12. Jednoprocesorové počítače, počítače s menším počtem procesorů, masivně paralelní počítače; distribuované systémy. Sdílená, distribuovaná a distribuovaná sdílená paměť; další alternativy. Masivně paralelní systémy, paralelní algoritmy, „jemný paralelismus. Distribuované systémy, dekompozice úloh, „hrubý paralelismus.

Jednoprocesorové počítače

Když se začaly vyvíjet počítače, bývaly doby, kdy paměť byla malá a drahá a přístupy do paměti obzvlášť pomalé. Programy musely být pokud možnost malé, aby se vešly do paměti, a byla snaha udělat většinu práce pomocí procesoru a jeho instrukcí. Instrukce byly složité a proměnné délky (CISC architektura), mohly provádět několik operací naráz, například vzít číslo z paměti, přičíst k registru a uložit do paměti – to vše v jedné instrukci, ale prováděly se i několik taktů. Velikost programu byla ale malá a na procesor se dal přímo napasovat programovací jazyk. Také nebyly dostatečně vyvinuté kompilátory.

Jak postupně instrukce rostly a stávaly se složitějšími, bylo nutné je začít rozkládat na jednodušší instrukce a vnikl tzv. *mikrokód*, který představuje v podstatě emulaci složitých instrukcí na úrovni procesoru. Přeprogramováním mikrokódu v procesoru mohl vzniknout procesor s jinými vlastnostmi. Takto fungoval např. IBM 360.

Tento přístup pak začal přinášet obtíže při požadavcích na vyšší výkon a také z důvodu zátěže zpětné kompatibility.

Proto se v polovině 60. let objevil opačný přístup tzv. RISC, který stavěl na malé sadě co nejjednodušších instrukcí, ale rychle prováděných. Zavedení RISC procesorů pomohl zejména:

- Rozvoj optimalizujících překladačů – nebylo nutné programovat ve strojovém kódu či assembleru.
- Zlevnění pamětí – tj. velikost programů začala být méně podstatná.

Vlastnosti RISC:

- Instrukce jednotné délky – pro co nejrychlejší zpracování
- Malý počet instrukcí – jen ty skutečně potřebné
- Dostatek registrů
- Jednotné kódování instrukcí
- Load/Store architektura – dvě instrukce pro práci s pamětí, nic víc.

CISC	RISC
- složité instrukce - kódy instrukcí různě dlouhé - krátké programy (z hlediska místa v paměti) - složitá adresace - 1 instrukce / několik taktů - jednodušší návrh HW **- malé nároky na překladače** - přístupy do paměti omezit za každou cenu - málo registrů	- jednoduché instrukce - kódy instrukcí stejně dlouhé - dlouhé programy - jednodušší adresace - 1 instrukce / 1 takt (dnes více instrukcí / 1 takt) - pečlivý výběr instrukcí **- vysoké nároky na překladače (optimalizace)** - přístupy do paměti tak moc nevadí - hodně registrů

- zátěž zpětné kompatibility	- problémy s kompatibilitou nejsou zásadní

RISC samozřejmě nebyla jediná myšlenka na zvyšování výkonu procesorů. V 80. letech se začalo přemýšlet nad tím jak zvýšit výkon procesoru i jinak než jen zvýšením taktu. Jednou z myšlenek bylo rozbít instrukce na malé součásti, které by se zpracovávaly paralelně pomocí pipeline.

Běžný rozklad mohl vypadat takto:

- Fetch instrukce
- Decode instrukce
- Fetch operandů
- Execute
- Write Back

přičemž jednotlivé instrukce jsou zpracovány paralelně s posunem o jednu část. Tím se procesor využil daleko efektivněji.

