť.

Private premenné

Premenná so súkromným rozsahom existuje samostatne pre každé vlákno. Každé vlákno má vlastnú kópiu tejto premennej.

Predvolené pravidlo je dôležité: premenné deklarované pred paralelným blokom sú štandardne **shared**. To môže byť nebezpečné, ak si programátor neuvedomí, že viac vlákien pracuje s tou istou premennou.

---

12. Prečo je private rozsah dôležitý

Predstavme si premennú, do ktorej si každé vlákno ukladá svoj lokálny výpočet. Ak by táto premenná bola shared, vlákna by si navzájom prepisovali hodnoty.

Preto lokálne medzivýsledky majú byť private:

```
#pragma omp parallel private(my_result)
{
```

```
    my_result = ...;
}
```

Každé vlákno potom dostane vlastnú premennú my_result. To znamená, že vlákno 0, vlákno 1 a vlákno 2 nepíšu do tej istej pamäťovej bunky, ale do svojich samostatných kópií.

Pri OpenMP je veľmi dôležité vedieť si pri každej premennej položiť otázku:

- Má byť táto premenná spoločná pre všetky vlákna?
- Alebo má mať každé vlákno vlastnú kópiu?

Toto rozhodnutie často rozhoduje o správnosti celého paralelného programu.

---

13. Klauzula default

OpenMP umožňuje použiť klauzulu default, ktorá núti programátora explicitne určiť rozsah premenných.

Najpraktickejšia verzia je:

```
#pragma omp parallel default(none) shared(...) private(...)
```

default(none) znamená, že prekladač nebude automaticky predpokladať, či sú premenné shared alebo private. Programátor musí rozsah každej vonkajšej premennej uviesť ručne.

To je užitočné najmä pri väčších programoch, pretože znižuje riziko, že omylom zdieľame premennú, ktorá mala byť súkromná.

---

14. Reduction: lepšie riešenie pre súčty

Pri lichobežníkovom pravidle alebo výpočte π často potrebujeme, aby každé vlákno vypočítalo čiastkový výsledok a všetky čiastkové výsledky sa potom spojili do jedného výsledku. Toto sa nazýva **redukcia**.

Redukcia je výpočet, ktorý opakovane používa tú istú binárnu operáciu na postupnosť operandov, aby vznikol jeden výsledok. Typické redukčné operácie sú:

- súčet,
- súčin,
- minimum,
- maximum,
- logické operácie,
- bitové operácie.

OpenMP podporuje redukciu pomocou klauzuly:

```
#pragma omp parallel reduction(+:global_result)
{
   global_result += local_value;
}
```

alebo pri cykle:

```
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < n; i++) {
   sum += a[i];
}
```

OpenMP v takom prípade vytvorí súkromnú kópiu redukčnej premennej pre každé vlákno. Každé vlákno počíta do svojej kópie a na konci runtime systém bezpečne spojí výsledky do jednej hodnoty.

To je lepšie než critical, pretože sa nemusí zamykať spoločná premenná pri každom pripočítaní. Redukcia je preto typický a veľmi dôležitý mechanizmus OpenMP.

---

15. critical vs. reduction

Tieto dva mechanizmy sa často používajú pri podobných problémoch, ale nie sú rovnako vhodné.

critical

Použije sa, keď potrebujeme všeobecne ochrániť kritickú sekciu:

```
#pragma omp critical
{
   global_result += my_result;
}
```

Výhoda:

- jednoduché,
- univerzálne,
- použiteľné aj pri zložitejších operáciách.

Nevýhoda:

- do kritickej sekcie môže vstúpiť iba jedno vlákno,
- môže výrazne spomaliť program, ak sa používa často.

reduction

Použije sa, keď robíme opakované skladanie výsledkov jednou operáciou:

```
#pragma omp parallel for reduction(+:sum)
```

Výhoda:

- výkonnejšie,
- jednoduchšie pre typické súčty a súčiny,
- OpenMP sa postará o súkromné kópie a bezpečné spojenie.

Nevýhoda:

- dá sa použiť len pri operáciách, ktoré majú redukčný charakter.

Na učenie si stačí zapamätať: **ak ide o súčet, súčin alebo podobnú akumuláciu, preferuj reduction; ak ide o všeobecnú kritickú sekciu, použi critical.**

---

16. Direktíva parallel for

Jedna z najdôležitejších direktív v OpenMP je:

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

Táto direktíva vytvorí tím vlákien a rozdelí iterácie cyklu medzi vlákna. Každé vlákno vykoná iba časť iterácií.

Napríklad ak máme 1000 iterácií a 4 vlákna, OpenMP môže rozdeliť prácu tak, že:

- vlákno 0 spracuje iterácie 0–249,
- vlákno 1 spracuje iterácie 250–499,
- vlákno 2 spracuje iterácie 500–749,
- vlákno 3 spracuje iterácie 750–999.

Toto je veľmi typické použitie OpenMP: vezmeme sekvenčný for cyklus a pomocou jednej direktívy ho rozdelíme medzi viac vlákien.

---

17. Kedy sa dá for cyklus bezpečne paralelizovať

Nie každý for cyklus je vhodný na paralelizáciu. OpenMP prekladač vo všeobecnosti nekontroluje všetky závislosti medzi iteráciami. Programátor musí vedieť, či je paralelizácia bezpečná.

Cyklus sa dá paralelizovať dobre vtedy, keď sú jeho iterácie nezávislé.

Príklad vhodný na paralelizáciu:

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

Každá iterácia pracuje s iným prvkom poľa. Výpočet c[10] nezávisí od c[9] ani c[11].

Príklad nevhodný na jednoduchú paralelizáciu:

```
fibo[0] = fibo[1] = 1;
for (int i = 2; i < n; i++) {
   fibo[i] = fibo[i-1] + fibo[i-2];
}
```

Tu každá iterácia závisí od predchádzajúcich výsledkov. Ak by sme iterácie vykonali paralelne, niektoré vlákna by mohli čítať hodnoty, ktoré ešte neboli správne vypočítané. Výsledok by bol nesprávny.

---

18. Právne formy paralelizovateľného for cyklu

OpenMP má požiadavky na tvar cyklu, ktorý sa dá použiť s parallel for.

Riadiaca premenná cyklu musí byť celočíselného alebo ukazovateľového typu. Nemala by byť napríklad typu float.

Výrazy určujúce začiatok, koniec a krok cyklu sa nesmú meniť počas vykonávania cyklu. Riadiaca premenná sa počas vykonávania cyklu môže meniť iba inkrementačnou časťou vo forme for.

Zmysel týchto obmedzení je jednoduchý: OpenMP musí vedieť pred začiatkom alebo počas vykonávania rozumne určiť, koľko iterácií existuje a ako ich rozdeliť medzi vlákna. Ak by sa hranice cyklu menili nepredvídateľne, rozdelenie práce by nebolo bezpečné.

---

19. Výpočet π v OpenMP

Prednáška používa aj príklad odhadu hodnoty π. Tento výpočet je podobný lichobežníkovému pravidlu v tom, že ide o veľký súčet mnohých členov.

Na začiatku sa môže zdať, že stačí dať nad cyklus:

```
#pragma omp parallel for
```

Problém však vznikne, ak všetky vlákna zapisujú do tej istej sumy. Vtedy vzniká race condition alebo závislosť v cykle.

Správne riešenia sú napríklad:

- zabezpečiť, aby každé vlákno malo vlastnú lokálnu sumu,
- použiť reduction,
- alebo použiť vhodnú kombináciu private premenných a bezpečného spojenia výsledkov.

Najčistejšie riešenie je zvyčajne:

```
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < n; i++) {
   sum += ...
}
```

Takto OpenMP vie, že sum je redukčná premenná, a zabezpečí jej správne spracovanie.

---

20. Triedenie a odd-even transposition sort

Prednáška prechádza aj na triedenie, konkrétne na **odd-even transposition sort**. Ide o triediaci algoritmus podobný bubble sortu, ale vhodnejší na paralelné vysvetlenie.

Algoritmus opakovane vykonáva dve fázy:

Párna fáza

Porovnávajú sa dvojice:

(0,1), (2,3), (4,5), ...

Nepárna fáza

Porovnávajú sa dvojice:

(1,2), (3,4), (5,6), ...

V každej fáze sú porovnávané dvojice nezávislé. Napríklad v párnej fáze dvojica (0,1) nepracuje s rovnakými prvkami ako dvojica (2,3), takže sa tieto porovnania dajú robiť paralelne.

Dôležité však je, že medzi fázami existuje závislosť. Nepárna fáza musí čakať, kým skončí párna fáza, pretože výsledok párnej fázy ovplyvňuje vstup nepárnej fázy. Preto sa tento algoritmus hodí na vysvetlenie, že paralelizmus môže existovať **vnútri fázy**, ale fázy medzi sebou môžu byť usporiadané sekvenčne.

21. OpenMP verzia odd-even sortu

Pri OpenMP možno jednotlivé fázy triedenia zapisovať pomocou parallel for.

Schématicky:

```
for (phase = 0; phase < n; phase++) {
   if (phase % 2 == 0) {
      #pragma omp parallel for
      for (i = 1; i < n; i += 2) {
         /* porovnaj a prípadne vymeň a[i-1], a[i] */
      }
   } else {
      #pragma omp parallel for
      for (i = 1; i < n-1; i += 2) {
         /* porovnaj a prípadne vymeň a[i], a[i+1] */
      }
   }
}
```

Dôležitá myšlienka: každá fáza má vlastný paralelný cyklus, ale medzi fázami musí zostať poradie. Preto je prirodzené, že na konci parallel for je implicitná bariéra. Tá zabezpečí, že všetky porovnania v danej fáze skončia skôr, než začne ďalšia fáza.

---

22. Plánovanie cyklov v OpenMP

Keď OpenMP paralelizuje cyklus, musí rozhodnúť, ktoré iterácie dostane ktoré vlákno. Toto sa nazýva **scheduling**, teda plánovanie alebo rozvrhovanie iterácií.

Nie vždy sú všetky iterácie rovnako náročné. Ak každá iterácia trvá približne rovnako dlho, jednoduché blokové rozdelenie funguje dobre. Ak však neskoršie iterácie trvajú dlhšie než skoršie, blokové rozdelenie môže byť nevyvážené.

Prednáška uvádza príklad funkcie f(i), ktorá volá sin približne i-krát. To znamená, že výpočet pre väčšie i trvá dlhšie. Ak jedno vlákno dostane prvú polovicu iterácií a druhé druhú polovicu, druhé vlákno bude mať oveľa viac práce. Výsledkom je slabé zrýchlenie. Pri cyklickom rozdelení sa náročné a menej náročné iterácie rozdelia rovnomernejšie.

---

23. Klauzula schedule

Rozdelenie iterácií možno riadiť pomocou:

```
#pragma omp parallel for schedule(type, chunksize)
```

type určuje spôsob rozdelenia iterácií. chunksize určuje veľkosť bloku iterácií, ktorý sa pridelí vláknu.

OpenMP pozná najmä tieto typy:

- static,
- dynamic,
- guided,
- auto,
- runtime.

Každý typ je vhodný pre inú situáciu.

---

24. Static schedule

Pri static rozdelení sa iterácie pridelia vláknam pred začiatkom cyklu. To znamená, že runtime systém už počas cyklu nemusí rozhodovať, kto dostane ďalšiu prácu.

Príklad:

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < n; i++) {
   ...
}
```

Ak sú iterácie približne rovnako náročné, static je veľmi dobré riešenie, pretože má malú režijnosť.

Môžeme použiť aj chunksize:

```
#pragma omp parallel for schedule(static, 1)
```

Pri static, 1 sa iterácie typicky rozdeľujú cyklicky:

- vlákno 0: iterácie 0, 4, 8, ...
- vlákno 1: iterácie 1, 5, 9, ...
- vlákno 2: iterácie 2, 6, 10, ...
- vlákno 3: iterácie 3, 7, 11, ...

Cyklické rozdelenie môže pomôcť pri nerovnomerne náročných iteráciách, pretože každé vlákno dostane zmes ľahších a ťažších iterácií.

---

25. Dynamic schedule

Pri dynamic rozdelení sa iterácie rozdelia na bloky a vlákna si ich berú postupne počas vykonávania. Keď vlákno dokončí svoj blok, požiada runtime systém o ďalší.

Príklad:

```
#pragma omp parallel for schedule(dynamic, 10)
for (int i = 0; i < n; i++) {
   ...
}
```

To znamená, že iterácie sa prideľujú po blokoch veľkosti 10. Ak jedno vlákno skončí skôr, dostane ďalší blok.

Výhoda:

- dobré pri nerovnomerne náročných iteráciách,
- lepšie vyvažovanie záťaže.

Nevýhoda:

- vyššia režijnosť, lebo runtime systém musí počas behu prideľovať prácu.

Ak chunksize nie je uvedený, pri dynamic sa často použije veľkosť bloku 1. To môže najlepšie vyvažovať záťaž, ale aj vytvárať najväčšiu režijnosť.

---

26. Guided schedule

Pri guided rozvrhovaní si vlákna tiež berú bloky práce postupne, ale veľkosť blokov sa postupne zmenšuje. Na začiatku sú bloky väčšie, neskôr menšie.

Príklad:

```
#pragma omp parallel for schedule(guided)
for (int i = 0; i < n; i++) {
   ...
}
```

Zmysel je kombinovať dve výhody:

- na začiatku väčšie bloky znižujú režijnosť,
- ku koncu menšie bloky zlepšujú vyváženie práce.

Ak nie je uvedený chunksize, veľkosť blokov môže klesať až na 1. Ak chunksize uvedený je, bloky klesajú po túto hodnotu, hoci posledný blok môže byť menší.

---

27. Auto a runtime schedule

auto

Pri schedule(auto) necháme rozhodnutie na prekladač alebo runtime systém:

```
#pragma omp parallel for schedule(auto)
```

Programátor nešpecifikuje konkrétnu stratégiu. Systém si vyberie sám.

runtime

Pri schedule(runtime) sa stratégia určí až pri spustení programu pomocou environmentálnej premennej OMP_SCHEDULE:

```
#pragma omp parallel for schedule(runtime)
```

Napríklad:

```
export OMP_SCHEDULE="dynamic,4"
```

Výhoda je, že môžeme testovať rôzne plánovania bez zmeny a prekladu programu.

---

28. Ako si vybrať správne plánovanie

Pri výbere schedule sa dá riadiť jednoduchou logikou:

Keď sú iterácie rovnako náročné

Použi:

```
schedule(static)
```

Má najnižšiu režijnosť.

Keď sú iterácie rôzne náročné

Použi:

```
schedule(dynamic, chunksize)
```

alebo:

```
schedule(guided, chunksize)
```

Tieto režimy lepšie vyvažujú prácu medzi vlákna.

Keď chceš experimentovať bez prekladu

Použi:

schedule(runtime)

a nastav OMP_SCHEDULE.

Dôležitá myšlienka: **nesprávne rozdelenie iterácií môže spôsobiť, že niektoré vlákna už skončia a čakajú, zatiaľ čo iné ešte stále pracujú. Taký program síce používa viac vlákien, ale nevyužíva ich efektívne.**

---

29. Bariéry v OpenMP

Bariéra je synchronizačný bod, kde musia počkať všetky vlákna. Žiadne vlákno nemôže pokračovať ďalej, kým sa k bariére nedostanú všetky vlákna v tíme.

V OpenMP existujú:

Implicitné bariéry

Niektoré OpenMP direktívy majú bariéru automaticky na konci. Napríklad na konci:

#pragma omp parallel for

sa vlákna typicky zosynchronizujú pred pokračovaním ďalej.

Explicitná bariéra

Programátor môže bariéru vložiť ručne:

#pragma omp barrier

Keď vlákno narazí na túto direktívu, zablokuje sa, kým na rovnaké miesto nedorazia aj ostatné vlákna.

Bariéra je dôležitá napríklad pri viacfázových algoritmoch. Ak druhá fáza potrebuje výsledky prvej fázy od všetkých vlákien, musí byť medzi nimi bariéra.

---

30. **critical**, **atomic** a locks

OpenMP poskytuje viac mechanizmov na ochranu kritických sekcií.

critical

Používa sa na všeobecnú kritickú sekciu:

```
#pragma omp critical
{
   /* kritická sekcia */
}
```

Naraz ju vykonáva iba jedno vlákno.

atomic

Používa sa na jednoduché operácie typu načítaj–uprav–zapíš, napríklad:

```
#pragma omp atomic
x += y;
```

atomic je obmedzenejšie než critical, pretože chráni iba jednoduchý príkaz určitého tvaru. Výhodou je, že môže byť efektívnejšie, lebo mnohé procesory majú špeciálne inštrukcie pre atomické load-modify-store operácie.

Locks

OpenMP poskytuje aj explicitné zámky. Zámok je dátová štruktúra a sada funkcií, pomocou ktorých programátor ručne zamyká a odomyká kritickú sekciu.

Používajú sa napríklad:

```
omp_lock_t lock;

omp_init_lock(&lock);
omp_set_lock(&lock);

/* kritická sekcia */

omp_unset_lock(&lock);
omp_destroy_lock(&lock);
```

OpenMP rozlišuje jednoduché a vnorené zámky. Jednoduchý zámok môže byť nastavený iba raz, kým sa neuvoľní. Vnorený zámok môže to isté vlákno nastaviť viackrát a musí ho potom zodpovedajúco viackrát uvoľniť.

---

31. Named critical sections

OpenMP umožňuje pomenovať kritickú sekciu:

```
#pragma omp critical(name)
{
   ...
}
```

Ak majú dve kritické sekcie rovnaký názov, navzájom sa vylučujú. Ak majú rôzne názvy, môžu sa vykonávať súčasne, pretože chránia nezávislé zdroje.

Príklad:

```
#pragma omp critical(sum_update)
sum += local_sum;

#pragma omp critical(log_update)
log_count++;
```

Tieto dve kritické sekcie by sa nemuseli navzájom blokovať, ak majú rôzne názvy a chránia rôzne dáta.

Zmysel pomenovaných kritických sekcií je znížiť zbytočné čakanie. Ak dve kritické sekcie nesúvisia, nemusia používať rovnaký globálny zámok.

---

32. Na čo si dať pozor pri vzájomnom vylúčení

Prednáška zdôrazňuje niekoľko upozornení:

Nemiešať mechanizmy pre tú istú kritickú sekciu

Ak raz chránime konkrétnu premennú pomocou `critical`, nemali by sme inde tú istú premennú chrániť pomocou iného zámku alebo `atomic`, ak presne nerozumieme dôsledkom.

OpenMP negarantuje férovosť

Nie je zaručené, že vlákna dostanú prístup ku kritickej sekcii v poradí, v akom o ňu požiadali. Preto netreba stavať logiku programu na predpoklade „spravodlivého“ poradia.

Pozor na vnáranie zámkov

Ak vlákno drží jeden zámok a počas toho sa pokúsi získať ďalší, môže vzniknúť deadlock. Vnorené synchronizačné konštrukcie treba používať opatrne.

---

33. Race condition v OpenMP

Rovnako ako pri Pthreads, aj v OpenMP je veľký problém **race condition**. Vzniká vtedy, keď viaceré vlákna pristupujú k zdieľanému zdroju, aspoň jedno z nich ho mení a výsledok závisí od poradia vykonávania.

Typický problém:

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {
   sum += a[i];
}
```

Toto je nesprávne, pretože sum je shared a viac vlákien ju mení naraz.

