# HIGH PERFORMANCE COMPUTING

* Formule 1 mezi počítači, drahé stroje špičkových parametrů
* nezáleželo na penězích, důležitý byl výkon
* specifické uživatelské skupiny
  + rozsáhlé simulace – jaderné elektrárny, atomový výbuch, ochlazení oceánů
  + modelování – digitální modely aut, letadel...
* požadavky rostou rychleji než výkon procesorů, narůstá teoretický výkon, klesá dosažený výkon
* je třeba pochopit:
  + architekturu použitého počítače (její silné a slabé stránky)
  + Proč jsou zdánlivě ekvivalentní matematické kódy různě rychlé?
  + způsoby měření reálného výkonu

# HIGH THROUGHPUT COMPUTING

* dlouhodobé efektivní využití počítačových systémů – velké množství menších úloh, aby se server nenudil
* podstatný je celkový čas zpracování, vhodná je stejná délka úloh, aby žádná nemusela čekat, využívají se clustery

# CO URČUJE VÝKON

* Latence – zpracování a přenos signálu (uvnitř procesoru, v paměti), přenos mezi pamětí a procesorem, zpoždění v paměti
* Rychlost – přepínání obvodů, frekvence obvodů, obnovení paměti (dynamické)
* Propustnost – instrukce per cyklus, přenos mezi komponentami, rychlost přenosu dat na chipu
* Granularita – hustota na chipu a na paměti, velikost úlohy

# CISC

* nedělej programem to, co může udělat hardware
* zpočátku počítačů, malé paměti a stejně malé programy (musely se vejít do paměti)
* velikost, cena a rychlost paměti = velikost, cena a rychlost procesoru
* přímá podpora překladačů (byly efektivní, znaly instrukce)
* málo registrů (nebylo potřeba, sahalo se rovnou do paměti)
* dlouhotrvající instrukce => mikroprogramování
* MIKROPROGRAMOVÁNÍ
  + dekompozice na menší instrukce, tzv. mikroinstrukce
  + složitá instrukce se nazývá mikroprogram
  + změna instrukční sady konkrétního počítače se dělala výměnou diskety
  + programy se neprodávaly, byly zdarma
  + nevýhodou jsou příliš složité instrukce, zátěž zpětné kompatibility a zvyšující se složitost analýzy instrukcí
* ZVYŠOVÁNÍ VÝKONU
  + omezeno aktuálními technickými možnostmi
  + nelze neomezeně zvyšovat závislost mezi komponentami a rychlost šíření signálu
  + řeší se paralelizací procesů (přidáním procesoru)

# PIPELINING

* překrývání instrukcí v různých fázích rozpracovanosti
* tři základní oblasti:
  + zpracování instrukcí
  + přístupy k paměti
  + výpočty v pohyblivé řadové čárce
* nezvyšujeme výkon jedné instrukce, ale všech dohromady
* rozklad instrukcí:
  + Instruction Fetch – načtení instrukce z paměti
  + Instruction Decode – rozeznání instrukce (je to skok, přesun, násobení?)
  + Operand Fetch – operandy připraveny (nejproblematičtější)
  + Execute – provedení instrukce (mám operandy, vím co to je)
  + Writeback – zapsání zpět do paměti/do registru (realizace)
* paralelní zpracování instrukcí s posunem o jednu fázi na pipeline
* NEVIDITELNÝ PIPELINING – přesunutí čtení/zápisu z/do paměti před vlastní instrukci pracující s daty
* VIDITELNÝ PIPELINING – přesně definovaný počet cyklů do dokončení