Pipelining

- překrývání instrukcí v různých fázích zapracovanosti
    - každou instrukci umíme dekomponovat na několik částí, zjistit co ta instrukce dělá, jaké má operandy, jak je získat z paměti, pak se provede vlastní výpočet, výsledek nějak reprezentovat. Zvýšení rychlosti tím, že lépe využijeme HW. Pokud se provede již část instrukce v sekci 1, pak se uvolní a může ji využít k provádění jiné instrukce.
    - Výpočty v pohyblivé řádové čárce – vlastní dedikovaný co-procesor. Oddělením celočíselné části od neceločíselné zvýšíme výkon.
- 2 přístupy k paměti
    - Neviditelný pipelining – aktuálně používáme, pipeline stejně dlouhé
        - Víme, co nás zdržuje, je získávání operandů, před ni dáme instrukci tak, aby zajistila, že jsou data opravdu připravena před vlastním vykonáním instrukce
        - Předsunutí čtení (zápisu) z (do) paměti před vlastní instrukci pracující s daty
    - Viditelný pipelining
        - Víme přesně jak máme rozbitou instrukci, víme, kterou část pipeliny budou zatěžovat například, aby nesoupeřili o execute.

Jedním z dalších nápadů byla *superskalární architektura,* která spouští více instrukcí na redundantních výpočetních jednotkách v procesoru. S tímto nápadem poprvé přišel pan Cray v roce 1965. Drobným problémem této architektury jsou závislosti mezi instrukcemi, které procesor musí kontrolovat.

Dále se v 80. letech objevily tzv. *Very Long Instruction Words*, které představovaly způsob zajištění paralelního provedení různých aritmetických operací již na úrovni překladače. Procesor tak pro paralelní spuštění instrukcí nemusel kontrolovat vzájemnou závislost mezi instrukcemi a mohl být jednodušeji navržen.

Pojem *superpipeline* je rozsáhlejší dekompozice pipeline, např. Pentium 4 má až 31 stupňovou pipeline.

S tím také souvisí pojem architektura ANDES (Architecture with Nonsequential Dynamic Execution Scheduling), která se vyznačuje tím, že:

- Nevykonává instrukce přesně tak, jak jsou zapsané v programu, ale dynamicky jejich posloupnost optimalizuje přičemž cílem je komplexní využití procesoru.
- Více výpočetních jednotek, které mohou pracovat současně např. ALU a FLOAT. Vícenásobné fronty instrukcí: **celočíselná fronta** pro celočíselné instrukce, **adresní fronta** pro operace Load/Store, **floating point fronta** pro operace pohyblivé řádové čárky. Každá má vlastní pipeline.
- Pořadí vykonávání instrukcí si řídí sám procesor (opět některé instrukce na sobě mohou být závislé) nikoliv překladač i když samozřejmě optimalizovaný překlad může pomoci ke zvýšení výkonu.
- Mezi další vlastnosti patří spekulativní skoky (nečeká se na výsledek operace if, ale rovnou se začne provádět předpovězená větev výpočtu), neblokující load/store.

Příkladem takové architektury je procesor MIPS 10000

Paralelní počítače

Hlavná myšlienka je spájať viacero procesorov za sebou za účelom získania vyššieho výkonu pri rovnakej alebo nižšej cene. Očakávalo sa, že výkon porastie exponenciálne, ale rástol logaritmicky, je to v dôsledku zložitej komunikácie medzi procesormy a tiež tzv. loadbalance pretože jeden procesor čaká na druhý kvôli algoritmom,ktoré nie sú rozparalizované. Podmienkami sú rozparalelnenie procesu, môže byť prirodzené, alebo neprirodzené

Architektury s více procesory lze dělit podle různých hledisek, například podle počtu procesorů:

- **Small scale multiprocessing:**
  - 2-16 procesorů
  - Výpočetní model je sdílená paměť
- **Large scale multiprocessing:**
  - > 100 procesorů
  - Výpočetní model je distribuovaná paměť

Nebo podle taxonomie pana Flynna z roku 1972:

- **SIMD (Single Instruction Multiple Data)**
  - Procesory pracují jako vektorové počítače, které jednou instrukcí zpracují prvky jednoho vektoru (pole data). Provádějí stejnou instrukci na více datech paralelně.
  - Jde o jednoduchý programovací model a i samotné procesory mohou být jednoduché.