Možné opravy:

```
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < n; i++) {
   sum += a[i];
}
```

alebo menej efektívne:

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {
   #pragma omp critical
   sum += a[i];
}
```

Správna voľba je v tomto prípade reduction, pretože ide o typickú redukčnú operáciu.

---

34. Task parallelism v kontexte prednášky

V roadmap sa spomína aj **task parallelism**, teda paralelizmus úloh. V tejto prednáške sa však hlavný dôraz dáva najmä na paralelizáciu cyklov a synchronizáciu. Myšlienka task parallelism je, že namiesto delenia iterácií jedného cyklu medzi vlákna rozdeľujeme rôzne typy práce.

Pri lichobežníkovom pravidle sa dá úloha chápať ako:

- veľa nezávislých úloh výpočtu plôch jednotlivých lichobežníkov,
- jedna úloha spojenia výsledkov.

Pri triedení zasa existujú fázy, v ktorých možno niektoré porovnania robiť paralelne, ale medzi fázami treba zachovať poradie.

Dôležité je pochopiť rozdiel:

- **data parallelism**: rovnaká operácia nad rôznymi dátami,
- **task parallelism**: rôzne úlohy alebo časti programu bežia paralelne.

OpenMP vie podporovať oba prístupy, ale v tejto prednáške je najviac rozobratý dátový paralelizmus cez parallel for.

---

35. Najdôležitejšie direktívy a klauzuly

#pragma omp parallel

Vytvorí paralelnú oblasť. Nasledujúci blok vykoná tím vlákien.

```
#pragma omp parallel
{
   ...
}
```

num_threads

Určí požadovaný počet vlákien.

```
#pragma omp parallel num_threads(4)
```

#pragma omp parallel for

Rozdelí iterácie cyklu medzi vlákna.

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

private

Každé vlákno má vlastnú kópiu premennej.

```
#pragma omp parallel private(x)
```

shared

Premenná je spoločná pre všetky vlákna.

```
#pragma omp parallel shared(a, b)
```

default(none)

Programátor musí explicitne určiť rozsah premenných.

```
#pragma omp parallel default(none) shared(a) private(i)
```

reduction

Bezpečne spojí lokálne výsledky vlákien.