# RISC

* zavedení vyrovnávací paměti (cache)
  + je-li hodnota v cache paměti, je to rychlejší
  + paměti byly pomalejší než procesory, které na ně pak musely čekat
* pokles ceny a nárůst velikosti hlavních pamětí
* lepší pipelining, kvalitní optimalizující překladače
* využití interních registrů (méně přístupů do paměti, RISC měl více registrů než CISC, vše se v nich muselo uchovávat)
* i rozsáhlé programy se vešly do paměti
* problémem bylo zdržení – čekání na výsledek předchozí instrukce
* zavedena jednotná délka instrukcí
* redukce počtu instrukcí – pečlivý výběr
* jednoduché adresní módy
* Load/store – jediné dvě instrukce, které pracují s pamětí – nevíme, jak dlouho budou načítat, zapisovat
* 1 instrukce/1 tik (dnes několik instrukcí za 1 tik)
* odložené skoky
  + provede se několik následujících instrukcí než se provede skok
  + ušetří se tím čas, když se vrátí na místo (po skoku se totiž pipeline vysype – výsledky se zneplatní, a ztratí tak všechno, co si „natahala“)

# SUPERSKALÁRNÍ PROCESORY

* více procesních jednotek
* zvyšování výpočetního výkonu
* čipy, které dokážou v každém strojovém cyklu začít zpracovávat více instrukcí naráz
* mikroprocesory vybaveny dvěma a více pipeline, někdy i více ALU
* musíme hlídat paralelní části
* INTERNÍ PARALELISMUS
  + pro programátora je skrytý, implementován v HW superskalárních procesorů
  + dovoluje urychlit jednotlivé části pipeline, například více execute nebo instruction fetch, protože výpočet je pomalejší než natažení z cache, míň sahání do cache
  + dochází ke zrychlení výpočtu
  + multiply and add MADD (nejdou-li dvě pustit naráz – 1. zapisuje, 2. čte), čeká se

# VERY LONG INSTRUCTION WORD – VLIW

* obdoba superskalárů
* paralelizaci kontroluje překladač
* pevná délka instrukcí (několik desítek až stovek bitů)
* instrukční slova rozdělena na pole (fields), v každém poli je kód jedné operace
* potenciál nižší spotřeby energie, není třeba složitý hardware

# ANDES

* zpomalení je způsobeno čekáním na data
* vícenásobné fronty instrukcí, každá má svou pipeline
* instrukce jsou vybírány podle připravenosti, ne podle pořadí
* dokončení instrukcí je zajištěno správným uspořádáním
* spekulativní kroky – výpočet pokračuje předpovězenou větví, nečeká na výsledek instrukce
* neblokující Load/Store
* přejmenování registrů - instrukce č. 1 je součet dvou prvků v registru 01, instrukce č. 2 je použití součtu, instrukce č. 3 je přepsání registru 01 jinou hodnotou, registr 01 tak můžeme přejmenovat na 02, tím může instrukce č. 1 a instrukce č. 3 probíhat paralelně

# OUT-OF-ORDER

* metoda, kdy se instrukce vykonávají v jiném pořadí, než jim bylo uloženo programem z operační paměti, procesor si sám rozhodne, která instrukce bude vykonána tak, aby všechny byly vykonány v co nejkratším čase, je to důsledek toho, že ne všechny instrukce mohou být vykonávány naráz nebo se nedají stejně rozpracovat = některé části mikroprocesoru nebyly plně využity po celou dobu

# PAMĚŤ

* matice – řádky a sloupce, čtení a zápis
* page mode – čtení související skupiny bytů naráz
* přístupová doba – vypiš řádek, vypiš sloupec, vypiš data
* cyklus paměti – jak často lze data z paměti číst
* hlavní paměť je neperzistentní (neudrží data při výpadku proudu), čtení z ní musí být nedestruktivní, nestačí-li, swapuje na disk

# VIRTUÁLNÍ PAMĚŤ

* adresový prostor tvoření logickou adresou, která se překládá na fyzickou adresu
* TLB MISS – je součástí hardware, překládá logickou adresu na fyzickou, není-li logická adresa načtena, musí ji získat, vypočítat, načíst