- **MIMD (Multiple Instruction Multiple Data)**
    - Procesory zde pracují samostatně nad vlastními daty.
    - Složitější programovací model. Je třeba explicitní synchronizace.

Programovací modely:

- SMPD – single program multiple data – jeden program využívá více procesorů (například paralelizace cyklů, z pohledu procesu je to jedno vlákno), běžně využíván v MIMD architektuře
- MPMD – problém rozdělíme do pod úloh a ty pak necháme řešit, (například prohledávání stavového prostoru), jedna paralelní úloha poskládána z jednodušších úloh. Na MIMD zcela vhodné, na SIMD teoreticky možné. Potrebná synchronizácia

Konečně podle přístupu do paměti se paralelní výpočetní prostředí může dělit na:

- **Sdílenou paměť (Symetrický přístup)**
    - Každý procesor vidí celou paměť a každý má do ní "stejně daleko", připojena na sběrnici, uniformní přístup.
    - Programování lze realizovat relativně jednoduše, ovšem může nastat problém s paměťovou propustností a s čekáním procesorů mezi sebou.
    - Výkon vzhledem k omezené paměťové propustnosti s přidáním procesoru nevzrůstá lineárně
- **Distribuovaná paměť (Předáváním zpráv)**
    - Zde má každý procesor svou paměť a s ostatními procesory komunikuje zasíláním zpráv.
    - Výkon roste prakticky lineárně, ovšem za cenu vysoké komunikace mezi procesory a tedy i složitějšího programování.
- **Hybridní systémy (NUMA)**
    - Každý procesor má svou lokální paměť a v systému je ještě sdílená paměť. Lokální paměť si lze představit jako cache pro hlavní paměť.
    - Procesor však stále může adresovat celou paměť.
- **Distribuovaná sdílená paměť v clusterech**
    - Může být implementovaná v operačním systému, v tom případě je před programátorem skrytá nebo jako knihovna, která pak umožňuje explicitní programování.
    - Vystupuje zde lokální paměť daného uzlu a vzdálená paměť ostatních uzlů.
- **Cache-only memory access architecture (COMA)**
    - NUMA se speciální architekturou paměti (řeší problém s rychlou a pomalou pamětí)
    - dívá se na paměť jako na veliký pool vyrovnávacích pamětí, funguje stejně jako normální vyrovnávací paměti. HW se stará o to, aby se stránka neztratila. (systém si pamatuje kolik je kopií stejných informací – nemůže se přehrát poslední kopie dat), složitost hlídání kopií ve všech pamětech není triviální

Systémy se sdílenou pamětí musí nějakým vhodným způsobem řešit problém paměťové koherence. (různá data v různých vyrovnávacích pamětech - musí se sdělit ostatním procesorům o změně) Možné metody jsou založeny např. na broadcastech nebo snoopy cache, kdy procesor sleduje změny v hlavní paměti.

Propojovací sítě

Požadavky na ideální síť

- nízká cena rostoucí lineárně s počtem procesorů
- minimální latence nezávislá na počtu procesorů
- propustnost rostoucí lineárně s počtem procesorů

Základní komponenty propojovacích sítí jsou:

- Topologie (jak je propojena)
- Přepínání (Jak data proudí mezi uzly)
- Směrování (Jak data hledají cestu mezi uzly)

S topologií souvisí pojem *bisection width*, který určuje minimální počet linek, které je třeba odstranit, aby se systém rozpadl na dvě stejné části. Vztažená na jeden procesor je ideálně konstantní.
Dalším pojmem je *poloměr sítě*, což je nejdelší nejkratší cesta mezi dvěma uzly.