```
#pragma omp parallel for reduction(+:sum)
```

schedule

Riadi rozdelenie iterácií cyklu.

#pragma omp parallel for schedule(dynamic, 4)

critical

Všeobecná kritická sekcia.

#pragma omp critical

atomic

Efektívnejšia ochrana jednoduchej aktualizácie.

#pragma omp atomic
x += y;

barrier

Synchronizačný bod pre všetky vlákna.

#pragma omp barrier

---

36. Ako rozmýšľať pri paralelizácii cyklu v OpenMP

Pri každom cykle si treba položiť tieto otázky:

1. Sú iterácie nezávislé?

Ak iterácia i potrebuje výsledok iterácie i-1, cyklus sa nedá jednoducho paralelizovať pomocou parallel for.

2. Ktoré premenné majú byť private?

Premenné používané ako lokálne medzivýsledky by mali byť súkromné pre každé vlákno.

3. Ktoré premenné majú byť shared?

Vstupné polia a globálne výsledky môžu byť zdieľané, ale pri zápise treba dávať pozor.

4. Je výsledok redukcia?

Ak sa veľa hodnôt spája do jedného výsledku cez +, *, &&, || a podobne, treba použiť reduction.

5. Sú iterácie rovnako náročné?

Ak áno, stačí static. Ak nie, môže byť lepší dynamic alebo guided.

6. Potrebujem synchronizáciu medzi fázami?

Ak ďalšia fáza závisí od dokončenia predchádzajúcej, treba bariéru. V mnohých OpenMP konštrukciách je bariéra implicitná.

---

37. Typické chyby v OpenMP

Paralelizácia cyklu so závislosťami

Napríklad Fibonacciho postupnosť nemožno jednoducho paralelizovať cez parallel for, pretože každá hodnota závisí od predchádzajúcich.

Zabudnutá redukcia

Ak viac vlákien aktualizuje tú istú sumu bez reduction, vzniká race condition.

Zle nastavený rozsah premenných

Ak premenná, ktorá mala byť private, zostane shared, vlákna si ju môžu prepisovať.

Príliš časté critical

Ak je critical vo vnútri veľkého cyklu a vykonáva sa pri každej iterácii, program môže byť skoro sériový.

Nevhodný schedule

Ak majú iterácie rôznu náročnosť a použije sa blokové statické rozdelenie, niektoré vlákna môžu skončiť skoro a zvyšok času čakať.

Zbytočné alebo vnorené zámky

Nesprávne použitie lockov môže spôsobiť deadlock alebo výrazné spomalenie.

---

38. Čo je najvhodnejšie na zapamätanie

OpenMP je najlepšie chápať ako nástroj, ktorý umožňuje relatívne jednoducho vytvoriť paralelný program v zdieľanej pamäti. Programátor nepíše priamo vytváranie vlákien ako v Pthreads, ale označuje oblasti programu, ktoré sa majú vykonať paralelne.

Najdôležitejšie pojmy:

- **OpenMP** – API pre paralelné programovanie v zdieľanej pamäti.
- **Pragma** – direktíva pre prekladač, napríklad `#pragma omp parallel`.
- **Parallel region** – blok kódu vykonávaný tímom vlákien.
- **Team** – skupina vlákien vykonávajúcich paralelný blok.
- **Clause** – doplnok direktívy, napríklad `num_threads`, `private`, `reduction`.
- **Shared variable** – premenná dostupná všetkým vláknam.
- **Private variable** – každé vlákno má vlastnú kópiu premennej.
- **Reduction** – bezpečné spojenie lokálnych výsledkov do jedného výsledku.
- **Parallel for** – rozdelenie iterácií cyklu medzi vlákna.
- **Schedule** – spôsob prideľovania iterácií vláknam.
- **Barrier** – bod, kde čakajú všetky vlákna.
- **Critical** – kritická sekcia, do ktorej vstúpi naraz iba jedno vlákno.
- **Atomic** – efektívna ochrana jednoduchej aktualizácie.
- **Lock** – explicitný zámok riadený programátorom.
- **Race condition** – chyba spôsobená nepredvídateľným poradím prístupu ku zdieľaným dátam.

---

39. Hlavná myšlienka celej prednášky

Prednáška 10 ukazuje, že OpenMP je praktický nástroj na paralelizáciu programov v zdieľanej pamäti. Je jednoduchší než Pthreads, pretože veľa práce s vláknami robí automaticky runtime systém. Programátor však stále musí rozumieť tomu, čo sa deje: ktoré premenné sú zdieľané, ktoré sú súkromné, či medzi iteráciami cyklu existujú závislosti a ako bezpečne spojiť výsledky viacerých vlákien.

Najdôležitejší princíp je tento: **OpenMP vie veľmi jednoducho rozdeliť prácu medzi vlákna, ale nezaručí automaticky, že program je logicky správny. Správnosť závisí od toho, či programátor správne rozpozná závislosti, rozsah premenných a potrebu synchronizácie.**

OpenMP je teda veľmi vhodné na cykly, kde sú iterácie nezávislé, napríklad spracovanie polí, numerické výpočty, súčty, integrácie a jednoduché transformácie dát. Menej vhodné je na cykly s pevnými závislosťami medzi iteráciami, napríklad Fibonacciho postupnosť alebo algoritmy, kde každý krok potrebuje výsledok predchádzajúceho kroku.

Ak si z tejto prednášky zapamätáš jednu vetu, tak túto: **OpenMP zrýchľuje hlavne tým, že delí nezávislé časti práce medzi vlákna, ale pri zdieľaných dátach musíš vždy riešiť rozsah premenných, race conditions a synchronizáciu.**

Prednáška 11-a: Graphics Processing Unit (GPU)

1. Čo je GPU

GPU (Graphics Processing Unit) je grafický procesor, ktorý bol pôvodne navrhnutý hlavne na spracovanie grafiky. Jeho úlohou bolo zrýchľovať vykresľovanie 2D a 3D obrazu, teda napríklad výpočet geometrie, osvetlenia, textúr, rasterizácie a výstupu obrazu na monitor.

Dnes však GPU už nie je len „grafická karta na obraz“. Moderné GPU je vysoko paralelný, viacvláknový a mnohojadrový procesor, ktorý dokáže vykonávať obrovské množstvo jednoduchších operácií naraz. Preto sa používa nielen v hrách a grafike, ale aj vo vedeckých výpočtoch, spracovaní obrazu, umelej inteligencii, strojovom učení a veľkých dátach.

Základná myšlienka GPU je: namiesto toho, aby malo niekoľko veľmi silných jadier ako CPU, má veľa menších výpočtových jednotiek, ktoré sú vhodné na masívne paralelné spracovanie dát.

---

2. GPGPU – GPU ako všeobecný výpočtový procesor

Pôvodne sa GPU programovalo cez grafické API, napríklad OpenGL alebo podobné grafické rozhrania. To znamenalo, že aj vedecký alebo numerický výpočet bolo nutné „preložiť“ do grafického spôsobu uvažovania. To bolo nepraktické a obmedzovalo to využitie GPU mimo grafiky.

Postupne vznikol pojem **GPGPU (General-Purpose computing on GPU)**. Znamená to používanie GPU ako všeobecného paralelného výpočtového procesora, nie iba ako grafického akcelerátora.

Moderné GPU podporujú programovacie rozhrania a jazyky ako:

- CUDA C,
- OpenCL,
- prípadne iné rozhrania pre výpočty nad GPU.

GPGPU umožňuje programátorovi spúšťať na GPU výpočty, ktoré majú veľa dátového paralelizmu. Ak je problém vhodný pre GPU, zrýchlenie môže byť veľmi výrazné oproti optimalizovanej CPU implementácii.

---

3. Prečo je GPU výkonné

GPU je výkonné najmä vtedy, keď sa veľa podobných operácií vykonáva nad veľkým množstvom dát. Typický príklad je:

A[i] = B[i] * C[i]

pre veľa hodnôt i.

Každý prvok poľa sa dá spracovať nezávisle. To znamená, že GPU môže priradiť veľa vlákien na veľa dátových prvkov a spracovať ich naraz.

GPU má vysoký výkon hlavne vďaka týmto vlastnostiam:

- veľa výpočtových jednotiek,
- vysoká priepustnosť pamäte,
- veľa súčasne bežiacich vlákien,
- schopnosť skrývať latencie prepínaním medzi vláknami,
- architektúra optimalizovaná na priepustnosť, nie na latenciu jedného vlákna.

Dôležité je zapamätať si rozdiel: **CPU sa snaží, aby jedno vlákno bežalo čo najrýchlejšie; GPU sa snaží, aby sa dokopy spracovalo čo najviac vlákien za jednotku času.**

---

4. CPU vs. GPU

CPU – latency-oriented design

CPU je optimalizované na všeobecné programy, ktoré môžu mať zložité vetvenie, nepravidelný prístup do pamäte a menší počet vlákien. Preto má CPU:

- silnú riadiacu logiku,
- veľké cache pamäte,
- zložité predikcie vetvenia,
- výkonné jadrá,
- nízku latenciu jedného vlákna.

CPU sa snaží znížiť čas vykonania jedného vlákna. Preto je výhodné pre programy, ktoré majú málo paralelizmu alebo veľa sekvenčnej logiky.

GPU – throughput-oriented design

GPU je optimalizované na priepustnosť. To znamená, že cieľom nie je, aby jedno vlákno skončilo čo najskôr, ale aby sa za jednotku času dokončilo čo najviac práce celkovo. GPU preto používa:

- veľa menších jadier,
- veľa výpočtových jednotiek,
- mnoho súčasne bežiacich vlákien,
- menší dôraz na veľké cache a zložitú riadiacu logiku,
- väčší dôraz na výpočty a priepustnosť dát.

GPU môže mať horšiu latenciu pre jednotlivé vlákno, ale vyššiu celkovú priepustnosť pri masívne paralelných úlohách. Preto sa hovorí, že GPU je vhodné najmä na úlohy, kde možno rovnaký alebo podobný výpočet aplikovať na veľké množstvo dát.

---

5. Kedy je lepšie CPU a kedy GPU

CPU je vhodnejšie, keď:

- program má málo vlákien,
- výpočet je prevažne sekvenčný,
- je veľa zložitého vetvenia,
- dátové prístupy sú nepravidelné,
- dôležitá je nízka odozva jednej úlohy,
- problém sa nedá dobre rozložiť na veľa nezávislých častí.

GPU je vhodnejšie, keď:

- máme veľa nezávislých dátových prvkov,
- nad dátami sa vykonáva rovnaký alebo podobný výpočet,
- výpočet je dátovo paralelný,
- potrebujeme vysokú priepustnosť,
- máme veľa aritmetických operácií, najmä FP výpočtov,
- program vie vytvoriť veľa vlákien.

Najjednoduchšie pravidlo: **GPU sa oplatí vtedy, keď vieme vytvoriť veľa práce rovnakého typu a rozdeliť ju medzi tisíce paralelných vlákien.**

---

6. Prečo GPU nemá takú veľkú cache a riadenie ako CPU

CPU používa veľkú časť čipu na cache pamäte a zložitú riadiacu logiku. To pomáha znižovať latenciu jednotlivých vlákien. GPU ide opačným smerom: väčšia časť tranzistorov je venovaná samotnému spracovaniu dát, teda výpočtovým jednotkám.

To má dôsledok:

- CPU je flexibilnejšie a lepšie zvláda zložité riadenie,
- GPU má vyššiu výpočtovú priepustnosť pri pravidelných paralelných úlohách.

Pri GPU sa latencie často neskrývajú veľkou cache, ale tým, že keď jedna skupina vlákien čaká na pamäť, GPU začne vykonávať inú skupinu vlákien. Na to však musí mať program dostatok paralelizmu.

---

7. GPU ako many-core procesor

GPU patrí medzi **many-core** architektúry. To znamená, že má veľmi veľa jednoduchších výpočtových jednotiek. Tieto jednotky sú usporiadané do väčších blokov, ktoré sa v CUDA terminológii nazývajú **streaming multiprocessors (SM)**.

GPU je organizované ako pole silno viacvláknových výpočtových multiprocesorov. Každý SM obsahuje viac menších výpočtových jednotiek, ktoré zdieľajú niektoré riadiace mechanizmy, inštrukčnú cache a lokálnu pamäť. GPU má zároveň vlastnú globálnu pamäť, typicky GDDR pamäť, ktorá slúži ako hlavná pamäť zariadenia.

Zjednodušene:

- GPU = viac SM,
- SM = viac výpočtových jednotiek,
- veľa vlákien sa mapuje na SM,
- vlákna v rámci SM sa vykonávajú v skupinách.

---

8. VPU a TPU

Prednáška spomína aj špecializované procesory blízke oblasti GPU.

VPU

VPU (Visual Processing Unit) je procesor určený najmä na spracovanie vizuálnych dát a strojové videnie. Netreba si ho mýliť s označením Video Processing Unit. V tomto kontexte ide skôr o akcelerátor pre algoritmy počítačového videnia a umelej inteligencie, kde vstupom aj výstupom môžu byť obrazové alebo video dáta.

TPU

TPU (Tensor Processing Unit) je špecializovaný akcelerátor pre hlboké učenie. V prednáške sa uvádza najmä v súvislosti s Googlom. TPU je ASIC, teda špecializovaný čip navrhnutý priamo na urýchlenie AI výpočtov.

Dobré zapamätanie:

- CPU – všeobecný procesor,
- GPU – masívne paralelný procesor, stále pomerne flexibilný,
- TPU – špecializovaný akcelerátor pre tensorové/deep learning výpočty,
- VPU – akcelerátor pre vizuálne spracovanie a strojové videnie.

Prednáška ukazuje hybridný prístup: CPU sa používa tam, kde výkon nie je kritický alebo kde je potrebná všeobecnosť; GPU tam, kde treba paralelný výpočet a chceme zachovať

programovateľnosť; ASIC/TPU tam, kde sa oplatí špecializovať hardvér na konkrétny typ AI výpočtov.

---

9. Stručná história GPU

Za dôležitý bod vo vývoji GPU sa považuje rok 1999, keď NVIDIA uviedla GeForce 256. Tá bola prezentovaná ako GPU pre PC priemysel a definícia GPU sa spájala s jedným čipom, ktorý má integrované jednotky na transformáciu, osvetlenie, prácu s trojuholníkmi a rendering.

V rokoch 1999–2000 sa GPU začali používať aj mimo grafiky. Vedci a výskumníci z rôznych oblastí si všimli, že GPU dokáže výrazne zrýchliť niektoré vedecké výpočty. Problém bol však v tom, že sa GPU muselo programovať cez grafické API, napríklad OpenGL alebo Cg, čo výrazne sťažovalo používanie GPU na všeobecné výpočty.

Toto viedlo k vývoju programovacích modelov ako CUDA a OpenCL, ktoré umožnili programovať GPU prirodzenejšie ako paralelný výpočtový procesor.

---

10. Typické oblasti použitia GPU

Hry

GPU je prirodzene vhodné na hry, pretože moderné hry vyžadujú veľa grafických výpočtov: spracovanie geometrie, textúr, tieňov, osvetlenia, fyziky a ďalších efektov. CPU by tieto úlohy vo veľkom rozsahu nezvládalo dostatočne rýchlo, preto sa graficky intenzívne časti presúvajú na GPU.

3D vizualizácia a CAD

V CAD a 3D vizualizácii treba v reálnom čase zobrazovať a manipulovať s veľmi zložitými modelmi. Môže ísť o budovy, mosty, stroje alebo celé scény s obrovským počtom trojuholníkov. GPU je vhodné, pretože vie paralelne spracovávať veľké množstvo geometrických a obrazových operácií.

Spracovanie obrazu

Algoritmy spracovania obrazu často vykonávajú rovnakú operáciu nad veľkým počtom pixelov. Napríklad filtrovanie, rozmazanie, detekcia hrán, transformácie alebo rekonštrukcie obrazu sú prirodzene dátovo paralelné. Preto sa GPU používa aj v medicínskom zobrazovaní, bezpečnosti, röntgenovom spracovaní a podobných oblastiach.

Big Data

Pri veľkých dátach sa GPU používa najmä tam, kde treba rýchlo spracovať veľké objemy údajov, vizualizovať ich alebo analyzovať vzťahy medzi veľkým množstvom prvkov. Prednáška spomína napríklad genomické alebo neuroimagingové úlohy, kde GPU pomáha spracovať veľké dátové súbory.

Deep Machine Learning

GPU je veľmi dôležité pre strojové učenie a hlboké neurónové siete. Trénovanie neurónových sietí obsahuje veľa maticových a vektorových operácií, ktoré sa výborne hodia na dátový paralelizmus. Preto sa GPU používa pri spracovaní obrazu, videa, reči, prirodzeného jazyka, autonómnych vozidlách a počítačovom videní.

---

11. Programovanie GPU

GPU sa zvyčajne nepoužíva samo. V bežnom systéme máme CPU ako **host** a GPU ako **device**. Program beží najprv na CPU a CPU potom spúšťa výpočtové časti na GPU.

Grafická karta obsahuje:

- programovateľné výpočtové jednotky,
- rôzne typy pamäte a cache,
- niektoré pevne dané funkčné jednotky pre grafické úlohy,
- globálnu pamäť zariadenia.

Hardvérové operácie GPU riadi program bežiaci na CPU cez API. Program môže byť písaný napríklad v CUDA C, OpenCL alebo v grafických jazykoch typu Cg/HLSL.

---

12. Rôzne pohľady na GPU podľa programovacieho prostredia

GPU možno programátorovi „ukázať“ rôznymi spôsobmi.

Grafické API

Grafické API prirodzene prezentuje GPU ako pipeline alebo stream procesor. To sedí grafickým aplikáciám, kde dáta prechádzajú postupnosťou grafických fáz.

CUDA / OpenCL

CUDA alebo OpenCL prezentujú GPU ako kolekciu multiprocesorov. Každý multiprocesor možno chápať ako široký SIMD procesor zložený zo skalárnych jednotiek. Tieto jednotky vykonávajú tú istú operáciu nad rôznymi dátami.

Toto je veľmi dôležitá myšlienka: **GPU je z programátorského hľadiska veľa paralelných vlákien, ale z hardvérového hľadiska ide často o SIMD/SIMT vykonávanie skupín vlákien.**

---

13. Kernel

Výpočtový program spúšťaný na GPU sa často nazýva **kernel**. Kernel je funkcia, ktorú spustí CPU na GPU a ktorú potom vykonáva veľké množstvo GPU vlákien.

Typický kernel predstavuje jeden nezávislý výpočtový krok, ktorý sa aplikuje nad veľkým množstvom dát. Napríklad pri sčítaní dvoch vektorov môže každé vlákno počítať jeden prvok výsledného vektora.

GPU preferuje:

- krátke výpočty,
- kompaktné dátové bloky,
- veľa nezávislých krokov,
- koherentné správanie vlákien,
- pravidelný prístup k dátam.

Najväčšou výzvou pri prenose algoritmu na GPU je rozložiť ho na veľa nezávislých výpočtových krokov.

---

14. CUDA-capable GPU – základná architektúra

CUDA-schopné GPU je organizované ako pole **streaming multiprocessors (SM)**. Každý SM obsahuje viac výpočtových jednotiek, často označovaných ako streaming processors alebo CUDA cores. Tieto jednotky zdieľajú určitú riadiacu logiku a inštrukčnú cache.

GPU má zároveň veľkú globálnu pamäť, typicky GDDR SDRAM, ktorá sa označuje ako **global memory**. Táto pamäť je dostupná pre výpočty na GPU, ale má vyššiu latenciu než lokálna alebo zdieľaná pamäť v SM.

Zjednodušený model:

- CPU spustí kernel,
- kernel vytvorí množstvo vlákien,
- vlákna sú rozdelené do blokov,
- bloky sú priradené na SM,
- SM vykonáva vlákna v skupinách,
- dáta sa čítajú a zapisujú cez rôzne úrovne pamäte.

---

15. Thread, block, grid

Toto je jedna z najdôležitejších častí celej prednášky.

Thread

Thread je jedno GPU vlákno. Typicky je priradené k jednému dátovému prvku alebo k malej časti dát. Napríklad jedno vlákno môže počítať jeden prvok vektora alebo jeden pixel obrazu.

Thread block

Vlákna sú organizované do **blokov**. Blok je skupina vlákien, ktoré môžu spolupracovať. Vlákna v jednom bloku môžu používať rýchlu zdieľanú pamäť a synchronizovať sa medzi sebou.

Grid

Bloky sú organizované do **gridu**. Grid predstavuje celú množinu blokov spustených pri jednom volaní kernelu.

Zjednodušene:

kernel spustí grid
grid obsahuje bloky
blok obsahuje vlákna
vlákno spracuje konkrétnu časť dát

---

16. Prečo bloky nemôžu ľubovoľne spolupracovať

Vlákna v jednom bloku môžu spolupracovať:

- môžu sa synchronizovať,
- môžu používať shared memory,
- môžu si vymieňať dáta cez nízkolatenčnú pamäť.

Vlákna z rôznych blokov však bežne nemôžu priamo spolupracovať takýmto spôsobom. Dôvod je architektonický: bloky môžu byť spustené na rôznych SM a ich plánovanie je riadené hardvérom. Programátor nemá garantované, že dva bloky budú bežať v rovnakom čase alebo na blízkych jednotkách.

Preto treba algoritmus navrhovať tak, aby bloky boli čo najviac nezávislé. Ak je medzi blokmi nutná komunikácia, často sa rieši cez globálnu pamäť a oddelené spustenia kernelov.

---

17. Streaming Multiprocessor (SM)

SM (Streaming Multiprocessor) je základná výpočtová jednotka GPU na vyššej úrovni. GPU obsahuje viac SM a každý SM vykonáva bloky vlákien.

SM obsahuje:

- výpočtové jednotky,
- riadiacu logiku,
- plánovače vlákien,
- registre,
- shared memory / L1 cache,
- inštrukčnú cache.

Vlákna sa na SM nespúšťajú jednotlivo úplne nezávisle, ale organizujú sa do skupín, ktoré sa vykonávajú spolu. Pri NVIDIA architektúrach sa takáto skupina typicky nazýva **warp**.

---

18. Warp a SIMD lanes

Prednáška vysvetľuje, že blok priradený na SM sa ďalej delí na jednotky po 32 vláknach, ktoré sa nazývajú **warps**. Veľkosť warpu je implementačný detail, ale pri NVIDIA sa typicky používa 32 vlákien.

Warp sa vykonáva na SIMD výpočtových dráhach, ktoré sa nazývajú **SIMD lanes**. Všetky vlákna vo warpe vykonávajú tú istú inštrukciu, ale nad rôznymi dátami. To je dôvod, prečo sa GPU označuje ako SIMT – Single Instruction, Multiple Threads.

Dôležité:

- warp = skupina vlákien vykonávaných spolu,
- SIMD lanes = fyzické výpočtové dráhy,
- všetky vlákna vo warpe idú rovnakou inštrukčnou cestou,
- ak sa vlákna vo warpe rozvetvia rôznymi smermi, výkon môže klesnúť.

---

19. SIMT model

CUDA používa programovací model **SIMT (Single Instruction Multiple Thread)**. Je podobný SIMD, ale programátor píše kód ako keby pre jednotlivé vlákna. Hardvér potom vlákna zoskupuje do warpov a vykonáva ich spolu.

Rozdiel:

- SIMD: programátor často priamo myslí vo vektoroch,
- SIMT: programátor píše kód pre jedno vlákno, ale hardvér spúšťa veľa vlákien naraz.

To je pohodlnejšie, pretože programátor môže písať prirodzenejší kód. Musí si však uvedomovať, že hardvér stále vykonáva skupiny vlákien spolu. Preto vetvenie vo warpe a nepravidelné prístupy do pamäte môžu zhoršiť výkon.

---

20. Plánovanie na GPU

Prednáška rozlišuje dve úrovne plánovania:

Thread Block Scheduler

Tento plánovač priraďuje bloky vlákien na SM. Programátor pri spustení kernelu určí, koľko blokov sa má spustiť, a hardvér ich rozdelí medzi dostupné SM.

SIMD Thread Scheduler

V rámci každého SM existuje plánovač, ktorý rozhoduje, ktorý warp alebo skupina SIMD inštrukcií sa bude práve vykonávať. Ak jeden warp čaká na pamäť, plánovač môže vybrať iný pripravený warp.

Toto je kľúčový mechanizmus skrývania latencie na GPU. GPU sa nesnaží čakať na jedno vlákno, ale prepína medzi veľkým množstvom pripravených warpov.

---

21. Pamäťový model GPU

GPU má viac typov pamäte, ktoré sa líšia rýchlosťou, veľkosťou a dostupnosťou.

Globálna pamäť

Globálna pamäť je veľká pamäť GPU. Je dostupná všetkým vláknam, ale má vyššiu latenciu. Dáta sa do nej typicky kopírujú z host pamäte a po výpočte sa výsledky kopírujú späť.

Lokálna pamäť vlákna

Každé vlákno môže mať vlastnú lokálnu pamäť. Používa sa pre dáta patriace konkrétnemu vláknu.

Registre

Registre sú najrýchlejšia pamäť dostupná vláknu. Sú na čipe a slúžia na dočasné hodnoty.

Shared memory

Shared memory je rýchla pamäť zdieľaná vláknami v jednom bloku. Je nízkolatenčná a používa sa na spoluprácu vlákien v bloku.

Constant memory

Constant memory je pamäť určená na čítanie, vhodná pre hodnoty, ktoré sa počas kernelu nemenia.

Texture memory

Texture memory je tiež čítacia pamäť optimalizovaná pre určité typy prístupov, historicky spojená s grafikou.

Dôležité zapamätanie: **globálna pamäť je veľká, ale pomalšia; registre a shared memory sú rýchle, ale obmedzené. Efektívny GPU program musí vedieť pracovať s touto hierarchiou.**

---

22. Prečo je znalosť architektúry pri GPU dôležitá

Pri CPU môže programátor často napísať relatívne všeobecný kód a spoliehať sa na cache, predikciu vetvenia a optimalizácie procesora. Pri GPU to nefunguje tak dobre. Prednáška zdôrazňuje, že efektívny GPU program sa nedá napísať bez znalosti architektúry.

Dôvody:

- pamäťové latencie treba skrývať množstvom vlákien,
- globálna pamäť je pomalšia než shared memory,
- bloky majú obmedzený počet vlákien,
- SM má obmedzený počet aktívnych blokov a vlákien,
- vetvenie v rámci warpu znižuje efektivitu,
- príliš malé bloky nevyužijú SM,
- príliš veľké bloky môžu prekročiť limity hardvéru.

---

23. Príklad: násobenie vektorov A = B * C

Prednáška uvádza príklad násobenia dvoch vektorov dĺžky 8192:

A = B * C

To znamená, že pre každý index i počítame:

A[i] = B[i] * C[i]

Tento problém je ideálny pre GPU, pretože každý prvok výsledku sa dá vypočítať nezávisle.

V príklade:

- vektor má 8192 prvkov,
- blok má 512 vlákien,
- grid má 16 blokov, pretože 8192 / 512 = 16,
- SIMD inštrukcia spracuje 32 prvkov naraz,
- jeden blok teda obsahuje 512 / 32 = 16 SIMD skupín.

Význam príkladu:

- grid reprezentuje celý výpočet,
- blok rozdeľuje výpočet na menšie časti,
- warp/SIMD skupina spracúva časť bloku,
- každé vlákno pracuje nad konkrétnym dátovým prvkom.

Toto je typický spôsob, ako sa dátovo paralelný problém mapuje na GPU.

---

24. Príklad: image blur a výber veľkosti bloku

Prednáška rieši príklad rozmazania obrazu, kde CUDA zariadenie umožňuje:

- maximálne 8 blokov na SM,
- maximálne 1024 vlákien na SM,
- maximálne 512 vlákien v jednom bloku.

Porovnávajú sa bloky:

- 8 × 8,
- 16 × 16,
- 32 × 32.

Blok 8 × 8

Blok má 64 vlákien. Aby sa zaplnilo 1024 vlákien na SM, potrebovali by sme 16 takých blokov, ale SM povoľuje len 8 blokov. To znamená, že na SM bude iba 8 × 64 = 512 vlákien. SM teda nebude plne vyťažený.

Blok 16 × 16

Blok má 256 vlákien. Na 1024 vlákien na SM stačia 4 bloky. To je v limite 8 blokov aj v limite 1024 vlákien. Preto je to vhodná konfigurácia.

Blok 32 × 32

Blok má 1024 vlákien. To prekračuje limit 512 vlákien na blok, takže taká konfigurácia nie je povolená.

Záver príkladu: **16 × 16 je najvhodnejšia konfigurácia z uvedených možností, pretože umožní vysoký počet aktívnych vlákien bez prekročenia limitov zariadenia.**

---

25. Occupancy a využitie SM

Aj keď prednáška nemusí používať slovo occupancy vždy priamo, príklad s image blur ho dobre vysvetľuje. Occupancy znamená, ako dobre sú výpočtové zdroje SM obsadené aktívnymi vláknami alebo warpmi.

Ak je aktívnych príliš málo vlákien:

- SM môže byť nevyužitý,
- je menej warpov na prepínanie,
- horšie sa skrývajú pamäťové latencie.

Ak je blok príliš veľký:

- môže prekročiť limit,
- môže znížiť počet blokov na SM,
- môže spotrebovať priveľa registrov alebo shared memory.

Preto pri GPU nestačí len „dať čo najviac vlákien“. Treba zvoliť takú veľkosť bloku, ktorá rešpektuje limity hardvéru a zároveň dáva SM dosť práce.

---

26. GeForce 20 series a Turing

Prednáška v závere uvádza architektúru NVIDIA GeForce 20 series, založenú na mikroarchitektúre Turing. Táto generácia priniesla najmä dôraz na real-time ray tracing a nové typy výpočtových jednotiek.

Medzi spomenuté vlastnosti Turing patria:

- CUDA Compute Capability 7.5,
- nový Streaming Multiprocessor,
- Turing Tensor Cores,
- Deep Learning Super Sampling (DLSS),
- real-time ray tracing acceleration,
- nové shading techniky,
- GDDR6 pamäťový subsystém,
- druhá generácia NVIDIA NVLink.

Zmysel tejto časti nie je len zapamätať si názvy funkcií, ale pochopiť trend: moderné GPU už nie je iba univerzálna množina CUDA jadier. Obsahuje aj špecializované jednotky na AI a ray tracing, napríklad Tensor Cores a RT Cores.

---

27. Turing TU102 ako príklad modernej GPU architektúry

Prednáška uvádza konkrétny príklad GPU Turing TU102. Obsahuje:

- 4608 CUDA jadier,
- 72 RT jadier,
- 576 Tensor jadier,
- 288 textúrovacích jednotiek,
- 12 32-bitových GDDR6 pamäťových radičov, spolu 384-bitovú pamäťovú zbernicu.

Turing SM je rozdelený na štyri spracovateľské bloky. Každý blok obsahuje:

- 16 FP32 jadier,
- 16 INT32 jadier,
- 2 Tensor Cores,
- jeden warp scheduler,
- jednu dispatch jednotku,
- L0 instruction cache,
- 64 KB register file.

Štyri bloky spolu zdieľajú 96 KB L1 data cache / shared memory.

Dôležitá myšlienka: moderný SM už nie je len jednoduchá skupina rovnakých ALU. Obsahuje viac typov výpočtových jednotiek, plánovače, cache, registre a špecializované bloky pre rôzne druhy výpočtov.

---

28. Prečo sa GPU hodí najmä na dátový paralelizmus

GPU je podobné vektorovým architektúram v tom, že najlepšie funguje pri problémoch s vysokým dátovým paralelizmom. To znamená, že máme veľa dátových prvkov a nad každým prvkom sa vykonáva rovnaký alebo podobný výpočet.

Typické vhodné úlohy:

- vektorové operácie,
- maticové operácie,
- spracovanie pixelov,
- filtrovanie obrazu,
- simulácie na mriežkach,

- neurónové siete,
- veľké množstvo nezávislých výpočtov.

Nevhodné alebo menej vhodné úlohy:

- silno sekvenčné algoritmy,
- nepravidelné vetvenie,
- malé množstvo dát,
- častá komunikácia medzi vláknami,
- závislosti medzi krokmi,
- nepravidelný prístup do pamäte.

---

29. Hlavné obmedzenia GPU

GPU má veľký výkon, ale nie je univerzálne riešenie na všetko.

Problém musí mať dosť paralelizmu

Ak program nevie vytvoriť veľa nezávislých vlákien, GPU sa nevyužije.

Prenos dát medzi CPU a GPU má réžiu

Ak je výpočet malý, prenos dát na GPU a späť môže stáť viac času než samotný výpočet.

Vetvenie vo warpe znižuje výkon

Ak rôzne vlákna vo warpe idú rôznymi vetvami programu, hardvér musí tieto vetvy často vykonať postupne.

Globálna pamäť má vysokú latenciu

Program musí dobre využívať pamäťovú hierarchiu, najmä shared memory, registre a koherentný prístup do pamäte.

Bloky majú hardvérové limity

Počet vlákien na blok, počet blokov na SM, počet registrov a shared memory sú obmedzené.

---

30. Ako rozmýšľať pri návrhu GPU programu

Pri návrhu algoritmu pre GPU sa oplatí postupovať takto:

1. Nájsť dátový paralelizmus

Treba nájsť časti výpočtu, kde sa rovnaká operácia vykonáva nad veľkým množstvom prvkov.

2. Priradiť vlákna dátovým prvkom

Typicky jedno vlákno spracuje jeden prvok, pixel, bunku mriežky alebo časť výpočtu.

3. Rozdeliť vlákna do blokov

Bloky by mali mať vhodnú veľkosť, aby dobre využili SM a neprekročili limity zariadenia.

4. Minimalizovať komunikáciu medzi blokmi

Bloky by mali byť čo najviac nezávislé, pretože priama spolupráca medzi blokmi je obmedzená.

5. Využívať shared memory tam, kde sa oplatí

Ak viac vlákien v bloku opakovane používa rovnaké dáta, oplatí sa ich preniesť do shared memory.

6. Dávať pozor na prístup do globálnej pamäte

Pravidelný a koherentný prístup do pamäte je pre výkon GPU veľmi dôležitý.

7. Vyhýbať sa zbytočnému vetveniu

Ak vlákna v tom istom warpe vykonávajú rôzne vetvy, výkon môže klesnúť.

---

31. Najdôležitejšie pojmy na zapamätanie

GPU je vysoko paralelný mnohojadrový procesor pôvodne určený na grafiku, dnes používaný aj na všeobecné výpočty.

GPGPU znamená využitie GPU na všeobecné výpočty, nielen na grafiku.

CPU je optimalizované na nízku latenciu a všeobecné sekvenčné programy.

GPU je optimalizované na vysokú priepustnosť a masívny dátový paralelizmus.

Kernel je funkcia spustená na GPU veľkým množstvom vlákien.

Thread je jedno GPU vlákno, často priradené jednému dátovému prvku.

Thread block je skupina vlákien, ktoré môžu spolupracovať a používať shared memory.

Grid je množina blokov spustených jedným kernelom.

SM (Streaming Multiprocessor) je výpočtový blok GPU, ktorý vykonáva bloky vlákien.

Warp je skupina typicky 32 vlákien vykonávaných spolu.

SIMT znamená Single Instruction Multiple Thread – programátor píše kód pre vlákno, ale hardvér vykonáva skupiny vlákien spolu.

Shared memory je rýchla pamäť zdieľaná vláknami v jednom bloku.

Global memory je veľká pamäť GPU dostupná všetkým vláknam, ale s vyššou latenciou.

Occupancy vyjadruje, ako dobre sú zdroje SM obsadené aktívnymi vláknami/warpmi.

Tensor Cores sú špecializované jednotky na maticové a AI výpočty.

RT Cores sú špecializované jednotky na ray tracing.

---

32. Hlavná myšlienka celej prednášky

Prednáška 11.a ukazuje, že GPU je špecializovaná paralelná architektúra orientovaná na priepustnosť. Je veľmi výkonná vtedy, keď problém obsahuje veľa dátového paralelizmu a dá sa rozdeliť na veľké množstvo nezávislých vlákien. GPU nie je náhrada CPU vo všetkom. CPU je vhodné na všeobecný riadiaci a sekvenčný výpočet, GPU je vhodné na masívne paralelné výpočty nad veľkým množstvom dát.

Kľúčové pochopenie je toto: **GPU získava výkon nie tým, že jedno vlákno vykoná extrémne rýchlo, ale tým, že naraz spracúva obrovské množstvo vlákien a tým dosahuje vysokú celkovú priepustnosť.**

Preto pri GPU nestačí len vedieť napísať program. Treba rozumieť tomu, ako sa výpočet mapuje na vlákna, bloky, gridy, SM, warpy a pamäťovú hierarchiu. Efektívny GPU program vzniká až vtedy, keď algoritmus dobre zodpovedá architektúre GPU.

Prednáška 11-b: CUDA

1. Čo je CUDA

CUDA znamená **Compute Unified Device Architecture**. Je to výpočtová architektúra a programovací model od NVIDIA, ktorý umožňuje programovať GPU ako paralelný výpočtový

procesor. CUDA vznikla preto, aby sa GPU nemuselo používať iba cez grafické API, ale aby sa dalo programovať podobnejšie ako bežný paralelný výpočtový systém.

CUDA používa **heterogénny výpočtový model**. To znamená, že v jednom programe spolupracujú dva typy výpočtových zariadení:

- **CPU** sa nazýva **host**,
- **GPU** sa nazýva **device**.

CPU riadi priebeh programu, pripravuje dáta, spúšťa výpočty na GPU a po skončení si berie výsledky späť. GPU vykonáva tie časti programu, ktoré obsahujú veľa dátového paralelizmu.

---

2. CUDA ako SIMT model

Programovací model CUDA je označený ako **SIMT – Single Instruction, Multiple Thread**. Znamená to, že programátor píše kód z pohľadu jedného vlákna, ale pri spustení sa ten istý kód vykoná vo veľkom počte vlákien nad rôznymi dátami.

Je to podobné ako SIMD, ale pre programátora prirodzenejšie:

- pri SIMD často rozmýšľame vo vektoroch,
- pri SIMT píšeme kód pre jedno vlákno,
- GPU potom spustí veľa vlákien, ktoré vykonávajú rovnaký kernel nad rôznymi časťami dát.

Príklad myslenia v CUDA:

```
out[tid] = a[tid] + b[tid];
```

Každé vlákno má svoje `tid` a vypočíta jeden prvok výsledného poľa.

---

3. Štruktúra CUDA C programu

CUDA C program odráža fakt, že v počítači existuje CPU aj GPU. Preto môže jeden CUDA zdrojový súbor obsahovať:

- **host code** – kód vykonávaný na CPU,
- **device code** – kód vykonávaný na GPU.

Host kód je väčšinou obyčajný C/C++ kód. Device kód je označený špeciálnymi CUDA kľúčovými slovami, aby bolo jasné, že sa má preložiť pre GPU.

Keď sa do súboru pridajú CUDA kľúčové slová a device funkcie, už ho nemožno prekladať obyčajným C prekladačom. Používa sa špeciálny prekladač:

NVCC – NVIDIA C Compiler

NVCC vie spracovať súbor, ktorý obsahuje host aj device kód. Host časť sa preloží pre CPU a device časť pre GPU.

---

4. Ako prebieha vykonávanie CUDA programu

CUDA program sa vždy začína vykonávať na CPU. CPU vykonáva sériový host kód. Keď program narazí na spustenie GPU funkcie, teda kernelu, CPU pošle prácu na GPU.

Typický priebeh CUDA programu:

1. CPU alokuje pamäť pre vstupné a výstupné dáta.
2. CPU pripraví vstupné dáta.
3. CPU alokuje pamäť na GPU.
4. CPU skopíruje vstupné dáta z host pamäte do device pamäte.
5. CPU spustí kernel na GPU.
6. GPU vykoná paralelný výpočet veľkým počtom vlákien.
7. CPU skopíruje výsledky z GPU späť do host pamäte.
8. CPU overí alebo ďalej spracuje výsledky.
9. Program uvoľní host aj device pamäť.

Toto je dôležité: GPU nie je samostatný „hlavný procesor“ programu. GPU vykonáva paralelné výpočtové časti, ktoré mu spúšťa CPU.

---

5. Kernel

Kernel je funkcia, ktorá sa vykonáva na GPU. V CUDA sa kernel označuje kľúčovým slovom:

__global__

Príklad:

```
__global__ void vector_add(float *out, float *a, float *b, int n) {
   int tid = blockIdx.x * blockDim.x + threadIdx.x;

   if (tid < n) {
      out[tid] = a[tid] + b[tid];
   }
}
```

Táto funkcia nebeží iba raz. Keď sa kernel spustí, vykonáva ho veľké množstvo CUDA vlákien. Každé vlákno vykoná ten istý kód, ale s iným identifikátorom tid. Preto každé vlákno pracuje s iným prvkom poľa.

---

6. Spustenie kernelu

Kernel sa z host kódu spúšťa špeciálnou syntaxou:

```
vector_add<<<grid_size, block_size>>>(d_out, d_a, d_b, N);
```

Význam:

- vector_add je kernel,
- grid_size určuje počet blokov,
- block_size určuje počet vlákien v jednom bloku,
- argumenty v zátvorkách sú bežné parametre kernelu.

Táto syntax je špecifická pre CUDA. Hovorí GPU, koľko blokov a koľko vlákien má vytvoriť. V prednáške je príklad s veľkosťou bloku 256 vlákien.

---

7. Grid, block, thread

CUDA organizuje vlákna hierarchicky. Táto časť je úplný základ CUDA programovania.

Thread

Thread je jedno CUDA vlákno. Každé vlákno vykonáva kernel a typicky spracúva jednu časť dát, napríklad jeden prvok poľa.

Block

Block je skupina vlákien. Počet vlákien v bloku určuje host kód pri spustení kernelu. Rovnaký kernel možno v rôznych častiach programu spustiť s rôznym počtom vlákien v bloku.

Grid

Grid je množina všetkých blokov vytvorených pri jednom spustení kernelu. Všetky vlákna, ktoré vzniknú jedným spustením kernelu, sa spoločne nazývajú grid.

Jednoduché zapamätanie:

kernel launch → grid
grid → blocks

block → threads
thread → spracuje konkrétnu časť dát

---

8. Vstavané premenné blockDim, threadIdx, blockIdx

CUDA kernel má prístup k vstavaným premenným, pomocou ktorých každé vlákno zistí, kto je a akú časť dát má spracovať.

threadIdx

Premenná threadIdx dáva vláknu jeho súradnicu v rámci bloku. Ak pracujeme v jednom rozmere, často používame:

threadIdx.x

blockIdx

Premenná blockIdx dáva súradnicu bloku v rámci gridu. Ak pracujeme v jednom rozmere, často používame:

blockIdx.x

blockDim

Premenná blockDim obsahuje počet vlákien v bloku. Pri jednom rozmere používame:

blockDim.x

Tieto premenné umožňujú každému vláknu vypočítať globálny index dátového prvku, ktorý má spracovať.

---

9. Výpočet globálneho indexu vlákna

Pri jednorozmernom poli sa globálny index vlákna často počíta takto:

```
int tid = blockIdx.x * blockDim.x + threadIdx.x;
```

Význam:

- blockIdx.x hovorí, v ktorom bloku sme,
- blockDim.x hovorí, koľko vlákien je v jednom bloku,
- threadIdx.x hovorí, ktoré vlákno v bloku sme.

Ak má blok 256 vlákien:

- blok 0 má vlákna s globálnymi indexmi 0 až 255,

- blok 1 má vlákna s globálnymi indexmi 256 až 511,
- blok 2 má vlákna s globálnymi indexmi 512 až 767.

Preto vzorec:

```
tid = blockIdx.x * blockDim.x + threadIdx.x;
```

priradí každému vláknu jedinečný index v rámci celého gridu.

---

10. Prečo sa v kerneli používa podmienka `if (tid < n)`

Pri spustení kernelu často počet vlákien zaokrúhľujeme nahor, aby pokryl celé pole. Ak veľkosť poľa nie je presne deliteľná veľkosťou bloku, vzniknú aj vlákna, ktorých index je mimo rozsahu dát.

Preto sa používa ochranná podmienka:

```
if (tid < n) {
   out[tid] = a[tid] + b[tid];
}
```

Táto podmienka zabezpečí, že vlákna s indexom mimo poľa neurobia neplatný prístup do pamäte.

Toto je veľmi dôležitý praktický detail CUDA programovania. Kernel často spúšťame s väčším počtom vlákien, než je presný počet dátových prvkov, a preto musí každé vlákno skontrolovať, či naozaj patrí do platného rozsahu.

---

11. Výpočet veľkosti gridu

Ak máme `N` prvkov a jeden blok má `block_size` vlákien, počet blokov sa vypočíta tak, aby pokryl všetky prvky.

V prednáške sa používa zápis:

```
int block_size = 256;
int grid_size = ((N + block_size) / block_size);
```

Presnejší bežný zápis býva:

```
int grid_size = (N + block_size - 1) / block_size;
```

Cieľ je zaokrúhliť delenie nahor. Napríklad ak máme 1000 prvkov a blok má 256 vlákien, potrebujeme 4 bloky, pretože 3 bloky by pokryli iba 768 prvkov.

Dôležité je pochopiť princíp: **grid musí obsahovať dosť vlákien na spracovanie všetkých prvkov, aj keby posledný blok nebol úplne využitý.**

---

12. Spolupráca vlákien v bloku

Vlákna v tom istom bloku môžu spolupracovať. Prednáška uvádza dve hlavné formy spolupráce:

- synchronizácia vykonávania,
- zdieľanie dát cez nízkolatenčnú shared memory.

To znamená, že ak vlákna v jednom bloku potrebujú spoločne spracovať určitú časť dát, môžu si ich uložiť do zdieľanej pamäte a synchronizovať sa.

Dôležité obmedzenie:

Vlákna z rôznych blokov nemôžu priamo spolupracovať takýmto spôsobom.

Preto sa CUDA algoritmy často navrhujú tak, aby bol každý blok čo najviac samostatný. Ak treba komunikovať medzi blokmi, často sa to robí cez globálnu pamäť a samostatné spustenia kernelov.

---

13. Prečo rôzne bloky nemôžu jednoducho spolupracovať

Bloky môžu byť naplánované na rôzne SM, môžu sa vykonávať v rôznom čase a CUDA negarantuje, že všetky bloky bežia súčasne. Preto nie je bezpečné navrhovať kernel tak, že blok A čaká priamo na blok B v rámci toho istého kernelu.

Správne uvažovanie:

- vlákna v jednom bloku môžu spolupracovať,
- bloky v jednom gride by mali byť do veľkej miery nezávislé,
- globálna synchronizácia medzi blokmi sa typicky rieši ukončením kernelu a spustením ďalšieho kernelu.

Toto je jeden z najdôležitejších rozdielov medzi CPU vláknami a CUDA blokmi.

---

14. Pamäťový model CUDA

CUDA rozlišuje viac druhov pamäte. Každý typ má inú rýchlosť, veľkosť a dostupnosť. Efektívny CUDA program musí vedieť, kam dáta ukladať a kto k nim môže pristupovať.

Device kód môže čítať a zapisovať:

- per-thread registre,
- per-thread local memory,
- per-block shared memory,
- per-grid global memory.

Device kód môže čítať:

- per-grid constant memory,
- per-grid texture memory.

Host môže čítať a zapisovať:

- global memory,
- constant memory,
- texture memory.

---

15. Registre

Registre sú najrýchlejšia pamäť dostupná vláknu. Sú súkromné pre každé vlákno. Používajú sa na lokálne premenné, medzivýsledky a hodnoty, ktoré vlákno často potrebuje.

Vlastnosti:

- veľmi rýchle,
- uložené na čipe,
- súkromné pre jedno vlákno,
- obmedzený počet.

Ak kernel používa príliš veľa registrov, môže to znížiť počet aktívnych vlákien na SM. Preto aj registre ovplyvňujú výkon.

---

16. Local memory

Local memory je tiež súkromná pre vlákno, ale nemusí byť taká rýchla ako registre. Používa sa napríklad vtedy, keď sa lokálne dáta vlákna nezmestia do registrov alebo keď kompilátor potrebuje uložiť určité hodnoty mimo registrov.

Názov „local“ môže byť trochu mätúci. Neznamená automaticky „rýchla lokálna pamäť“. Znamená hlavne to, že patrí jednému vláknu.

---

17. Shared memory

Shared memory je rýchla pamäť zdieľaná vláknami v jednom bloku. Nachádza sa na čipe, podobne ako registre, a má oveľa nižšiu latenciu než globálna pamäť.

Použitie shared memory:

- dočasné uloženie dát, ktoré používa viac vlákien v bloku,
- zníženie počtu prístupov do globálnej pamäte,
- spolupráca vlákien v bloku,
- blokové algoritmy, napríklad dlaždicové násobenie matíc.

Dôležité pravidlo:

- shared memory vidia iba vlákna v tom istom bloku,
- vlákna z iného bloku k nej nemajú prístup.

---

18. Global memory

Global memory je hlavná pamäť GPU. Je veľká a prístupná všetkým vláknam v celom gride, ale má vyššiu latenciu než registre alebo shared memory.

Dáta z CPU sa pred výpočtom zvyčajne kopírujú do global memory. Po skončení výpočtu sa výsledky kopírujú späť z global memory do host pamäte.

Vlastnosti:

- veľká kapacita,
- dostupná všetkým vláknam,
- čitateľná aj zapisovateľná device kódom,
- pomalšia než registre a shared memory,
- spravovaná cez funkcie ako cudaMalloc, cudaMemcpy, cudaFree.

Najväčšia výkonová chyba pri CUDA programoch býva často zlá práca s globálnou pamäťou.

---

19. Constant memory

Constant memory je pamäť určená na čítanie zo strany device kódu. Je vhodná pre hodnoty, ktoré sa počas výpočtu nemenia a číta ich veľa vlákien.

V prednáške sa uvádza, že constant memory podporuje krátkolatenčný a vysokopriepustný read-only prístup zo strany device.

Príklady použitia:

- konštantné koeficienty,
- parametre výpočtu,
- malé tabuľky hodnôt, ktoré sa nemenia.

---

20. Texture memory

Texture memory je pamäť historicky spojená s grafickým spracovaním, ale dá sa použiť aj vo výpočtových úlohách. Je určená na čítanie a môže byť výhodná pri určitých typoch prístupov k dátam.

V CUDA pamäťovom modeli patrí medzi per-grid read-only pamäte dostupné device kódu.

Pre základné pochopenie stačí vedieť, že texture memory je špeciálna čítacia pamäť s vlastnými optimalizáciami, najmä pre dátové prístupy podobné grafickým alebo priestorovým vzorom.

---

21. Kto môže pristupovať ku ktorej pamäti

Pre učenie je veľmi užitočné zapamätať si túto logiku:

Typ pamäte	Kto ju používa	Viditeľnosť	Typické použitie
registre	device	jedno vlákno	lokálne premenné
local memory	device	jedno vlákno	súkromné dáta vlákna
shared memory	device	jeden blok	spolupráca vlákien v bloku
global memory	host aj device	celý grid / celé GPU	vstupy, výstupy, veľké polia
constant memory	host nastaví, device číta	celý grid	konštanty
texture memory	host nastaví, device číta	celý grid	špeciálne read-only prístupy

Základná myšlienka: **čím je pamäť bližšie k vláknu, tým je rýchlejšia, ale menšia a menej zdieľaná. Čím je pamäť globálnejšia, tým je väčšia, ale zvyčajne pomalšia.**

---

22. Správa device pamäte: cudaMalloc

Na alokáciu pamäte na GPU sa používa:

```
cudaMalloc((void**)&d_a, sizeof(float) * N);
```

cudaMalloc sa volá z host kódu a alokuje objekt v global memory zariadenia. Má dva hlavné parametre:

- adresu ukazovateľa, do ktorého sa uloží adresa alokovaného objektu na device,
- veľkosť alokovaného objektu v bajtoch.

Preto sa často používa zápis:

```
float *d_a;
cudaMalloc((void**)&d_a, sizeof(float) * N);
```

Premenná d_a je ukazovateľ na pamäť na GPU. Písmeno d_ sa často používa ako konvencia pre device pointer.

---

23. Uvoľnenie device pamäte: cudaFree

Pamäť alokovanú cez cudaMalloc treba po skončení použiť uvoľniť:

```
cudaFree(d_a);
```

Ak program alokuje viac polí na GPU, každé treba uvoľniť:

```
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_out);
```

Je to analogické k malloc a free na CPU, ale týka sa pamäte zariadenia.

---

24. Kopírovanie dát: cudaMemcpy

Na kopírovanie dát medzi host a device pamäťou sa používa:

```
cudaMemcpy(destination, source, number_of_bytes, transfer_type);
```

Funkcia má štyri hlavné parametre:

- ukazovateľ na cieľ,
- ukazovateľ na zdroj,
- počet kopírovaných bajtov,
- typ prenosu.

Typy prenosu:

```
cudaMemcpyHostToHost
cudaMemcpyHostToDevice
```

cudaMemcpyDeviceToHost
cudaMemcpyDeviceToDevice

Najčastejšie používame:

```
cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);
```

na kopírovanie vstupov z CPU do GPU a:

```
cudaMemcpy(out, d_out, sizeof(float) * N, cudaMemcpyDeviceToHost);
```

na kopírovanie výsledkov z GPU späť do CPU.

---

25. Host pointer vs. device pointer

Pri CUDA treba jasne rozlišovať ukazovatele na host pamäť a ukazovatele na device pamäť.

Príklad:

```
float *a, *b, *out;
float *d_a, *d_b, *d_out;
```

- `a`, `b`, `out` sú host polia v pamäti CPU,
- `d_a`, `d_b`, `d_out` sú device polia v pamäti GPU.

CPU nemôže bežne priamo pracovať s device pointerom ako s obyčajným poľom. GPU kernel zasa pracuje s device pointermi. Preto sú potrebné explicitné prenosy cez `cudaMemcpy`.

---

26. Sekvenčná C verzia sčítania vektorov

Prednáška ukazuje najprv obyčajnú C verziu sčítania vektorov. Funkcia vyzerá približne takto:

```
void vector_add(float *out, float *a, float *b, int n) {
   for (int i = 0; i < n; i++) {
      out[i] = a[i] + b[i];
   }
}
```

Táto verzia beží na CPU. Má jeden cyklus, v ktorom sa postupne prechádza celé pole. Ak je `N = 10000000`, CPU vykoná 10 miliónov iterácií sekvenčne alebo podľa vlastných CPU optimalizácií.

Tento príklad je vhodný, pretože každá iterácia cyklu je nezávislá. Výpočet `out[i]` nezávisí od `out[i-1]`. Preto sa výborne hodí na CUDA paralelizáciu.

---

27. CUDA verzia sčítania vektorov

CUDA verzia presunie slučku z CPU na GPU. Namiesto toho, aby jedno CPU vlákno prechádzalo všetky prvky, spustí sa veľa GPU vlákien a každé spracuje jeden index.

Kernel:

```
__global__ void vector_add(float *out, float *a, float *b, int n) {
   int tid = blockIdx.x * blockDim.x + threadIdx.x;

   if (tid < n) {
      out[tid] = a[tid] + b[tid];
   }
}
```

Rozdiel oproti CPU verzii:

- CPU verzia má explicitný for cyklus,
- CUDA verzia nemá cyklus cez všetky prvky,
- každý index spracuje samostatné GPU vlákno,
- globálny index vlákna sa vypočíta pomocou blockIdx, blockDim, threadIdx.

---

28. Kompletný postup CUDA sčítania vektorov

Typický CUDA program pre sčítanie vektorov obsahuje tieto kroky:

1. Alokácia host pamäte

```
a   = (float*)malloc(sizeof(float) * N);
b   = (float*)malloc(sizeof(float) * N);
out = (float*)malloc(sizeof(float) * N);
```

Tieto polia existujú v pamäti CPU.

2. Inicializácia host polí

```
for (int i = 0; i < N; i++) {
   a[i] = 1.0f;
   b[i] = 2.0f;
}
```

CPU pripraví vstupné dáta.

3. Alokácia device pamäte

```
cudaMalloc((void**)&d_a, sizeof(float) * N);
cudaMalloc((void**)&d_b, sizeof(float) * N);
cudaMalloc((void**)&d_out, sizeof(float) * N);
```

GPU dostane vlastné polia v global memory.

4. Kopírovanie vstupov na GPU

```
cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, sizeof(float) * N, cudaMemcpyHostToDevice);
```

5. Spustenie kernelu

```
int block_size = 256;
int grid_size = (N + block_size - 1) / block_size;

vector_add<<<grid_size, block_size>>>(d_out, d_a, d_b, N);
```

6. Kopírovanie výsledkov späť na CPU

```
cudaMemcpy(out, d_out, sizeof(float) * N, cudaMemcpyDeviceToHost);
```

7. Overenie výsledku

```
for (int i = 0; i < N; i++) {
   assert(fabs(out[i] - a[i] - b[i]) < MAX_ERR);
}
```

8. Uvoľnenie pamäte