# VYROVNÁVACÍ PAMĚŤ

* hit poměr – udává efektivitu, poměr čtení z cache vůči všem operacím
* PŘÍMO ADRESOVANÁ – statické mapování, každý řádek v cache odpovídá předem určeným oblastem hlavní paměti, je rychlá, jednoduchá, potenciálně neefektivní
* PLNĚ ASOCIATIVNÍ – dynamické mapování, každý řádek zná adresu bloku, současný dotaz na všechny řádky, výběr řádku pro zneplatnění, velmi složité a drahé, velmi efektivní
* MNOŽINOVĚ ASOCIATIVNÍ – kombinace lepších vlastností předchozích dvou

# PARALELNÍ POČÍTAČE

* SMALL SCALE MULTIPROCESSING – převážně sdílená paměť, jeden paměťový prostor, kam můžou všichni přistupovat
* LARGE SCALE MULTIPROCESSING – distributivní, posílají se zprávy
* PROGRAMOVACÍ MODELY
  + SIMD – stejná instrukce, různé procesory, synchronizováno, jednoduché procesory, jednodušší programovací model
  + MIMD – plně asynchronní, každý procesor má vlastní data, vlastní operace, náročné na programování, větší flexibilita, teoreticky větší efektivita
  + SPMD – paralelní, rozpad na úlohy řešené na více procesorech, iluze jednoho operačního systému
  + MPMD – tam, kde je více OS
* KOMUNIKAČNÍ MODELY
  + SDÍLENÁ PAMĚŤ – paměť oddělená od procesorů, uniformní přístup k paměti, sběrnice, nevýhoda – hodně odkazů do paměti, 2 procesory spolu soupeří o komunikační cestu
  + PŘEDÁVÁNÍ ZPRÁV – každý procesor je viditelný, má vlastní paměť, vysoká cena – zprávy v obálkách (zabalení, rozbalení), procesory nemají problém dokud si nemusí předávat data

# SNÍŽENÍ ZPOŽDĚNÍ – hybridní systémy (je to tam pak ještě jednou, jinak)

* NUMA
  + přístup k fyzickým adresám trvá různou dobu
  + vyšší škálovatelnost, nižší propustnost, cache – ccNUMA
  + home-node – kam patří daná schránka v paměti
* COMA
  + NUMA s charakteristikou vyrovnávací paměti
  + nemá home-node, vytváří kopie dat, původní zneplatňuje, musíme dávat pozor, ať nám jedna zbyde
* DISTRIBUOVANÁ PAMĚŤ
  + lokální paměť u každého uzlu a vzdálená u ostatních = fikce jedné rozsáhlé paměti
  + clustery
  + HW – principy virtuální paměti, transparentnost (pokud je maximální, byl hloupý programátor)
  + SW – knihovny, netransparentní (programátor musí explicitně přizpůsobit)