Podle topologie se sítě klasifikují na:

- **Jednorozměrné** - řetěz, má všechny špatné vlastnosti, veliký poloměr, bisection width, masivní síť
- **Dvourozměrné** – kruh, hviezda, strom (vnitřní uzly maji stupěň 3, malý poloměr sítě, stále špatná redundance a bisekce, tlustý strom přidává redundatní cesty ve vyšších úrovních), dvojrozmerna mriežka (poloměr, bisection, redundance, špatná – vyšší cena a proměný stupeň uzlu), torus (uzavřená dvourozměrná mřížka, vyšší cena – přidá mnoho hran)
- **Třírozměrné** – až hyperkryhle – maly polomer, bisection, konstrukčně složitá
- **Plně propojené sítě**. toeoretická konstrukce, vynikající poloměr (1), neakceptovatelná cena

Podle přepínání se sítě rozdělují na:

- Přepínání paketů - paket se uloží a přepošle, jednoduché (store&forward). - vysoká latence
- Přepínání obvodů (okruhy) – ustaví se spojení, přenáší se, zruší se spojení - výrazně nižší latence
- Virtuální propojení (cut – through) – rozdělení do menších bloků – flow control digits (flits) a posílá se kontinuálně, díky flitsům posíláme data ještě dříve než jsou data kompletně připravena
- Červí díra - speciální případ cut-through, buffery mají právě délku flitsu, latence nezávisí na vzdálenosti, podporuje replikace paketů - vhodné pro multicast (jedna data na více míst)

Podle směrování se sítě dělí na:

- Statické (Zdrojové nebo distribuované)

- Adaptivní (vždy distribuované)

Koherenci paměti u takovýchto schémat nelze založit z důvodu rozšiřitelnosti, proto se používají tzv. *adresářové přístupy*.

Např. plně mapované adresáře fungují tak, že každý blok paměti má u sebe seznam ukazatelů na lokální vyrovnávací paměti. Zpravila to není potřebné a používají se omezené adresáře, které se alokují dynamicky. Mezi další přístupy patří provázané seznamy nebo hierarchické adresáře.

Příklady přístupů

Multiprocessing

Např. běžné PC s dvěma procesory nebo procesorem se dvěma jádry a se sdílenou pamětí.

Cluster

Cluster je více propojených počítačů, které spolupracují tak, že v některých aspektech vystupují navenek jako jeden počítač.

Clusterů je mnoho druhů např.

- **High availability cluster** – dva nebo více počítačů, i když zpravidla dva pro zajištění redundance v případě výpadku.
- **Load balancing cluster** – pracují tak, že počítač vystupující jako front-end rozděluje práce jednotlivým počítačům v clusteru. Cílem je zvýšení výkonu, většinou poskytuje i funkce z první skupiny tj. zajištění redundance. Příkladem je projekt Linux Virtual Server.
- **Výpočetní clustery** – používají se primárně pro zvýšení výpočetního výkonu. Zpravidla jde o běžné počítače propojené na jednom místě lokální počítačovou sítí případně lokální počítačovou sítí uzpůsobenou pro nízkou latenci – předpokládá se, že paralelní procesy spolu budou intenzivně komunikovat. Pro programování takových výpočtů existují speciální knihovny. např. MPI. Příkladem clusteru je BeoWulf.
- **Gridy** – jsou podobné clusterům v tom smyslu, že jde opět o skupinu propojených počítačů. Počítače jsou zpravidla propojeny na větší vzdálenost buď Internetem nebo nějakým dedikovaným či ad-hoc vytvářeným kanálem. Gridy na rozdíl od clusterů jsou více heterogenní, nemusí jít o stejné počítače. Také jejich výpočetní prostředí je uzpůsobeno spíše pro samostatnou práci jednotlivých uzlů .
- **Masivně paralelní superpočítače** – masivně paralelní počítače fungují na základě těsně propojených procesorových uzlů s nějakým řídícím subsystémem. Subsystém má již zmíněný charakter sítě. Distribuovanápaměť zde znemožňuje uplatnění sdílených proměnných, takže procesy běžící na různých procesorových uzlech komunikují výhradně pomocí předávání zpráv.Dle Flynnovy klasifikace mají architekturu MIMD. Umožňují

mnohem větší škálovatelnost, která se pohybuje v řádu stovek, někdy i tisíců procesorových uzlů.