```
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_out);

free(a);
free(b);
free(out);
```

Toto je základný vzor CUDA programu: **alokuj → skopíruj → spusti kernel → skopíruj späť → uvoľni**.

---

29. Čo je na CUDA programe najdôležitejšie pochopiť

Pri CUDA programovaní nejde len o to, že prepíšeme for cyklus na kernel. Treba pochopiť celý model presunu výpočtu:

- CPU má svoje dáta v host pamäti.
- GPU má svoje dáta v device global memory.

- Pred výpočtom sa dáta musia dostať na GPU.
- Kernel sa spustí s množstvom vlákien.
- Každé vlákno spracuje svoju časť dát.
- Výsledky sa musia dostať späť na CPU, ak ich CPU potrebuje.

Najväčší rozdiel oproti OpenMP alebo Pthreads je v tom, že CUDA typicky pracuje s oddelenou pamäťou CPU a GPU. Preto sú kopírovania dát významnou časťou návrhu programu.

---

30. Prečo nestačí presunúť hocijaký program na GPU

GPU je výkonné hlavne pri dátovo paralelných výpočtoch. Ak program obsahuje veľa nezávislých operácií nad veľkým poľom dát, CUDA môže byť veľmi vhodná.

CUDA sa hodí napríklad na:

- sčítanie vektorov,
- násobenie matíc,
- spracovanie obrazu,
- filtre nad pixelmi,
- simulácie na mriežkach,
- neurónové siete,
- veľké numerické výpočty.

CUDA sa menej hodí, ak:

- program má málo dát,
- výpočet je silno sekvenčný,
- každá operácia závisí od predchádzajúcej,
- je veľa nepravidelného vetvenia,
- je veľa komunikácie medzi vzdialenými časťami dát,
- prenos dát CPU ↔ GPU je väčší problém než samotný výpočet.

---

31. Význam GPU knižníc

Prednáška upozorňuje aj na GPU-akcelerované knižnice. Ich hlavná výhoda je, že programátor nemusí vždy písať vlastný kernel. Namiesto toho môže použiť hotovú optimalizovanú funkciu.

NVIDIA GPU knižnice poskytujú vysoko optimalizované operácie a často môžu nahradiť CPU knižnice ako MKL, IPP alebo FFTW s malými zmenami v kóde. Výhodou je, že tieto knižnice sú už optimalizované pre GPU architektúru a často sa vedia škálovať aj cez viac GPU.

To je prakticky veľmi dôležité: ak existuje dobrá GPU knižnica pre daný problém, často je lepšie ju použiť než ručne písať vlastný kernel.

---

32. Lineárna algebra a matematické knižnice

cuBLAS

cuBLAS je GPU-akcelerovaná verzia BLAS knižnice. BLAS znamená Basic Linear Algebra Subroutines a obsahuje základné operácie lineárnej algebry, napríklad vektorové a maticové operácie.

Použitie:

- násobenie matíc,
- skalárne súčiny,
- operácie s vektormi,
- základné stavebné bloky vedeckých výpočtov a strojového učenia.

CUDA Math Library

CUDA Math Library poskytuje matematické funkcie pre GPU. Prednáška uvádza podporu štandardných C99 matematických funkcií pre `float` aj `double`.

cuSPARSE

cuSPARSE je BLAS-like knižnica pre riedke matice. Riedka matica je taká, ktorá obsahuje veľa nulových prvkov. Pri takýchto maticiach sa neoplatí ukladať ani spracúvať všetky prvky ako v hustej matici.

cuRAND

cuRAND je knižnica na generovanie náhodných čísel na GPU. Hodí sa napríklad pri Monte Carlo simuláciách alebo stochastických algoritmoch.

cuSolver

cuSolver poskytuje riešiče pre husté aj riedke lineárne systémy. Používa sa napríklad v počítačovom videní, CFD, výpočtovej chémii a lineárnej optimalizácii.

AmgX

AmgX je knižnica pre GPU-akcelerované lineárne riešiče, najmä pre simulácie a implicitné neštruktúrované metódy.

---

33. Knižnice pre signály, obraz a video

cuFFT

cuFFT je GPU-akcelerovaná knižnica pre Fast Fourier Transform. FFT sa používa v spracovaní signálov, obrazu, fyzike, numerických metódach a mnohých ďalších oblastiach.

NVIDIA Performance Primitives

Táto knižnica poskytuje GPU-akcelerované funkcie pre spracovanie obrazu a signálu. Hodí sa na operácie ako filtre, transformácie, spracovanie pixelov alebo základné signálové operácie.

NVIDIA Codec SDK

NVIDIA Codec SDK poskytuje nástroje a API pre hardvérovo akcelerované kódovanie a dekódovanie videa. To je dôležité pri video aplikáciách, streamovaní, spracovaní videa a video analytike.

---

34. Knižnice pre paralelné algoritmy

NCCL

NCCL je Collective Communications Library. Používa sa na škálovanie aplikácií cez viac GPU a viac uzlov. Podporuje kolektívne komunikácie, ktoré sú dôležité napríklad pri distribuovanom trénovaní neurónových sietí.

nvGRAPH

nvGRAPH je GPU-akcelerovaná knižnica pre grafové analýzy. Grafové algoritmy sa používajú napríklad pri sieťach, sociálnych grafoch, odporúčacích systémoch alebo analýze vzťahov medzi objektmi.

Thrust

Thrust je knižnica paralelných algoritmov a dátových štruktúr. Dá sa chápať ako CUDA-podobná paralelná verzia niektorých vysokoúrovňových algoritmických nástrojov. Hodí sa, keď nechceme písať nízkoúrovňový kernel pre každú operáciu.

---

35. Knižnice pre deep learning

cuDNN

cuDNN je GPU-akcelerovaná knižnica primitív pre hlboké neurónové siete. Obsahuje optimalizované operácie používané v deep learningu, napríklad konvolúcie a ďalšie základné bloky neurónových sietí.

TensorRT

TensorRT je knižnica na optimalizovanú inferenciu neurónových sietí. Inference znamená používanie už natrénovaného modelu na výpočet výsledkov. TensorRT sa hodí tam, kde chceme deep learning model spúšťať rýchlo a efektívne.

DeepStream SDK

DeepStream SDK je knižnica pre GPU-akcelerovanú video inferenciu. Používa sa najmä pri aplikáciách, kde sa nad videom vykonáva analýza pomocou neurónových sietí.

---

36. Kedy písať vlastný CUDA kernel a kedy použiť knižnicu

Vlastný kernel sa oplatí písať vtedy, keď:

- problém je špecifický,
- neexistuje vhodná hotová knižnica,
- potrebujeme vlastnú dátovú transformáciu,
- chceme optimalizovať konkrétnu časť algoritmu.

Knižnicu je lepšie použiť vtedy, keď:

- riešime štandardnú operáciu,
- ide o lineárnu algebru, FFT, deep learning alebo grafové operácie,
- knižnica je výrazne optimalizovaná,
- nechceme riešiť nízkoúrovňové optimalizácie CUDA.

Praktická zásada: **nepíš vlastný kernel na operáciu, ktorú už dobre rieši optimalizovaná GPU knižnica, pokiaľ na to nemáš konkrétny dôvod.**

---

37. Najdôležitejšie CUDA pojmy na zapamätanie

CUDA je architektúra a programovací model od NVIDIA pre výpočty na GPU.

Host je CPU časť programu.

Device je GPU časť programu.

Kernel je funkcia spustená na GPU veľkým počtom vlákien.

Grid je množina všetkých blokov vytvorených pri jednom spustení kernelu.

Block je skupina vlákien, ktoré môžu spolupracovať cez shared memory a synchronizáciu.

Thread je jedno CUDA vlákno vykonávajúce kernel.

threadIdx označuje pozíciu vlákna v bloku.

blockIdx označuje pozíciu bloku v gride.

blockDim hovorí, koľko vlákien je v bloku.

tid = blockIdx.x * blockDim.x + threadIdx.x je typický vzorec na výpočet globálneho indexu vlákna.

__global__ označuje kernel volaný z CPU a vykonávaný na GPU.

cudaMalloc alokuje pamäť v device global memory.

cudaFree uvoľní device pamäť.

cudaMemcpy kopíruje dáta medzi host a device pamäťou.

Global memory je veľká pamäť GPU dostupná všetkým vláknam, ale s vyššou latenciou.

Shared memory je rýchla pamäť zdieľaná vláknami v jednom bloku.

Registers sú najrýchlejšia súkromná pamäť vlákna.

Constant memory je čítacia pamäť vhodná pre konštanty.

Texture memory je špeciálna čítacia pamäť s optimalizáciami pre určité prístupy.

GPU knižnice sú hotové optimalizované funkcie pre časté výpočtové oblasti, napríklad cuBLAS, cuFFT, cuDNN alebo TensorRT.

---

38. Hlavná myšlienka celej prednášky

Prednáška 11.b ukazuje, ako sa GPU programuje pomocou CUDA. Kým prednáška 11.a vysvetľovala GPU ako architektúru, tu je dôraz na programovací model: CPU ako host pripravuje dáta a spúšťa výpočty, GPU ako device vykonáva masívne paralelný kernel pomocou veľkého množstva vlákien.

Najdôležitejšie je pochopiť hierarchiu:

kernel → grid → blocks → threads

a pamäťový model:

registre / local memory → pre jedno vlákno
shared memory → pre jeden blok
global memory → pre celý grid/GPU
constant a texture memory → čítacie špecializované pamäte

CUDA program zvyčajne funguje podľa vzoru:

alokuj host dáta
alokuj device dáta
skopíruj vstupy na GPU
spusti kernel
skopíruj výsledky späť
uvoľni pamäť

CUDA je výkonná vtedy, keď problém obsahuje veľa dátového paralelizmu a dá sa rozdeliť na veľké množstvo nezávislých vlákien. Pri bežných operáciách sa však často oplatí použiť hotové GPU knižnice, pretože sú už optimalizované a šetria veľa práce.

Prednáška 12-a: Paradigmy MIMD, DataFlow a distribuované systémy

1. Čo znamená MIMD

MIMD znamená **Multiple Instruction, Multiple Data**. Ide o triedu paralelných architektúr, v ktorej viacero procesných elementov môže vykonávať **rôzne inštrukčné prúdy** nad **rôznymi dátovými prúdmi**.

To znamená, že každý procesný element môže robiť niečo iné. Jeden procesor môže spracúvať jednu časť programu, druhý procesor inú časť programu a zároveň môžu pracovať nad odlišnými dátami.

MIMD architektúra je preto všeobecnejšia a flexibilnejšia ako SIMD. Pri SIMD všetky výpočtové jednotky vykonávajú tú istú inštrukciu nad rôznymi dátami. Pri MIMD môže mať každý procesor vlastný riadiaci tok.

Základná predstava MIMD:

- systém obsahuje viac nezávislých procesných elementov,
- každý procesný element môže mať vlastnú kópiu programu,
- každý procesný element môže pracovať s iným dátovým prúdom,
- procesné elementy môžu spolupracovať cez zdieľanú alebo distribuovanú pamäť.

---

2. MIMD multiprocesory podľa prístupu k pamäti

Pri MIMD architektúrach je veľmi dôležité, ako procesory pristupujú k pamäti. Podľa toho rozlišujeme najmä dve veľké skupiny:

MIMD so zdieľanou pamäťou

V tomto modeli majú procesory prístup k spoločnému adresnému priestoru. To znamená, že procesory môžu komunikovať tak, že čítajú a zapisujú spoločné premenné v pamäti.

Sem patria najmä:

- **UMA**,
- **NUMA**,
- **COMA**.

MIMD s distribuovanou pamäťou

V tomto modeli má každý výpočtový uzol vlastnú lokálnu pamäť. Ak chce jeden uzol komunikovať s druhým, musí mu poslať správu cez sieť.

Sem patria najmä:

- **multipočítače**,
- klastre,
- distribuované výpočtové systémy.

Rozdiel je veľmi dôležitý:

- pri zdieľanej pamäti sa komunikuje cez spoločné dáta,
- pri distribuovanej pamäti sa komunikuje explicitne cez správy.

---

3. UMA, NUMA a COMA

UMA – Uniform Memory Access

V UMA systéme majú všetky procesory rovnaký prístupový čas ku každej časti pamäte. Z pohľadu procesora teda nezáleží na tom, ku ktorej pamäťovej adrese pristupuje – prístup je približne rovnako rýchly.

Tento model je jednoduchý na programovanie, ale horšie sa škáluje na veľký počet procesorov, pretože všetci súťažia o tú istú pamäťovú infraštruktúru.

NUMA – Non-Uniform Memory Access

V NUMA systéme je pamäť fyzicky rozdelená medzi uzly. Každý procesor má svoju „bližšiu“ lokálnu pamäť, ku ktorej pristupuje rýchlejšie. K vzdialenej pamäti iného uzla sa síce dostať môže, ale prístup je pomalší.

To znamená, že program stále môže pracovať so zdieľaným adresným priestorom, ale výkon závisí od toho, kde sú dáta fyzicky umiestnené.

Zjednodušene:

- lokálna pamäť = rýchlejšia,
- vzdialená pamäť = pomalšia.

Preto je pri NUMA dôležité umiestňovať dáta čo najbližšie k procesoru, ktorý ich používa.

COMA – Cache-Only Memory Architecture

COMA je špeciálny model, kde sa pamäť správa ako veľká distribuovaná cache. Dáta nie sú pevne priradené k jednému pamäťovému miestu tak ako pri bežnom NUMA, ale môžu sa presúvať podľa toho, kde sú potrebné.

Myšlienka je znížiť cenu vzdialených prístupov tým, že sa dáta dynamicky presúvajú bližšie k procesorom, ktoré ich používajú.

---

4. Paradigmy MIMD

Prednáška neberie MIMD iba ako jednu architektúru, ale ako základ, na ktorom môžu vzniknúť rôzne paradigmy a kombinované prístupy.

Medzi paradigmy MIMD patria:

- špecializované architektúry,
- hybridné architektúry,
- MSIMD,
- SPMD,
- MIMD/SIMD,
- počítače DataFlow,
- redukčné počítače.

Dôležité je pochopiť, že MIMD je všeobecný rámec. Konkrétne architektúry môžu kombinovať viacero princípov. Napríklad systém môže byť navonok MIMD, ale vnútri niektorých výpočtových jednotiek môže používať SIMD alebo vektorové spracovanie.

---

5. SPMD ako praktická forma MIMD

SPMD znamená **Single Program, Multiple Data**. Na prvý pohľad pripomína SIMD, ale nie je to to isté.

V SPMD všetky procesory spúšťajú ten istý program, ale každý procesor môže vykonávať inú časť programu podľa svojho identifikátora alebo podľa dát, ktoré má pridelené.

Príklad:

```
if (my_rank == 0) {
   /* práca pre proces 0 */
} else {
   /* práca pre ostatné procesy */
}
```

Všetky procesy majú rovnaký program, ale správanie môže byť rôzne.

Rozdiel medzi SIMD a SPMD:

- **SIMD**: všetky jednotky vykonávajú naraz tú istú inštrukciu,
- **SPMD**: všetky procesory majú rovnaký program, ale môžu ísť rôznymi vetvami a vykonávať rôzne inštrukcie.

SPMD je veľmi dôležitý model v paralelnom programovaní, najmä pri MPI programoch.

---

6. MIMD/SIMD hybridné architektúry

Niektoré systémy kombinujú MIMD a SIMD princípy. Napríklad viacero nezávislých procesorov môže pracovať MIMD spôsobom, ale každý z nich má vnútri SIMD jednotku alebo vektorovú jednotku.

Takýto systém využíva výhody oboch modelov:

- MIMD poskytuje flexibilitu a nezávislé riadenie,
- SIMD poskytuje vysoký výkon pri dátovo paralelných operáciách.

Tento princíp je bežný aj v moderných procesoroch a GPU. CPU jadro môže vykonávať vlastný inštrukčný prúd, ale vnútri používa SIMD inštrukcie. GPU zasa kombinuje mnoho vláken s hardvérovým SIMD/SIMT vykonávaním.

---

7. Počítače riadené tokom dát – DataFlow

Jednou z dôležitých paradigiem MIMD sú **počítače riadené tokom dát**, teda **DataFlow počítače**.

V klasickom von Neumannovom modeli sa program vykonáva podľa poradia inštrukcií. Procesor má programové počítadlo a postupuje inštrukciu po inštrukcii.

V DataFlow modeli je to inak. Inštrukcia sa nevykoná preto, že je „ďalšia v poradí“, ale preto, že má dostupné všetky potrebné vstupné operandy.

Základná myšlienka:

Inštrukcia čaká, kým dostane všetky potrebné dáta. Keď ich má, môže sa vykonať.

Tento model sa označuje ako **data-driven**, teda riadený dátami.

---

8. Ako funguje výpočet v DataFlow modeli

V DataFlow programe inštrukcie pasívne čakajú na príchod operandov. Keď príde správna kombinácia operandov, inštrukcia sa stane vykonateľnou.

Interval, počas ktorého inštrukcia čaká na operandy, sa chápe ako jej výberová fáza. V tejto fáze sa sleduje, ktoré inštrukcie už majú dostupné dáta, a tým pádom môžu dostať pridelené výpočtové prostriedky.

To znamená, že v DataFlow systéme sa paralelizmus objavuje prirodzene:

- ak viac inštrukcií má dostupné operandy,
- a sú dostupné výpočtové prostriedky,
- môžu sa vykonať súčasne.

DataFlow model preto podporuje paralelizmus na úrovni inštrukcií, ale iným spôsobom než superskalárny procesor. Superskalár sa snaží dynamicky hľadať nezávislé inštrukcie v sekvenčnom prúde. DataFlow model vyjadruje závislosti priamo grafom.

---

9. DataFlow graf – DFG

Strojovou reprezentáciou DataFlow programu je **Data Flow Graph (DFG)**, teda graf toku dát.

DFG je orientovaný graf:

- uzly reprezentujú inštrukcie alebo operátory,
- orientované hrany reprezentujú tok výsledných dát,
- dáta sa po hranách prenášajú vo forme aktivačných značiek, teda tokenov.

Tokeny môžu byť:

- **DT – data tokens**, teda dátové tokeny,
- **CT – control tokens**, teda riadiace tokeny.

Uzol grafu sa môže aktivovať vtedy, keď sú na jeho vstupných hranách dostupné potrebné tokeny. Tým sa výpočet riadi dostupnosťou dát, nie poradím zápisu inštrukcií v programe.

---

10. Vlastnosti DataFlow grafu

DFG má niekoľko dôležitých vlastností.

Funkcionalita

Operácie v DFG sa chápu ako funkcie. To znamená, že výstup operácie závisí iba od jej vstupov a operácia nemá vedľajšie účinky.

Táto vlastnosť je veľmi dôležitá pre paralelizmus. Ak operácia nemení nejaký skrytý spoločný stav, môže sa bezpečne vykonať paralelne s inými nezávislými operáciami.

Asynchrónnosť

Uzly grafu sa aktivujú vtedy, keď sú dostupné ich vstupné tokeny. Nemusia čakať na globálne programové poradie.

To znamená, že rôzne časti grafu môžu bežať nezávisle a v rôznom čase, podľa toho, kedy sú pripravené ich dáta.

Komponovateľnosť

DFG možno spájať. Menšie grafy môžu tvoriť väčšie výpočtové štruktúry.

To je užitočné pri návrhu zložitejších programov, pretože výpočet sa dá skladať z menších častí.

---

11. Pravidlo vykonateľnosti a pravidlo aktivácie

Prednáška rozlišuje dve pravidlá:

Pravidlo vykonateľnosti (PV)

Inštrukcia je vykonateľná, ak sú všetky jej operandy prístupné.

To znamená, že uzol v grafe už má všetky potrebné vstupy.

Pravidlo aktivácie (PA)

Inštrukcia je aktivovaná, keď je vykonateľná a zároveň sú dostupné zdroje na jej aktiváciu.

Rozdiel je dôležitý:

- vykonateľná inštrukcia má pripravené dáta,
- aktivovaná inštrukcia už má aj pridelené výpočtové prostriedky.

Inými slovami: nie každá vykonateľná inštrukcia sa okamžite musí vykonať. Musí byť k dispozícii aj procesný element alebo iný výpočtový zdroj.

---

12. Charakteristické princípy DataFlow modelu

DataFlow model má dva hlavné princípy:

Asynchrónnosť

Operácie sa vykonajú vtedy, keď sú dostupné požadované operandy.

To znamená, že systém nepotrebuje centrálne programové počítadlo určujúce presné poradie všetkých inštrukcií. Poradie vzniká prirodzene z dátových závislostí.

Funkcionalita

Operácie sú funkcie bez vedľajších účinkov.

To znamená, že výsledok závisí iba od vstupov. Operácie neovplyvňujú skrytý globálny stav, a preto sa môžu bezpečnejšie vykonávať paralelne.

Toto je hlavný dôvod, prečo je DataFlow model vhodný na odhaľovanie paralelizmu. Ak sú operácie funkčné a závislosti sú vyjadrené hranami grafu, systém presne vidí, čo možno spustiť súčasne.

---

13. Klasifikácia DataFlow architektúr

Prednáška rozlišuje viac typov DF architektúr:

- statická DF architektúra,
- dynamická DF architektúra,
- čistá dynamická DF architektúra,
- DF architektúra s explicitným ukladaním dátových tokenov,
- DF architektúra s priamym spájaním operandov,

- hybridná DF architektúra.

Hybridné DF architektúry sa ďalej delia napríklad na:

- skoré hybridné DF architektúry,
- DF architektúry s vláknovým prístupom,
- hrubozrnné DF architektúry,
- RISC-ovské DF architektúry.

Dôležité je pochopiť, že DataFlow nie je jedna konkrétna architektúra. Je to výpočtový princíp, ktorý sa dá realizovať viacerými spôsobmi.

---

14. Statická DataFlow architektúra

V statickej DF architektúre je operátor, teda uzol grafu, vykonateľný vtedy, keď sú tokeny prítomné na všetkých vstupných hranách a zároveň na výstupnej hrane nie je prítomný žiadny token.

Toto obmedzenie súvisí s tým, že statický DataFlow model typicky neumožňuje mať viac aktívnych inštancií toho istého uzla naraz. Výstupná hrana musí byť voľná, aby sa výsledok mohol bezpečne uložiť alebo poslať ďalej.

Statická DF architektúra využíva najmä:

- **štrukturálny paralelizmus**,
- **paralelizmus zreťazenia**.

Štrukturálny paralelizmus znamená, že rôzne operátory grafu sa môžu vykonávať naraz. Paralelizmus zreťazenia znamená, že v rôznych častiach grafu sa v tom istom čase konzumujú rôzne tokeny.

Príklady statických DF architektúr:

- LAU,
- DDM1,
- HDFM.

Tieto príklady sú historické, ale dôležité je pochopiť princíp: statický DataFlow je jednoduchší, ale obmedzenejší pri slučkách a rekurzii.

---

15. Dynamická DataFlow architektúra

Dynamická DF architektúra rieši problém, že v reálnych programoch môžu slučky a rekurzia vytvárať viacero aktívnych inštancií rovnakého uzla.

V dynamickom DataFlow modeli je operátor vykonateľný vtedy, keď všetky vstupné hrany obsahujú tokeny, ktorých značky sú identické.

Tieto značky pomáhajú rozlíšiť, ktoré tokeny patria k tej istej aktivácii operácie. To je dôležité napríklad pri slučkách, kde ten istý uzol môže byť aktivovaný opakovane pre rôzne iterácie.

Dynamická DF architektúra využíva:

- slučkový paralelizmus,
- rekurzívny paralelizmus,
- spájanie tokenov, teda matching.

Matching

Matching znamená, že systém musí nájsť tokeny, ktoré patria k sebe. Ak má operácia dva vstupy, nestačí mať „nejaké“ dva tokeny. Musia mať rovnakú značku, aby patrili k tej istej aktivácii.

Preto dynamické DF architektúry často potrebujú asociatívne ukladanie údajov, aby vedeli efektívne vyhľadávať zodpovedajúce tokeny.

Príklady dynamických DF architektúr:

- Manchester Dataflow Machine,
- EXMAN,
- SIGMA-1.

Modifikované alebo hybridné prístupy:

- EM-4,
- TAM – Threaded Abstract Machine,
- DF KPI.

---

16. Statická vs. dynamická DataFlow architektúra

Najjednoduchšie porovnanie:

Statická DF architektúra

- jednoduchšia,
- operátor sa aktivuje pri dostupnosti vstupných tokenov a voľnom výstupe,
- horšie podporuje viac súčasných aktivácií rovnakého uzla,

- vhodná najmä pre jednoduchšie grafy.

Dynamická DF architektúra

- zložitejšia,
- tokeny majú značky,
- podporuje slučky a rekurziu,
- potrebuje matching tokenov,
- lepšie vyjadruje paralelizmus zložitejších programov.

Pre učenie je dôležitá veta:

Statická DF architektúra rieši jednoduchší tok dát, dynamická DF architektúra pridáva značkovanie tokenov, aby zvládla viac aktivácií, slučky a rekurziu.

---

17. Hybridné DataFlow architektúry

Čistý DataFlow model je zaujímavý teoreticky, ale v praxi môže byť zložitý na implementáciu a programovanie. Preto vznikali hybridné architektúry, ktoré kombinujú DataFlow princípy s prvkami klasického riadiaceho toku.

Hybridné DF architektúry sa snažia zachovať výhody DataFlow, najmä prirodzené odhaľovanie paralelizmu, ale zároveň znížiť réžiu spájania tokenov, plánovania a správy veľmi jemnozrnných operácií.

Typické hybridné myšlienky:

- zoskupovať viac operácií do väčších vlákien,
- používať DataFlow princíp na aktiváciu väčších blokov,
- vnútri bloku vykonávať inštrukcie sekvenčne alebo RISC spôsobom,
- znížiť náklady na tokeny a matching.

Toto súvisí aj s predchádzajúcimi prednáškami o granularite. Príliš jemná granularita síce odhalí veľa paralelizmu, ale môže mať veľkú réžiu. Hybridný DF prístup preto často používa hrubšiu granularitu.

---

18. Redukčné počítače

Prednáška medzi paradigmy MIMD zaraďuje aj **redukčné počítače**. Tie sú založené na inom výpočtovom princípe než klasický riadiaci tok.

Redukčný výpočet sa dá zjednodušene chápať ako postupné zjednodušovanie výrazu. Výpočet sa neriadi programovým počítadlom, ale tým, ktoré časti výrazu možno vyhodnotiť a nahradiť jednoduchším výsledkom.

Pre účely tejto prednášky je najdôležitejšie vedieť, že redukčné počítače patria medzi špecializované paradigmy paralelného spracovania a súvisia s funkcionálnym pohľadom na výpočet.

---

19. Distribuované systémy – základná definícia

Druhá veľká časť prednášky sa venuje **distribuovaným systémom**.

Distribuovaný systém možno chápať dvoma spôsobmi.

Definícia 1

Distribuovaný systém je sieťové prostredie vzájomne prepojených počítačových uzlov, ktoré umožňuje prenos informácií medzi nimi a zdieľanie ich hardvérových a softvérových prostriedkov.

Definícia 2

Distribuovaný systém je systém prepojených počítačových uzlov, ktorý sa navonok, z pohľadu používateľa, javí ako jeden výkonný počítačový uzol.

Druhá definícia je veľmi dôležitá. Hovorí, že používateľ nemusí vnímať, že systém je v skutočnosti zložený z viacerých počítačov. Ideálom je, aby sa distribuovaný systém správal ako jeden celok.

---

20. Základné vlastnosti distribuovaných systémov

Hardvérová nezávislosť uzlov

Uzly distribuovaného systému sú nezávislé počítače. Každý má vlastný procesor, vlastnú pamäť a vlastný operačný systém alebo lokálne prostredie.

To je rozdiel oproti klasickému multiprocesoru so zdieľanou pamäťou, kde procesory typicky patria do jedného tesnejšie integrovaného systému.

Komunikácia cez sieť

Uzly komunikujú pomocou komunikačného alebo sieťového rozhrania. Podľa rozsahu siete môže ísť napríklad o:

- LAN – lokálna sieť,
- MAN – metropolitná sieť,
- WAN – rozsiahla sieť.

Dojem jedného systému

Používateľ alebo softvér by mal komunikovať so systémom ako s jedným celkom. V ideálnom prípade používateľ nemusí vedieť, na ktorom konkrétnom uzle sa jeho výpočet vykonáva alebo kde sú uložené dáta.

---

21. Architektonické riešenia distribuovaných systémov

Prednáška uvádza viacero typických architektonických riešení:

- klaster,
- sieť pracovných staníc,
- grid,
- cloud,
- pervasívne systémy,
- peer-to-peer systémy.

Každé z týchto riešení má iný cieľ a inú organizáciu.

Klaster

Klaster je skupina výpočtových uzlov, ktoré sú zvyčajne relatívne homogénne. To znamená, že majú podobný hardvér, podobný operačný systém a sú spojené rýchlou lokálnou sieťou.

Klastre sa často používajú na vysokovýkonné výpočty.

Sieť pracovných staníc

Tu sa využívajú bežné pracovné stanice prepojené sieťou. Môžu slúžiť ako lacnejšia forma distribuovaného výpočtového prostredia.

Grid

Grid spája heterogénne uzly, často geograficky vzdialené. Môžu mať rôzny hardvér, rôzne operačné systémy a rôzne lokálne politiky správy.

Grid computing je vhodný na situácie, kde sa zdroje z viacerých organizácií spájajú do spoločnej výpočtovej infraštruktúry.

Cloud

Cloud poskytuje výpočtové zdroje ako služby. Používateľ si neobjednáva konkrétny fyzický stroj, ale službu, napríklad virtuálny server, platformu alebo softvérovú aplikáciu.

Pervasívne systémy

Pervasívne systémy sú systémy, kde sú výpočtové zdroje dostupné všade a často sú tvorené veľkým počtom jednoduchých, meniacich sa uzlov.

Peer-to-peer systémy

P2P systémy sú tvorené rovnocennými uzlami, ktoré môžu byť súčasne klientmi aj servermi.

---

22. Vývoj distribuovaných operačných systémov

Prednáška rozlišuje vývoj približne podľa období.

Distribuované OS v 90. rokoch

V 90. rokoch sa vyvíjali operačné systémy priamo pre distribuované výpočty. Podpora distribúcie, komunikácie a synchronizácie bola súčasťou jadra operačného systému.

Cieľom bolo, aby z pohľadu používateľa nebol rozdiel medzi lokálnou a distribuovanou aplikáciou.

Príklady:

- Amoeba,
- Sprite,
- Chorus.

Myšlienka bola veľmi ambiciózna: operačný systém mal zakryť distribúciu a vytvoriť dojem jedného systému.

Distribuované systémy po roku 2000

Neskôr sa viac presadil model, v ktorom každý uzol používa bežný lokálny OS, napríklad Windows alebo Linux, a distribuované vlastnosti poskytuje medzivrstva, teda **middleware**.

Príklady middleware technológií:

- CORBA,
- DCOM,
- Globe.

Tento prístup je praktickejší, pretože namiesto špeciálneho distribuovaného OS možno použiť bežné operačné systémy a distribúciu riešiť nad nimi.

---

23. Middleware

Middleware je softvérová vrstva medzi aplikáciou a lokálnymi operačnými systémami jednotlivých uzlov.

Jeho úloha je zjednodušiť vývoj distribuovaných aplikácií. Aplikácia nemusí priamo riešiť všetky detaily siete, komunikácie a vzdialených služieb, pretože tieto veci zabezpečuje middleware.

Middleware môže poskytovať napríklad:

- vzdialené volanie procedúr,
- pomenovanie služieb,
- komunikáciu medzi objektmi,
- správu distribuovaných zdrojov,
- podporu transakcií,
- bezpečnostné mechanizmy.

Zjednodušene:

Middleware robí z viacerých samostatných počítačov použiteľné distribuované prostredie.

---

24. Cloud computing

Cloud computing predstavuje model, kde sú výpočtové zdroje poskytované ako služba. Používateľ nemusí vlastniť hardvér ani spravovať celú infraštruktúru. Využíva zdroje poskytovateľa cez sieť.

Prednáška uvádza tri základné modely cloudových služieb:

SaaS – Software as a Service

Používateľ používa hotovú aplikáciu ako službu.

Príklad:

- Dropbox.

Používateľ nerieši operačný systém, platformu ani hardvér. Používa iba aplikáciu.

PaaS – Platform as a Service

Používateľ dostane platformu na vývoj a prevádzku aplikácií.

Príklad:

- Google App Engine.

Používateľ nerieši fyzický hardvér, ale môže nasadzovať svoje aplikácie do pripraveného prostredia.

IaaS – Infrastructure as a Service

Používateľ dostane infraštruktúru, napríklad virtuálne stroje, sieť a úložisko.

Príklad:

- Amazon Elastic Compute Cloud.

Používateľ má väčšiu kontrolu nad prostredím, napríklad nad operačným systémom vo virtuálnom stroji.

---

25. Virtualizácia v cloude

Cloud computing vyžaduje podporu virtualizácie. Virtualizácia umožňuje, aby na jednom fyzickom hardvéri bežalo viac virtuálnych strojov alebo izolovaných prostredí.

Výhody virtualizácie:

- lepšie využitie hardvéru,
- izolácia používateľov,
- flexibilné prideľovanie zdrojov,
- jednoduchšie presúvanie služieb,
- škálovanie podľa potreby.

V cloude používateľ často nevie a ani nepotrebuje vedieť, na ktorom fyzickom serveri jeho služba beží. Dôležité je, že služba je dostupná a má pridelené potrebné zdroje.

---

26. Vysokovýkonné distribuované počítanie

Prednáška rozlišuje tri hlavné prístupy:

- cluster computing,

- grid computing,
- cloud computing.

Cluster computing

Cluster computing je založený na homogénnom hardvéri a operačnom systéme, typicky v lokálnej sieti LAN.

Komunikácia v klastri môže prebiehať pomocou:

- zdieľanej pamäte,
- RDMA,
- distribuovaného message queuing.

RDMA (Remote Direct Memory Access) znamená, že dáta sa môžu prenášať medzi pamäťami uzlov bez priamej účasti operačného systému. To znižuje réžiu a latenciu komunikácie.

Klaster je vhodný na vysokovýkonné výpočty, pretože uzly sú blízko, sieť je rýchla a prostredie je relatívne kontrolované.

Grid computing

Grid computing je založený na heterogénnom hardvéri a operačných systémoch. Často pracuje cez WAN a spája zdroje z viacerých organizácií.

V grid prostredí treba počítať s tým, že:

- uzly môžu mať rôzny výkon,
- používajú rôzne systémy,
- sieťové oneskorenia sú väčšie,
- algoritmy a protokoly musia zvládnuť heterogenitu.

Grid je preto vhodný pre rozsiahle distribuované výpočty, ale jeho riadenie je zložitejšie než pri klastri.

Cloud computing pre výpočty

Cloud podporuje distribuované výpočty cez služby. Používateľ môže využiť napríklad:

- virtuálne stroje,
- kontajnery,
- high-throughput computing,
- online storage,
- prenos veľkých dát,
- zdieľanie datasetov.

Cloud je flexibilný, pretože zdroje možno dynamicky prideľovať podľa potreby.

---

27. Cluster vs. Grid vs. Cloud

Pre zapamätanie:

Cluster

- homogénne uzly,
- väčšinou jedna organizácia,
- LAN,
- rýchla komunikácia,
- vhodný pre HPC.

Grid

- heterogénne uzly,
- viac organizácií,
- často WAN,
- zložitejšia správa,
- vhodný na spájanie distribuovaných zdrojov.

Cloud

- zdroje ako služba,
- virtualizácia,
- elastické prideľovanie výkonu,
- orientácia na služby,
- používateľ často nerieši fyzickú infraštruktúru.

Jednoducho:

Cluster je tesne prepojený výpočtový systém, grid je federácia rôznych zdrojov a cloud je službový model poskytovania výpočtovej infraštruktúry.

---

28. Distribuované informačné systémy

Distribuované systémy sa nepoužívajú iba na vedecké výpočty. Veľmi dôležitou oblasťou sú aj distribuované informačné systémy.

Patrí sem:

- distribuované spracovanie dát,
- data-intenzívne výpočty,
- Big Data,

- Hadoop,
- dolovanie dát,
- distribuované spracovanie transakcií,
- distribuované a vnorené transakcie.

Tieto systémy riešia najmä spoľahlivé a efektívne spracovanie veľkého množstva dát, často uložených na viacerých uzloch.

---

29. ACID vlastnosti transakcií

Pri distribuovaných transakciách sú dôležité vlastnosti **ACID**.

A – Atomicity / atomičnosť

Transakcia je nedeliteľná. Buď sa vykoná celá, alebo sa nevykoná nič.

Príklad: pri bankovom prevode sa nesmie stať, že peniaze sa odpíšu z jedného účtu, ale nepripíšu sa na druhý.

C – Consistency / konzistentnosť

Transakcia transformuje databázu z jedného konzistentného stavu do druhého konzistentného stavu.

To znamená, že po dokončení transakcie musia stále platiť pravidlá systému.

I – Isolation / izolácia

Transakcie musia byť navzájom nezávislé. Čiastkové efekty jednej transakcie nemajú byť viditeľné iným transakciám, kým transakcia nie je dokončená.

D – Durability / trvanlivosť

Efekty potvrdenej transakcie musia zostať uložené v databáze. Aj keď nastane porucha, potvrdené zmeny sa nemajú stratiť.

ACID je dôležité najmä preto, že v distribuovanom prostredí je oveľa náročnejšie zabezpečiť spoľahlivosť než v jednom lokálnom systéme.

---

30. Pervasívne systémy

Pervasive computing alebo **ubiquitous computing** znamená výpočtové prostredie, kde sú výpočtové zdroje dostupné vždy a všade.

Typické vlastnosti:

- veľký počet jednoduchých uzlov,
- meniaca sa organizácia systému,
- rôzne typy zariadení,
- mobilita,
- senzory a mikrouzly,
- prepojenie s fyzickým svetom.

Príklady zariadení:

- PC,
- notebook,
- tablet,
- mobil,
- mikrokontroléry,
- snímače,
- IoT zariadenia,
- komponenty inteligentnej domácnosti.

Prednáška spomína aj tri základné formy podľa veľkosti interakčných zariadení:

- **Tabs** – nositeľné zariadenia, približne centimetrová mierka,
- **Pads** – handheld zariadenia, približne decimetrová mierka,
- **Boards** – väčšie zariadenia pre plnohodnotnú interakciu, približne metrová mierka.

Dôležitá myšlienka: v pervazívnych systémoch výpočet prestáva byť viazaný na jeden počítač. Je rozptýlený do prostredia.

---

31. Peer-to-peer systémy

Peer-to-peer (P2P) systém je distribuovaný systém, v ktorom sú uzly rovnocenné. Každý uzol môže vystupovať zároveň ako klient aj ako server.

V klasickom client-server modeli klient žiada službu od servera. V P2P modeli si uzly komunikujú priamo medzi sebou.

Typické vlastnosti P2P:

- neexistuje centrálny server alebo nie je nevyhnutný,
- uzly sú rovnocenné,
- každý uzol môže poskytovať aj využívať služby,
- komunikácia prebieha priamo medzi peer uzlami,
- každý uzol si môže uchovávať informácie o ostatných uzloch.

Nevýhody:

- môže sa prenášať viac dát než pri centralizovanom modeli,
- správa systému môže byť zložitejšia,
- vyhľadávanie a konzistencia dát sú náročnejšie.

Výhoda:

- systém nepotrebuje centrálny server,
- lepšie znáša výpadky niektorých uzlov,
- výkon a kapacita môžu rásť s počtom uzlov.

Príklady:

- DHT,
- zdieľanie súborov,
- P2P computing,
- Bitcoin,
- BitTorrent,
- Napster,
- Skype.

---

32. MPI – Message Passing Interface

Pri distribuovanej pamäti je dôležité programovanie pomocou posielania správ. Najznámejší štandard je **MPI – Message Passing Interface**.

MPI je štandard pre podporu medziprocesorovej komunikácie. Umožňuje komunikáciu medzi stovkami až tisíckami paralelne pracujúcich procesorov.

Používa sa najmä v:

- sieťach pracovných staníc,
- klastroch,
- superpočítačoch,
- distribuovaných pamäťových systémoch.

MPI definuje knižnicu podprogramov, ktoré poskytujú funkcie na:

- posielanie správ,
- prijímanie správ,
- kolektívnu komunikáciu,
- dynamické vytváranie procesov,
- tvorbu komunikačných topológií.

Implementácie MPI:

- MPICH,
- Open MPI.

Podporované jazyky:

- C/C++,
- Fortran,
- Python,
- R,
- Java.

Dôležité je pochopiť rozdiel oproti OpenMP a Pthreads:

- OpenMP a Pthreads predpokladajú zdieľanú pamäť,
- MPI predpokladá komunikáciu pomocou správ,
- každý proces má vlastný adresný priestor,
- dáta sa medzi procesmi prenášajú explicitne.

---

33. Základné problémy distribuovaných systémov

Prednáška uvádza aj širšie oblasti, ktoré sa riešia v distribuovaných systémoch.

Procesy a vlákna

Distribuované systémy musia riešiť, kde procesy bežia, ako sa vytvárajú, ako sa presúvajú a ako spolu komunikujú.

Komunikácia

Dôležité formy komunikácie:

- vzdialené volanie procedúr,
- posielanie správ,
- komunikácia pomocou prúdov,
- skupinová komunikácia na aplikačnej vrstve.

Menná služba

Systém musí vedieť pomenovať entity. Mená môžu byť jednoduché alebo štruktúrované, prípadne rozšírené o atribúty.

Synchronizácia

Distribuované systémy musia riešiť:

- synchronizáciu času,
- logické hodiny,
- vzájomné vylúčenie,
- voľbu vedúceho procesu,
- použitie GPS pri časovej synchronizácii.

Konzistencia a replikácia

Ak sú dáta replikované na viacerých uzloch, systém musí riešiť, ako zabezpečiť, aby kópie dávali zmysel.

Modely môžu byť:

- zamerané na údaje,
- zamerané na klienta.

Návrhové vzory

Distribuované systémy používajú rôzne návrhové vzory pre:

- základnú architektúru systému,
- rozdelenie rozhraní,
- rozdelenie na komponenty,
- správu aplikácie,
- súbežný prístup ku zdrojom,
- synchronizáciu,
- interakciu medzi objektmi,
- adaptáciu a rozšírenie,
- správu zdrojov.

---

34. Ako si zapamätať celú prednášku

Prednáška má dve veľké línie:

1. MIMD paradigmy

Tu sa rieši, ako môžu viaceré nezávislé procesory vykonávať rôzne inštrukčné prúdy nad rôznymi dátami.

Najdôležitejšie pojmy:

- MIMD,
- zdieľaná vs. distribuovaná pamäť,

- UMA, NUMA, COMA,
- SPMD,
- MIMD/SIMD,
- DataFlow,
- DFG,
- statická DF architektúra,
- dynamická DF architektúra,
- matching tokenov,
- hybridné DF architektúry.

2. Distribuované systémy

Tu sa rieši, ako spolupracujú samostatné počítačové uzly cez sieť.

Najdôležitejšie pojmy:

- distribuovaný systém,
- klaster,
- grid,
- cloud,
- middleware,
- SaaS/PaaS/IaaS,
- RDMA,
- Big Data,
- ACID,
- pervazívne systémy,
- P2P,
- MPI,
- konzistencia a replikácia.

---

35. Najdôležitejšie rozdiely

MIMD vs. SIMD

- SIMD: jedna inštrukcia, viac dát.
- MIMD: viac inštrukcií, viac dát.
- SIMD je vhodné na pravidelné dátovo paralelné úlohy.
- MIMD je flexibilnejšie a vhodné na všeobecnejší paralelizmus.

Zdieľaná pamäť vs. distribuovaná pamäť

- Zdieľaná pamäť: procesory komunikujú cez spoločné premenné.
- Distribuovaná pamäť: procesy komunikujú posielaním správ.
- Zdieľaná pamäť je často jednoduchšia na programovanie.
- Distribuovaná pamäť lepšie zodpovedá veľkým systémom a klastrom.

Control Flow vs. DataFlow

- Control Flow: vykonávanie riadi poradie inštrukcií.
- DataFlow: vykonávanie riadi dostupnosť operandov.
- Control Flow je klasický model procesorov.
- DataFlow prirodzene odhaľuje paralelizmus podľa závislostí.

Cluster vs. Grid vs. Cloud

- Cluster: homogénny, lokálny, výkonný.
- Grid: heterogénny, distribuovaný medzi organizáciami.
- Cloud: zdroje poskytované ako služba.

P2P vs. client-server

- Client-server: klient žiada službu od centrálneho servera.
- P2P: každý uzol môže byť klient aj server.
- P2P nepotrebuje centrálny server, ale má zložitejšiu komunikáciu.

---

36. Hlavná myšlienka celej prednášky

Prednáška 12.a ukazuje, že MIMD architektúry sú veľmi široká a flexibilná trieda paralelných systémov. Môžu byť založené na zdieľanej pamäti, distribuovanej pamäti alebo na hybridných prístupoch. Ich podstatou je, že viac procesných elementov môže vykonávať rôzne inštrukčné prúdy nad rôznymi dátami.

Dôležitou alternatívou ku klasickému riadiacemu toku je DataFlow model. V ňom sa inštrukcie nespúšťajú podľa poradia v programe, ale podľa dostupnosti operandov. Výpočet sa reprezentuje grafom toku dát, kde uzly sú operácie a hrany prenášajú tokeny. Statické DataFlow architektúry sú jednoduchšie, dynamické pridávajú značkovanie tokenov a lepšie podporujú slučky a rekurziu.

Druhá časť prednášky rozširuje pohľad na distribuované systémy. Tie sú tvorené viacerými prepojenými počítačovými uzlami, ktoré sa môžu používateľovi javiť ako jeden systém. Patria sem klastre, gridy, cloudy, pervazívne systémy a peer-to-peer siete. Pri distribuovaných systémoch sú kľúčové komunikácia, pomenovanie, synchronizácia, konzistencia, replikácia a programovanie pomocou správ, najmä cez MPI.

Ak si z tejto prednášky zapamätáš jednu vetu, tak túto:

MIMD systémy umožňujú všeobecný paralelizmus viacerých nezávislých výpočtových tokov, DataFlow ukazuje alternatívny spôsob riadenia výpočtu podľa dostupnosti dát a distribuované systémy rozširujú paralelizmus na spoluprácu viacerých samostatných počítačových uzlov cez sieť.

Prednáška 12-b: FPGA

1. Čo je FPGA

FPGA znamená **Field Programmable Gate Array**, po slovensky približne **programovateľné hradlové pole**.

FPGA je integrovaný obvod, ktorý umožňuje vytvoriť vlastný digitálny obvod. To znamená, že nejde o procesor, ktorý by iba vykonával program inštrukciu po inštrukcii. Pri FPGA sa skôr „poskladá“ hardvérová štruktúra, ktorá potom vykonáva požadovanú funkciu.

Dôležitá myšlienka:

- **mikroprocesor** vykonáva program,
- **FPGA** sa nakonfiguruje tak, aby sa správalo ako konkrétny digitálny obvod.

FPGA teda samo o sebe po výrobe nevykonáva žiadnu konkrétnu funkciu. Funkciu mu dá až konfigurácia, ktorá určí, ako sa majú správať jeho logické bloky a ako majú byť medzi sebou prepojené.

---

2. FPGA vs. mikrokontrolér

Mikrokontrolér už po výrobe obsahuje konkrétne integrované moduly, napríklad CPU jadro, časovače, AD prevodníky, komunikačné rozhrania a podobne. Programátor doň nahrá program, ktorý tieto moduly používa.

FPGA je iné. Nemá vopred danú jednu výpočtovú funkciu. Obsahuje množstvo programovateľných logických blokov a programovateľných prepojení. Pomocou konfigurácie si z nich vytvoríme požadovaný obvod.

Príklad rozdielu:

- pri mikrokontroléri napíšem program, ktorý povie procesoru, čo má robiť,
- pri FPGA opíšem hardvér, ktorý sa má vytvoriť.

Preto sa pri FPGA často hovorí, že „neprogramujeme algoritmus“ v klasickom zmysle, ale **opisujeme digitálny obvod**.

---

3. Funkcia FPGA a konfigurácia

Funkciu FPGA definuje **programovateľná logika**. Tá určuje:

- aké logické funkcie budú realizovať jednotlivé bloky,
- ako budú bloky medzi sebou prepojené,
- ako sa budú používať vstupno-výstupné piny,
- aké registre, pamäte alebo špecializované bloky budú zapojené.

Mnohé FPGA sú **volatilné zariadenia**. To znamená, že po vypnutí napájania stratia svoju konfiguráciu. Preto sa konfigurácia často uchováva v externej **FLASH pamäti**.

Po zapnutí napájania sa konfigurácia z FLASH pamäte nahrá do FPGA. Až potom začne FPGA vykonávať svoju funkciu. Z toho vyplýva, že FPGA nie je okamžite po zapnutí pripravené na prácu. Trvá určitý čas, typicky niekoľko milisekúnd až desiatky milisekúnd, kým sa nakonfiguruje a „nabehne“.

---

4. Základná štruktúra FPGA

FPGA sa skladá z troch hlavných častí:

1. Logické bloky

Logické bloky realizujú samotnú výpočtovú alebo riadiacu logiku. V nich sa vytvárajú logické funkcie, registre, jednoduché aritmetické obvody a podobne.

2. Programovateľná prepojovacia matica

Prepojovacia matica určuje, ako sú logické bloky medzi sebou prepojené. Je to veľmi dôležitá časť FPGA, pretože samotné logické bloky by nestačili. Aby vznikol funkčný obvod, treba ich správne pospájať.

3. Programovateľné I/O bloky

I/O bloky zabezpečujú komunikáciu FPGA s vonkajším svetom. Cez ne sa pripájajú vstupné a výstupné signály, napríklad tlačidlá, LED, pamäte, zbernice, sieťové rozhrania alebo obrazové výstupy.

Zjednodušená predstava:

I/O piny → prepojovacia sieť → logické bloky → prepojovacia sieť → I/O piny

FPGA teda nie je jeden procesor, ale veľké pole logických blokov, ktoré sa dajú ľubovoľne zapájať do rôznych digitálnych obvodov.

---

5. Prečo má FPGA veľa vývodov

FPGA často disponuje veľkým počtom vývodov, rádovo približne stovky až tisíce pinov. Dôvod je jednoduchý: FPGA má slúžiť ako univerzálny konfigurovateľný obvod, ktorý sa môže pripájať k rôznym externým zariadeniam.

Preto sa FPGA často vyrába v puzdrách typu **BGA (Ball Grid Array)**. Pri BGA sú kontakty umiestnené ako mriežka guľôčok na spodnej strane čipu. To umožňuje veľký počet pinov na malej ploche.

Nevýhoda je, že takéto puzdro je ťažšie použiteľné pre amatérske alebo ručné spájkovanie. FPGA dosky sa preto často používajú ako hotové vývojové platformy.

---

6. CLB / LAB – konfigurovateľné logické bloky

Základnou stavebnou jednotkou FPGA sú konfigurovateľné logické bloky.

Rôzni výrobcovia používajú rôzne názvy:

- **CLB – Configurable Logic Block** používa Xilinx,
- **LAB – Logic Array Block** používa Altera/Intel FPGA.

Takýto logický blok typicky obsahuje:

- **LUT (Look-Up Table)**,
- sčítačky alebo rýchle prenosové logiky,
- klopné obvody,
- multiplexery,
- lokálne prepojenia.

---

7. LUT – Look-Up Table

LUT je základný prvok, pomocou ktorého FPGA realizuje logické funkcie.

LUT si môžeme predstaviť ako malú pamäťovú tabuľku, ktorá pre každú kombináciu vstupov obsahuje požadovanú výstupnú hodnotu.

Napríklad ak má LUT 3 vstupy, existuje 8 možných kombinácií vstupov. Do tabuľky sa uloží, aký výstup má vzniknúť pre každú kombináciu. Týmto spôsobom vie LUT realizovať ľubovoľnú logickú funkciu s daným počtom vstupov.

Príklad:

- AND,
- OR,

- XOR,
- multiplexor,
- časť aritmetickej jednotky,
- vlastná kombinačná logika.

Dôležité je pochopiť, že LUT neráta funkciu ako procesor. LUT je nakonfigurovaná tabuľka pravdivostných hodnôt. Výstup je daný konfiguráciou.

---

8. Klopné obvody v FPGA

Okrem kombinačnej logiky FPGA obsahuje aj **klopné obvody**. Tie umožňujú vytvárať sekvenčné obvody, teda obvody so stavom.

Kombinačný obvod dáva výstup iba podľa aktuálnych vstupov. Sekvenčný obvod si pamätá aj predchádzajúci stav.

Klopné obvody sa používajú napríklad na:

- registre,
- čítače,
- stavové automaty,
- pipeline registre,
- synchronizáciu signálov.

Preto FPGA vie vytvoriť nielen jednoduché logické funkcie, ale aj zložité procesory, radiče, komunikačné jednotky alebo akcelerátory.

---

9. Programovateľná prepojovacia matica

Logické bloky musia byť medzi sebou pospájané. Na to slúžia **programovateľné prepojovacie bloky** a **switch matrix / switch boxes**.

Konfigurácia FPGA určuje, ktoré spojenia budú aktívne. Predstav si to ako veľkú sieť vodičov a prepínačov. Niektoré prepínače sa zapnú, iné zostanú vypnuté. Tak sa vytvorí konkrétna dátová cesta medzi logickými blokmi.

Táto prepojovacia sieť je veľmi dôležitá pre výkon. Aj keď logické bloky zvládajú požadovanú funkciu, zlé alebo príliš dlhé prepojenia môžu obmedziť maximálnu frekvenciu obvodu.

Preto sa pri implementácii rieši nielen logika, ale aj **rozmiestnenie a prepojenie**.

---

10. Prídavné bloky v moderných FPGA

Moderné FPGA neobsahujú iba LUT a klopné obvody. Často obsahujú aj špecializované bloky, ktoré urýchľujú bežné operácie.

Patria sem:

Embedded processors

Niektoré FPGA môžu obsahovať alebo podporovať procesorové jadrá, napríklad:

- **MicroBlaze**,
- **Nios**.

Tieto procesory môžu byť soft-core alebo hard-core. Soft-core znamená, že procesor je vytvorený z programovateľnej logiky FPGA.

DSP bloky

DSP bloky sú špecializované na rýchle aritmetické operácie, napríklad násobenie, akumuláciu alebo operácie používané pri spracovaní signálov.

Používajú sa pri:

- filtrovaní signálov,
- FFT,
- spracovaní obrazu,
- neurónových sieťach,
- numerických akcelerátoroch.

On-chip memory

FPGA môže obsahovať interné pamäťové bloky. Tie sú rýchlejšie než externá pamäť a používajú sa ako buffre, FIFO, malé RAM, cache-like štruktúry alebo úložisko medzivýsledkov.

Clock management

FPGA obsahuje bloky na správu hodín, napríklad:

- **PLL (Phase-Locked Loop)**,
- **DCM (Digital Clock Management)**.

Tie umožňujú generovať, deliť, násobiť alebo fázovo upravovať hodinové signály.

---

11. HDL – jazyk na opis hardvéru

Na opis digitálneho obvodu pre FPGA sa používa **HDL – Hardware Description Language**.

Najdôležitejšia myšlienka:

HDL kód sa nevykonáva ako klasický program. Z HDL opisu sa odvodí hardvérová konfigurácia.

Výnimkou je simulácia. Pri simulácii sa HDL kód „vykonáva“ v simulačnom prostredí, aby sme overili správanie obvodu. Ale pri implementácii na FPGA sa z HDL nevytvorí program pre procesor, ale hardvérová štruktúra.

To je zásadný rozdiel oproti jazykom ako C, Java alebo Python.

---

12. Paralelnosť v HDL

V klasickom programe sa príkazy často chápu sekvenčne: najprv sa vykoná prvý, potom druhý, potom tretí.

V HDL je situácia iná. Hardvér pracuje prirodzene paralelne. Ak v obvode existujú dve nezávislé časti, obe môžu fungovať súčasne.

Preto HDL používa koncepty ako:

- procesy,
- signály,
- súbežné priradenia,
- časovanie,
- hodinové hrany.

Pri FPGA treba myslieť hardvérovo: viac výrazov v HDL môže reprezentovať viac obvodov pracujúcich súčasne, nie postupnosť príkazov vykonaných jedným procesorom.

---

13. Spôsoby opisu digitálneho obvodu

HDL opis môže byť na rôznej úrovni abstrakcie.

Behaviorálny opis

Opisuje, ako sa má obvod správať. Nezameriava sa priamo na konkrétne hradlá, ale na požadovanú funkciu.

Príklad: „ak je vstup enable = 1, zvýš čítač“.

Štrukturálny opis

Opisuje, z akých komponentov sa obvod skladá a ako sú prepojené.

Príklad: „tento multiplexor pripoj na tento register, výstup registra na ALU“.

Tok dát

Opisuje, ako dáta pretekajú cez kombinačné vzťahy medzi signálmi.

Príklad: `sum = a xor b xor cin`.

V praxi sa tieto štýly často kombinujú.

---

14. Nízkoúrovňové HDL jazyky

Medzi najznámejšie HDL jazyky patria:

- **VHDL**,
- **Verilog**,
- **SystemVerilog**.

V prednáške sa uvádza, že VHDL je tradične rozšírenejší v Európe a Verilog v USA.

VHDL

VHDL je veľmi explicitný a formálny. Často sa používa v akademickom prostredí, pri návrhu spoľahlivých systémov a vo firmách, kde sa kladie dôraz na presnú špecifikáciu.

Verilog

Verilog má syntax bližšiu jazyku C. Mnohým programátorom sa zdá stručnejší a jednoduchší na zápis.

SystemVerilog

SystemVerilog rozširuje Verilog o ďalšie možnosti pre návrh aj verifikáciu hardvéru.

---

15. Príklad: full adder

Prednáška ukazuje príklad plnej sčítačky vo VHDL a Verilogu.

Plná sčítačka má vstupy:

- a,
- b,
- cin – prenos z nižšieho bitu.

Výstupy:

- s – výsledný bit súčtu,
- cout – prenos do vyššieho bitu.

Logika:

```
p = a xor b
g = a and b
s = p xor cin
cout = g or (p and cin)
```

Dôležité je pochopiť, že tento opis neznamená, že CPU vykoná tieto riadky jeden po druhom. Znamená to, že sa vytvorí kombinačný logický obvod, ktorý bude tieto vzťahy realizovať hardvérovo.

---

16. Vysokoúrovňové HDL / HLS prístupy

Okrem klasických HDL existujú aj vyššie úrovne opisu hardvéru, napríklad:

- **SystemC**,
- **Catapult C**,
- **Vivado HLS / Vivado HDL**,
- **Visual HDL**,
- **MyHDL**.

Tieto prístupy sa snažia umožniť návrh hardvéru pomocou jazykov podobných C, C++ alebo Pythonu.

Výhoda:

- rýchlejší návrh,
- vyššia abstrakcia,
- jednoduchšie prototypovanie.

Nevýhoda:

- návrhár stále musí rozumieť hardvéru,
- nie každý algoritmus sa dá efektívne previesť na FPGA,
- výsledná kvalita závisí od nástroja aj od štýlu zápisu.

Pri HLS je nebezpečné myslieť si, že hocijaký C program sa automaticky dobre premení na hardvér. Efektívny FPGA návrh stále vyžaduje pochopenie paralelizmu, pipeline, pamäte a dátových ciest.

---

17. Od návrhu k implementácii

Proces vytvorenia FPGA konfigurácie z HDL opisu má viac krokov. Prednáška zdôrazňuje tri hlavné:

1. syntéza,
2. mapovanie na cieľovú technológiu,
3. rozmiestnenie a prepojenie.

Tieto kroky sú dôležité, pretože ukazujú, že HDL opis nie je priamo konfigurácia FPGA. Medzi opisom a hotovým hardvérom je celý automatizovaný návrhový tok.

---

18. Krok 1 – syntéza

Syntéza analyzuje HDL opis a hľadá zodpovedajúce hardvérové štruktúry.

Výsledkom syntézy sú napríklad:

- kombinačné obvody,
- registre,
- čítače,
- multiplexery,
- stavové automaty,
- aritmetické jednotky.

Syntéza teda prekladá abstraktný HDL opis na logickú štruktúru.

Príklad:

Ak v HDL napíšeme podmienku:

```
if enable then
   counter <= counter + 1
```

syntéza môže vytvoriť register pre counter, sčítačku a riadiacu logiku podľa signálu enable.

---

19. Krok 2 – mapovanie na cieľovú technológiu

Po syntéze ešte nemáme konkrétne obsadené prvky FPGA. Máme len logickú štruktúru.

Mapovanie prispôsobí túto štruktúru konkrétnemu FPGA čipu. Mapper sa snaží nájsť najúspornejšie alebo najrýchlejšie ekvivalenty pomocou dostupných prvkov cieľového FPGA.

Rieši sa napríklad:

- či sa logika zmestí do počtu dostupných LUT,
- či je dosť klopných obvodov,
- či je dosť DSP blokov,
- či je dosť pamäťových blokov,
- či vyhovujú časové charakteristiky.

Inak povedané: mapovanie odpovedá na otázku, či je návrh na zvolenom FPGA realizovateľný a ako ho najlepšie priradiť na dostupné hardvérové prvky.

---

20. Krok 3 – rozmiestnenie a prepojenie

Po mapovaní treba rozhodnúť, kde presne budú jednotlivé komponenty na čipe umiestnené a ako budú prepojené.

Tento krok sa nazýva:

- **placement** – rozmiestnenie,
- **routing** – prepojenie.

Cieľom je nájsť vhodné usporiadanie v prepojovacej matici FPGA.

Táto fáza výrazne ovplyvňuje:

- maximálnu frekvenciu,
- oneskorenia signálov,
- spotrebu,
- úspešnosť splnenia časových požiadaviek.

Ak sú dva často komunikujúce bloky ďaleko od seba, signál medzi nimi môže mať veľké oneskorenie. Preto nástroj hľadá také rozmiestnenie, ktoré rešpektuje logiku aj časovanie.

---

21. Static Timing Analysis

Pri rozmiestnení a prepojení sa rieši najmä časovanie pomocou **static timing analysis**.

Tá overuje, či bude návrh fungovať na zvolenej frekvencii. Kontrolujú sa časové obmedzenia, napríklad:

- **setup time**,
- **hold time**,
- maximálne oneskorenie cesty,
- synchronizácia medzi registrami.

Setup time

Setup time znamená, že vstup registra musí byť stabilný určitý čas pred aktívnou hranou hodín.

Hold time

Hold time znamená, že vstup registra musí zostať stabilný určitý čas po aktívnej hrane hodín.

Ak návrh tieto podmienky nespĺňa, môže sa správať nespoľahlivo. Preto FPGA návrh nie je hotový len tým, že logicky funguje v simulácii. Musí fungovať aj časovo na konkrétnom čipe.

---

22. Simulácia

Pred implementáciou sa návrh simuluje. Simulácia overuje, či HDL opis robí to, čo od neho očakávame.

Simulácia sa používa na:

- overenie logickej správnosti,
- testovanie vstupných kombinácií,
- kontrolu časového priebehu signálov,
- hľadanie chýb pred nasadením na hardvér.

V simulácii môžeme sledovať hodnoty signálov v čase a vidieť, či sa obvod správa správne.

Dôležité:

- simulácia overuje správanie modelu,
- implementácia overuje, či sa model dá realizovať na konkrétnom FPGA,
- časová analýza overuje, či bude fungovať pri požadovanej frekvencii.

---

23. Prečo používať FPGA

FPGA má viac výhod, ktoré ho robia atraktívnym najmä pri návrhu špecializovaných paralelných architektúr a akcelerátorov.

Flexibilita

FPGA je flexibilnejšie než dedikovaný čip, pretože jeho konfiguráciu možno meniť. Ak sa návrh zmení, nemusí sa vyrábať nový čip. Stačí vygenerovať novú konfiguráciu.

Rýchle prototypovanie

FPGA umožňuje rýchlo otestovať hardvérový návrh v praxi. To je veľmi užitočné pred výrobou ASIC alebo pri výskume nových architektúr.

IP cores

FPGA ekosystém používa hotové komponenty nazývané **IP cores**. Môže ísť napríklad o procesorové jadro, pamäťový radič, komunikačné rozhranie alebo DSP blok. Návrhár tak nemusí všetko vytvárať od nuly.

Výkon

FPGA môže byť výrazne rýchlejšie než mikrokontrolér alebo mikroprocesor pri úlohách, ktoré sa dajú realizovať priamo hardvérovo a paralelne.

Preprogramovateľnosť

FPGA možno preprogramovať:

- vo fáze návrhu,
- vo fáze výroby,
- aj počas prevádzky.

Nižšie riziko než ASIC

Pri ASIC je výroba drahá a chyby sú veľmi nákladné. Pri FPGA možno návrh opravovať opakovanou konfiguráciou. Preto je vývoj menej riskantný.

---

24. FPGA vs. ASIC

ASIC

ASIC je špecializovaný čip navrhnutý pre konkrétnu funkciu. Má výborný výkon a výbornú efektivitu, ale po výrobe sa už nedá meniť.

Výhody ASIC:

- veľmi vysoký výkon,
- veľmi dobrá energetická efektivita,
- najlepšie riešenie pri veľkosériovej výrobe.

Nevýhody ASIC:

- vysoké počiatočné náklady,
- dlhý vývoj,
- vysoké riziko chyby,
- nulová flexibilita po výrobe.

FPGA

FPGA je pomalšie a menej efektívne než ideálne navrhnutý ASIC, ale je oveľa flexibilnejšie.

Výhody FPGA:

- preprogramovateľnosť,
- rýchle prototypovanie,
- nižšie vstupné náklady,
- vhodné pre menšie série a výskum,
- možnosť opráv a aktualizácií.

Jednoduché porovnanie:

ASIC je najlepší, keď presne vieme, čo chceme, vyrábame vo veľkom a nechceme nič meniť. FPGA je lepšie, keď potrebujeme flexibilitu, prototypovanie alebo možnosť neskoršej úpravy.

---

25. FPGA vs. CPU, GPU a DSP

Prednáška porovnáva FPGA s viacerými technológiami podľa výkonu, programovania, rýchlosti reakcie a flexibility.

Generic CPU

CPU je najjednoduchšie programovateľné a najflexibilnejšie, ale pri silne paralelných alebo špecializovaných úlohách nemusí mať najlepší výkon/cenu.

CPU je vhodné na:

- všeobecné programy,
- zložité riadenie,
- sekvenčné úlohy,
- operačné systémy,

- aplikácie s častými zmenami logiky.

GPU

GPU má veľmi dobrý výkon/cenu pri dátovo paralelných úlohách. Programovanie je náročnejšie než pri CPU, ale jednoduchšie než návrh FPGA. GPU je vhodné najmä na veľké množstvo rovnakých operácií nad veľkými dátami.

DSP

DSP je špecializovaný procesor pre digitálne spracovanie signálov. Má dobrý výkon pri signálových operáciách, ale stále vykonáva inštrukcie ako procesor.

FPGA

FPGA je medzi softvérovým procesorom a dedikovaným hardvérom. Vie realizovať paralelné dátové cesty priamo v hardvéri, ale stále zostáva preprogramovateľné.

Zapamätanie:

- **CPU** = najľahšie programovanie, najvyššia flexibilita,
- **GPU** = výborné pre masívny dátový paralelizmus,
- **DSP** = dobré pre signálové výpočty,
- **FPGA** = konfigurovateľný hardvér s vysokou paralelnosťou,
- **ASIC** = najvyšší výkon a efektivita, ale bez flexibility.

---

26. Prečo môže byť FPGA rýchle

FPGA môže byť rýchle nie preto, že by malo extrémne vysokú frekvenciu, ale preto, že vie vytvoriť **vlastnú paralelnú hardvérovú štruktúru**.

CPU typicky vykonáva inštrukcie postupne alebo s obmedzeným paralelizmom. GPU vykonáva veľa vlákien, ale podľa pevne danej architektúry. FPGA umožňuje navrhnúť vlastnú dátovú cestu presne pre danú úlohu.

Príklad:

Ak potrebujeme spracovávať prúd dát, môžeme na FPGA vytvoriť pipeline, kde každý stupeň robí inú časť výpočtu. Po naplnení pipeline môže v každom takte vypadnúť nový výsledok.

FPGA teda získava výkon cez:

- paralelizmus,
- pipeline,
- špecializované dátové cesty,

- lokálne pamäte,
- DSP bloky,
- minimalizáciu zbytočného riadenia.

---

27. Prečo FPGA nie je vždy najlepšie riešenie

FPGA má aj nevýhody.

Náročnejší vývoj

Návrh FPGA je náročnejší než písanie programu pre CPU. Treba rozumieť digitálnej logike, časovaniu, paralelizmu a HDL.

Dlhší návrhový cyklus

Syntéza, mapovanie, place & route a časová analýza môžu trvať dlhšie než obyčajná kompilácia softvéru.

Obmedzené zdroje

Každé FPGA má obmedzený počet LUT, registrov, DSP blokov, pamäťových blokov a I/O pinov.

Nižšia frekvencia než CPU/GPU

FPGA často beží na nižšej frekvencii než moderné CPU alebo GPU. Výkon získava skôr paralelizmom než vysokým taktom.

Zložité ladenie

Chyby v hardvérovom návrhu sa môžu hľadať ťažšie než chyby v softvéri.

---

28. Softvérové nástroje pre FPGA návrh

Prednáška spomína programový balík **Mentor Graphics HEP's Design, Verification and Test**. Ide o sadu nástrojov na návrh číslicových systémov na báze ASIC a FPGA.

Používané typy nástrojov:

Návrh číslicového systému

Nástroje ako HDL Designer alebo Visual Elite HDL umožňujú pracovať s blokovými a stavovými diagramami, VHDL, Verilogom a vyššími modelmi.

Syntéza

Precision RTL podporuje syntézu pre viacero FPGA a ASIC technológií, napríklad Xilinx, Altera, Actel, Atmel alebo Lattice.

Simulácia, verifikácia a prototypovanie

Questa alebo ModelSim umožňujú simuláciu a verifikáciu návrhu. To je kľúčové, pretože pred reálnym nasadením na FPGA treba overiť, či sa obvod správa správne.

Dôležité je pochopiť, že FPGA vývoj nie je len o napísaní HDL súboru. Potrebujeme celý nástrojový reťazec: návrh, simuláciu, syntézu, mapovanie, rozmiestnenie, prepojenie, časovú analýzu a nahratie konfigurácie.

---

29. Príklady FPGA hardvéru

Prednáška uvádza vývojové dosky založené na FPGA.

Kintex XC7K325T

Príklad parametrov:

- 326 080 logic cells,
- MicroBlaze soft-core približne 241–337 MHz,
- 1 GB DDR3 pamäť,
- Ethernet 10/100/1000,
- PCI Express,
- AD/DA prevodník,
- HDMI,
- LCD displej.

Táto doska predstavuje výkonnejšiu FPGA platformu vhodnú na zložitejšie návrhy.

Spartan-3 XC3S700AN

Príklad parametrov:

- 700K system gates,
- 13 248 logic cells,
- MicroBlaze soft-core približne 66,67 MHz,
- DDR2 pamäť,
- Ethernet,

- AD/DA prevodník,
- VGA,
- RS-232,
- LCD displej.

Tento typ je jednoduchší a vhodný na výučbu a základné experimenty.

---

30. FPGA a paralelné počítačové systémy

FPGA je v predmete Paralelné počítačové systémy dôležité preto, že umožňuje vytvárať vlastné paralelné architektúry.

Na FPGA môžeme realizovať:

- špecializované výpočtové jednotky,
- pipeline architektúry,
- paralelné dátové cesty,
- hardvérové akcelerátory,
- vlastné procesorové jadrá,
- komunikačné rozhrania,
- neurónové siete,
- DSP algoritmy,
- kryptografické obvody,
- spracovanie obrazu.

Z pohľadu paralelizmu je FPGA veľmi zaujímavé, pretože sa nesnaží vykonávať veľa softvérových vlákien ako CPU/GPU, ale umožňuje vytvoriť fyzicky paralelné obvody. Každá časť hardvéru môže pracovať súčasne.

---

31. FPGA benchmark a význam komunikačnej réžie

Prednáška uvádza aj zaujímavé porovnanie pri FFT benchmarku medzi CPU, GPU a FPGA.

Dôležitá myšlienka z benchmarku nie je len konkrétne číslo, ale hlavne to, že výkon závisí od toho, či započítame aj komunikačnú réžiu.

Bez zohľadnenia komunikačnej réžie môže GPU vyzerať extrémne rýchlo. Keď sa však započíta prenos dát medzi host systémom a akcelerátorom, výsledný pomer sa môže výrazne zmeniť.

To je veľmi dôležité pri akcelerátoroch všeobecne:

- samotný výpočet môže byť veľmi rýchly,

- ale prenos dát môže znížiť celkový prínos,
- preto treba hodnotiť celý systém, nie iba výpočtové jadro.

Táto myšlienka platí pre GPU aj FPGA. Ak sa dáta musia neustále presúvať cez pomalé rozhranie, výkon akcelerátora sa nemusí prejaviť naplno.

---

32. Ako rozmýšľať pri návrhu pre FPGA

Pri návrhu pre FPGA sa oplatí postupovať takto:

1. Určiť požadovanú funkciu

Najprv treba presne vedieť, aký digitálny obvod alebo akcelerátor chceme vytvoriť.

2. Rozhodnúť, čo bude kombinačné a čo sekvenčné

Kombinačná logika určuje výstupy podľa vstupov. Sekvenčná logika obsahuje registre a stav.

3. Navrhnúť dátovú cestu

Dátová cesta určuje, kadiaľ tečú dáta a aké operácie sa nad nimi vykonávajú.

4. Navrhnúť riadenie

Riadenie určuje, kedy sa čo aktivuje, ktoré registre sa zapisujú a aký stav má obvod.

5. Overiť návrh simuláciou

Pred nahratím do FPGA treba overiť správanie.

6. Syntetizovať a implementovať

Nástroje preložia HDL opis na konkrétnu FPGA konfiguráciu.

7. Skontrolovať časovanie

Návrh musí splniť požadovanú pracovnú frekvenciu a časové obmedzenia.

8. Nahrať konfiguráciu a testovať na doske

Až potom sa overuje reálne správanie na hardvéri.

---

33. Najdôležitejšie pojmy na zapamätanie

FPGA je programovateľné hradlové pole umožňujúce vytvoriť vlastný digitálny obvod.

Konfigurácia FPGA určuje funkciu logických blokov a prepojovacej siete.

Volatilné FPGA stratí konfiguráciu po vypnutí napájania a po zapnutí ju musí načítať napríklad z FLASH pamäte.

CLB / LAB je konfigurovateľný logický blok.

LUT je tabuľka, ktorá realizuje logickú funkciu podľa vstupov.

Klopný obvod umožňuje vytvárať registre a sekvenčné obvody.

Switch matrix / switch box zabezpečuje programovateľné prepojenie medzi blokmi.

I/O block pripája FPGA k vonkajším signálom.

DSP bloky urýchľujú aritmetické a signálové operácie.

On-chip memory je interná pamäť FPGA.

HDL je jazyk na opis hardvéru, nie klasický programovací jazyk.

VHDL, Verilog, SystemVerilog sú nízkoúrovňové jazyky na opis hardvéru.

SystemC, HLS, MyHDL sú vyššie úrovne opisu hardvéru.

Syntéza prekladá HDL opis na logickú hardvérovú štruktúru.

Mapovanie priraďuje logiku na konkrétne dostupné prvky FPGA.

Place & route rozmiestňuje komponenty na čipe a prepája ich.

Static timing analysis overuje, či návrh spĺňa časové požiadavky.

Setup time a hold time sú základné časové podmienky registrov.

IP core je hotový opakovane použiteľný hardvérový komponent.

ASIC je dedikovaný čip s výborným výkonom, ale bez flexibility.

FPGA je kompromis medzi flexibilitou softvéru a výkonom špecializovaného hardvéru.

---

34. Najdôležitejšie rozdiely

FPGA vs. CPU

CPU vykonáva program. FPGA sa nakonfiguruje ako hardvér.

CPU je jednoduchšie programovať, ale FPGA vie vytvoriť veľmi paralelné špecializované obvody.

FPGA vs. GPU

GPU je hotová masívne paralelná architektúra vhodná na dátovo paralelné výpočty. FPGA umožňuje vytvoriť vlastnú paralelnú dátovú cestu presne pre danú úlohu.

GPU sa programuje jednoduchšie než FPGA, ale FPGA môže byť efektívnejšie pri špecifických úlohách s vhodnou hardvérovou implementáciou.

FPGA vs. ASIC

ASIC je rýchlejší a efektívnejší, ale po výrobe nemeniteľný. FPGA je pomalšie než ASIC, ale preprogramovateľné.

HDL vs. klasické programovanie

V klasickom programovaní píšeme postupnosť inštrukcií. V HDL opisujeme štruktúru a správanie hardvéru, ktorý pracuje paralelne.

---

35. Hlavná myšlienka celej prednášky

Prednáška 12.b ukazuje FPGA ako programovateľnú hardvérovú platformu, ktorá umožňuje vytvárať vlastné digitálne obvody. FPGA sa líši od procesora tým, že nevykonáva program ako sekvenciu inštrukcií, ale po nakonfigurovaní sa správa ako konkrétne navrhnutý hardvér.

Základom FPGA sú logické bloky, LUT tabuľky, klopné obvody, programovateľná prepojovacia matica a I/O bloky. Moderné FPGA navyše obsahujú DSP bloky, interné pamäte, správu hodín a niekedy aj procesorové jadrá.

Návrh pre FPGA sa robí pomocou HDL jazykov, napríklad VHDL alebo Verilog. HDL nie je klasický programovací jazyk – opisuje hardvér. Z HDL opisu sa cez syntézu, mapovanie, rozmiestnenie, prepojenie a časovú analýzu vytvorí konfigurácia FPGA.

FPGA je výhodné tam, kde potrebujeme flexibilný, preprogramovateľný a paralelný hardvér. Je vhodné na prototypovanie, výskum architektúr, hardvérové akcelerátory, signálové spracovanie, spracovanie obrazu, špecializované výpočty a návrh vlastných digitálnych systémov.

Ak si z tejto prednášky zapamätáš jednu vetu, tak túto:

FPGA nie je procesor, ktorý vykonáva program, ale preprogramovateľný hardvér, z ktorého si pomocou HDL opisu vytvoríme konkrétny paralelný digitálny obvod.