# KOHERENCE VYROVNÁVACÍCH PAMĚTÍ

* zavedeno, když paměti začaly pokulhávat za procesory
* řeší se u multiprocesorů s distribuovanou pamětí, znamá, že ve všech pamětech jsou stejná data a je třeba propagovat změnu do všech pamětí, čehož není jednoduché docílit
* PŘÍČINY VÝPADKŮ:
  + Compulsory miss – první načtení, data ještě nejsou v cache
  + Capacity miss – nedostatečná kapacita, data se musí přepsat nebo zapsat jinam
  + Conflict miss – různé adresy mapovány na stejné místo
  + Coherence miss – pro multiprocesory, různá data v různých cache, musíme druhé cache, která s nimi pracuje, říct, že se to v první změnilo
* ŘEŠENÍ:
  + vyrovnávací paměti musí vědět o změně
  + metody založené na broadcastu (snoopy cache – zneplatnění, update) – zpráva, kterou dostanou všechna připojená síťová rozhraní
  + adresářové metody ( plně mapované, částečně mapované, provázané)
* BROADCAST – SNOOPY CACHE
  + propojovací sítě s „přirozeným“ broadcastem – sběrnicí, každý procesor sleduje všechny přístupy k pamětem – jeden procesor požádá o přístup, vidí to všichni ostatní
  + ZNEPLATNĚNÍ – reakce na změnu ve vzdálené cache paměti, řádka v aktuální vyrovnávací paměti je zneplatněna, při opětovném přístupu je přehrána ze vzdálené paměti
  + UPDATE – okamžité obnovení, při opětovném přístupu již k dispozici, nevýhodou je falešné sdílení (děláme operace, které nejsou potřeba)
  + nelze rozhodnout, co je obecně lepší
  + ne u složitějších propojovacích sítí
* ADRESÁŘOVÉ METODY
  + PLNĚ MAPOVANÉ ADRESÁŘE – každá adresářová položka má tolik údajů, kolik je vyrovnávacích pamětí, bitový vektor přítomnosti (má/nemá kopii bloku paměti), příznak exkluzivity, příznak platnosti
  + OMEZENÉ ADRESÁŘE – informovaná je jen explicitně uvedená vyrovnávací paměť, velikost paměti P^2\*M/B, P – počet vyrovnávacích pamětí, M je velikost hlavní paměti, B je velikost bloku; výhodnější než přímo mapovaná pokud k< P/log2 P, kdy log2 P je počet bitů ukazatele, k je počet položek v poolu ukazatelů
  + PŘETEČENÍ – příliš mnoho sdílených bloků, zneplatnění všech sdílených nebo vyberu jednu a tu zneplatním
  + PROVÁZANÁ SCHÉMATA – tam, kde počty sdílených řádků nejsou vysoké, nemáme problém s kapacitou + minimální nároky na paměť, jediný centrální ukazatel (řádek) - ostatní provázány s vyrovnávací pamětí (s řádky), nevýhodou je zvýšená komunikace a dlouhý zápis, procházení seznamu
  + HIERARCHICKÉ ADRESÁŘE – sběrnice (hierarchie snoopy protokolů), nevýhodou je, že vyšší úrovně musí držet kopie paměťových bloků sdílených nižší úrovní
* ROZŠIŘITELNOST
  + výkon roste lineárně s cenou
  + je zachován konstantní poměr Ceny/Výkon
  + Míra rozšiřitelnosti – změna výkonu přidáním procesoru, změna ceny přidáním procesoru, smysluplný rozsah počtu procesorů
* ZRYCHLENÍ
  + teoretický pojem, realita závisí na aplikaci
  + vždy jsou části, které nelze paralelizovat

# ROZŠIŘITELNOST PROPOJOVACÍ SÍTĚ

* nízká cena rostoucí lineárně s počtem procesorů N (počet „drátů“)
* minimální latence nezávislá na N (nezahltitelná síť)
* propustnost rostoucí lineárně s N (pořád je místo)

# VLASTNOSTI SÍTÍ

* TOPOLOGIE, PŘEPÍNÁNÍ DAT, SMĚROVÁNÍ DAT
* ROZLIŠUJEME TYTO PARAMETRY:
  + velikost sítě – počet uzlů N
  + stupeň uzlu d – maximální stupeň v síti
  + poloměr sítě D – 2 prvky, které to k sobě mají nejdál (nejdelší nejkratší cesta)
  + bisection width B - minimální počet linek, které je třeba odstranit, aby se síť rozpadla na dvě STEJNÉ části, bisection bandwidth – propustnost při rozpůlení (celková propustnost odstraněných linek)
  + redundance sítě A – minimální počet linek, které je třeba odstranit, aby se síť rozpadla na dvě části
  + cena C – počet komunikačních linek v síti