Masivně paralelní systémy, paralelní algoritmy, „jemný paralelismus.

Paralelní programování je o designu a implementaci a ladění počítačových programů, které jsou schopny využít paralelních systémů.

Amdhalov zákon

V praxi sa najčastejšie stretávame s problémamy, v ktorých sa striedajú úseky výpočtu, ktorý je treba realizovať čisto sekvenčne s úsekmi, ktoré sú vhodné pre paralérne spracovanie. Pri posudzovaní účinku paralerizácie výpočtu sa používa Amdhalov zákon. Predpokladajme, že určitý výpočet ide čiastočne paralerizovať. To znamené, že okrem častí, ktoré budú spracovávané seriovo (sekvenčne) na jednom procesore (napr. vstup a výstup dát) ide zvyšok výpočtu zadať medzi procesory tak, aby spracovanie prebiehalo na nich súčastne. Celkové zrýchlenie výpočtu určíme podľa vzorca:

$$S_z = \frac{t}{t.f_s + t.f_p/p} = \frac{1}{(1 - f_p) + f_p/p}.$$

p-procesory
fp je podiel paralerizovanej časti
fs je podiel seriovej časti

Dekompozice úloh

Paralelní programování rozděluje celkový problém do úkolů pro jednotlivé procesory a synchronizuje běh jednotlivých úkolů tak, aby na konci vznikl smysluplný výsledek.

Paralelní programování lze nahlížet také ze dvou úhlů:

- **Implicitní paralelismus** – o paralelizaci se stará kompilátor nebo nějaký jiný systém a automaticky alokuje práci procesorům.
- **Explicitní paralelismu** – programátor sám určí rozdělení práce.

Jemný paralelismus

Používají se u systémů se sdílenou pamětí. Vlákna běží v rámci jedné úlohy, tj. sdílí její kontext. Jednotlivé úlohy sú na seba previazané, a výsledok jedného podproblému závisí od výsledkov iných podproblémov. (problémy vhodné pre superpočítače, počítačový klaster alebo velmi tesne viazané PC klastre s rýchlou sieťou)

Distribuované systémy

MPI (Message Passing Interface)

Je komunikační rozhraní pro meziprocesorovou komunikaci v paralelních programech.
Vlastnosti:

- Přenositelnost, vazba na různé programovací jazyky, nezávislé implementace
- Nezávislá optimalizace na různé architektury, standardizováno.
- Funkcionalita – snaha pokrýt všechny aspekty meziprocesorové komunikace.
- Skupiny, kontext, tagy, synchronní, asynchronní zprávy, …

Určena pro clustery i gridy.

PVM (Parallel Virtual Machine)

Byl vyvinut na přelomu 80. a 90. let s cílem vytvářet virtuální superpočítač. Znamenal první krok pro vývoj distribuovaného počítaní. Síť byla tvořena démony *pvmd* běžícími na různých počítačích s různými OS a knihovny přilinkované k programům. Výpočetní model byly samostatné úlohy řešené na jednotlivých procesorech s komunikací pomocí zpráv.

Koordinační jazyk Linda

- pracuje se sdílenou tabulí přístupnou všem procesům zpracovávajícím úlohu
- číst, zapsat a mazat může každý proces
- má charakter asociativní paměti
- pomocí zápisu speciální n-tice do tabulky lze procesy i vytvářet
- jde o jednoduchý přístup

Hrubý paralelismus

Granularita paralelismu na úrovni samostatných úloh pro jednotlivé OS, kdy si jednotlivé úlohy pouze zasílají zprávy. Používá se MPI nebo PVM, distribuovaná paměť.
Jednotlivé podproblémy sú nezávislé od výsledkov iných podproblémov (tzn., že oneskorenie pri výpočte jedného nemá vplyv na iný výpočet). Takéto počítanie nazývame vysokopriepustné počítanie (vhodne pre volne-viazane PC siete).