# TOPOLOGIE

* JEDNOROZMĚRNÉ
  + prvky navázány na sobě, lineární pole - korálek
  + nejjednodušší, nejhorší vlastnosti, D = B = 1
* DVOUROZMĚRNÉ
  + kruh (uzavřené lineární pole)
  + hvězda
  + strom (D = 2log \*((N+1)/2), špatná redundance i bisection (band)width, má úzkou část v místech, kde se strom dělí na více větví
  + dvourozměrná mřížka (populární, vyšší cena, proměnlivý stupeň uzlu)
  + torus (vyšší cena)
* TROJROZMĚRNÉ
  + složitá konstrukce, problém s chlazením
* HYPERKRYCHLE
  + 4rozměrný útvar převedený do 3rozměrného
  + poloměr = redundance = log N, bisection N/2, vyšší cena (N log N)/2
  + statický směrovací algoritmus – z pozice dvou uzlů zjistíme jejich vzdálenost
  + cenou platíme za lepší vlastnosti
* PLNĚ PROPOJENÉ SÍTĚ
  + teoretické
  + poloměr 1 (vynikající), každý uzel stupně N-1, drahé

# PŘEPÍNÁNÍ

* od internetového přepínání se liší, u sítí může být dynamická i statická (obzvlášť pro malé sítě, lépe se tam optimalizuje díky znalosti polohy zařízení), kdežto internet se skládá z velkého počtu malých kusů, kde nemůžeme zaručit, že se něco nebude měnit, pořád posíláme tabulky
* STORE-AND-FORWARD
  + paket se uloží, zabalí, pošle
  + jednoduché, vysoká latence
  + nepředpokládáme chybovost, nemusíme kontrolovat
* PŘEPÍNÁNÍ OKRUHŮ
  + ustanoví spojení, přenese, zruší
  + zavře cestu napevno, je-li paket barevný, tak si jej nevšímá
  + latence nezávislá na délce cesty, ale na kapacitě
  + vhodné pro paralelní počítače
* VIRTUÁLNÍ PROPOJENÍ – CUT THROUGHT
  + zpráva rozdělena do menších bloků – flits
  + 1. flits obsahuje informaci o cestě (uvolní tak místo), 2. data, poslední ruší cestu
  + flitsky se posílají spojitě, nepřetržitě
* ČERVÍ DÍRA – WORMHOLE ROUTING
  + speciální virtuální propojení
  + délka bufferu = délka flits – analogie pipeline, paket je rozložen v bufferu několika uzlů
  + kontinuální proud zahltí celou cestu
* VIRTUÁLNÍ KANÁLY
  + několik bufferů nad stejným kanálem
  + využívá se, když je přetížené spojení
  + jako zábrana deadlocku (úspěšné ukončení druhé akce závisí na úspěšném dokončení první akce)
  + mapování logické topologie na fyzickou topoogii

# SMĚROVÁNÍ V PROPOJOVACÍCH SÍTÍCH

* STATICKÉ
  + zdrojové (zdrojový uzel rozhoduje)
  + distribuované (rozhodneme se lokálně na daném uzlu)
* ADAPTIVNÍ – víme například, které uzly spolu budou kdy komunikovat
* minimální – nejkratší nejdelší cesta, byl-li schopen spočítat cestu, není nekonečná smyčka, není nekonečná cesta
* neminimální – riziko vzniku cyklů nejde, kontrola, zda nejde 2x do jednoho uzlu (ne v dynamické podobě)
* u hyperkrychle můžeme ovlivnit její cestu, i když je tam více možností
* v internetu je přirozený požadavek, aby zpráva šla nejkratší cestou

# FAULT TOLERANCE PROPOJOVACÍCH SÍTÍ

* odolnost proti výpadku
* internet je schopen jakýkoliv lokální výpadek izolovat, u počítačů je pravděpodobnost výpadku asi 5 hodin, zhroutil by se systém, tzn. restart, nevhodné
* systém musí některé typy chyb sám detekovat samo opravné kódy (detekce i napravení)
* KONTROLA CHYB – zamezení duplicit kontrol, na nejnižší úrovni
* POTVRZOVÁNÍ ZPRÁV – na nejnižší úrovni, počítáme s tím, že špatné vůbec nepřevezmeme (na internetu bychom potvrzovali tomu, kdo vysílá, v počítači by hrozilo zahlcení, navíc zbytečné zahřívání)
* OPAKOVANÉ ZASÍLÁNÍ ZPRÁV – tam, kde provádíme kontrolu (nejnižší vrstva), po každém přenosu, z druhé strany se buffery smažou až potvrdíme, že je už nebudeme potřebovat

# ZPOŽDĚNÍ PAMĚTI

* vypořádání se s latencí v přístupu k datům
* čekání na paměť snižuje výkon systému, je totiž výrazně pomalejší než procesor
* VYVÍJENÍ NOVÝCH TECHNOLOGIÍ (zrychlují komponenty, zvětšují propustnost)
* SNÍŽENÍ ZPOŽDĚNÍ – zrychlení přístupu
  + k procesoru co nejblíž dáme nejrychlejší cache paměť a chceme, aby právě naše data v ní byla včas
  + NUMA – každé logické adrese odpovídá konkrétní fyzická adresa, pomáhá snížit latenci tak, že přitahuje data blíž k rychlejší paměti v neuniformní architektuře (bližší a vzdálenější paměti) tím, že máme právě jednu kopii u nás, zbytek jsou odkazy, minimalizuje počet přístupů ke vzdáleným pamětem
  + COMA – rozšíření vyrovnávací paměti, děláme kopie (musíme řešit koherence), pozor, ať nám při jejich mazání nějaká zbyde, nejrychlejší paměť je nejblíž a data jsou v ní, rozšíření principu vyrovnávacích pamětí, neuniformní architektura
* UKRYTÍ ZPOŽDĚNÍ
  + SLABÁ KONZISTENCE – propagace hodnot až je třeba, není nutné striktní uspořádání ke sdíleným proměnným (vyjma synchronizačních), RELEASE CONSISTENCY – acquire a release – pracujeme s proměnnou jako s lokální, neřešíme cache coherenci (řeší programátor), FENCE OPERACE – vynucení dokončení rozpracovaných operací, dáme plot, tam se zastaví, u sdílených proměnných si vymění hodnoty, pak mohou pokračovat, za plotem vidí všichni jeden stejný proces, drahé
  + PREFETCH – abychom nezdržovali se zpracováním hodnoty, vydáme s dostatečným předstihem požadavek, aby se hodnota dostala do nejbližší cache, BINDING FETCH – přesunutí až k procesoru (možnost porušení konzistence – zamrazíme cestu z paměti do registru), NONBINDING FETCH – častější, pouze v cache, HW PREFETCH – deep pipeline, rychle zjišťuju, která data potřebuji, vydám požadavek, jaká chci data, přeskládám pořadí v pipeline, SW PREFETCH – speciální instrukce v instrukční sadě, PREFETCH-EXCLUSIVE – uzamykání proměnných před ostatními instrukcemi (přečte a zamkne)
  + PROCESORY S VÍCENÁSOBNÝMI KONTEXTY – experimentální, větší množství registrů, přepneme do jiného procesu (programu), kde je to již připraveno, využito u vícejaderných procesorů
  + KOMUNIKACE INICIOVANÁ PRODUCENTEM – ten, kdo produkuje výsledek, jej přímo uloží do místa, kde ho bude potřebovat zpracovávající, ten přijde, hned má data (nečeká - což minimalizuje zpoždění), vhodné pro velké bloky, synchronizaci zámky

# PODPORA SYNCHRONIZACE

* dva předtím samostatné procesy musí „něco“ vidět stejně
* když pustíme výpočet opakovaně, chceme stejný výsledek
* VZÁJEMNÉ VYLOUČENÍ
  + garantujeme, že k jedné proměnné má přístup nejvýše jeden proces (nesmí být vykonává no více kritických kódů nad stejným sdíleným prostředkem), univerzální, drahé
  + synchronizační prostředek
  + potřebujeme podporu hardware, když ji nemáme, je to těžké (velká latence, asi jsou to semafory a monitory)
  + nesprávně použití – zpomalení procesu – blokování procesů ze vzájemného čekání, případně uváznutí dvou a více procesů (je nutno alespoň jeden ukončit)
  + test&set – atomicky (bez přerušení) zjistí hodnotu proměnné a nastaví ji, máme proměnnou zámek, ten má hodnotu 0 (OPEN) nebo 1 (CLOSED). Je-li OPEN, dáme CLOSED, je-li CLOSED, čekáme, až jiný proces tuto hodnotu pustí (měl by existovat), HW podpora znamená, že přečíst si může jenom jeden proces, nevýhodou je, že všichni jsou v jednom cyklu, zatěžují se všechny procesory

char \*lock;

while (exchange(lock, CLOSED) == CLOSED);

* + test-and-test&set – nekombinuje zápis a čtení, nejdřív čtení, vidím CLOSED – všichni ostatní čtou z lokální paměti (kopie), nezatěžuje se tolik paměť

for(;;)

while (\*lock == CLOSED);

if (exchange(lock == CLOSED) != CLOSED)

break;

* + FRONTY - Fronty vlaků (procesů), dva přijedou naráz, první dostane zelenou, druhý červenou, až první odjede, rozsvítí se zelená druhému (nebo dalším v pořadí, kdo přijel dřív, má přednost)
  + SEMAFORY – vlak, MONITORY – systém má kapacitu
  + ZÁMKY
  + KRITICKÁ SEKCE – je oblast ve zdrojovém kódu, kde dochází k přistupu ke sdílenému prostředku, ke kterému nemůžou naráz přistupovat dva nebo více procesů či vláken

# OPTIMALIZACE PŘEKLADAČŮ

* fáze 1 - překlad do metajazyka, fáze 2 – optimalizace (meziprocedurální analýza, optimalizace cyklů, globální optimalizace), fáze 3 - generování kódu
* optimalizace pro konkrétní HW nebo procesor se provádí na úrovni kódu, snižuje to práci
* MEZIJAZYK – čtveřice
  + Instrukce: operátor, dva operandy, výsledek
  + Paměť – přes dočasné tn
  + Skoky – podmínka počítána samostatně
  + Skoky – na absolutní adresy
* ZÁKLADNÍ BLOKY
  + program je reprezentován grafem toku
  + blok je část programu bez skoků má jeden vstupní a jeden výstupní bod
  + optimalizuje se uvnitř bloků – odstranění opakovaných (pod)výrazů, odstranění přebytečných proměnných
  + podobně i mezi bloky hlídáme, aby vysílaná hodnota přišla (neztratila se)
* KLASICKÉ OPTIMALIZACE
  + propagace kopírováním

X = Y => X = Y

Z = 1. + X Z = 1. + Y

* + zpracování konstant

Y :=3, X:=Y => X:=3

* + odstranění mrtvého kódu

nedosažitelný kód, bude to přehlednější, ušetří se cache pro instrukce

* + strength reduction (mocniny –> násobení)

K\*\*2 => K\*K

* + přejmenování proměnných

ne na úrovni registrů, předcházíme synchronizaci

* + odstranění společných podvýrazů

k = j \* 2, m = j \* 2

* + přesun invariantního kódu z cyklu

nemá uvnitř cyklu smysl

zjednodušení indukčních proměnných

* + přiřazení registrů proměnným

časté proměnné držíme přímo v registrech

* ODSTRAŇOVÁNÍ SMETÍ
  + netriviální, optimalizace programu uvnitř cyklu (cykly zdržují)
  + podpora makra (inlining)
  + podmíněné výrazy (reorganizace)
  + podmíněné výrazy uprostřed cyklů – skoky snižují efektivitu, vysype se pipeline
  + konverze typů (double a single)
  + ruční optimalizace – společné podvýrazy, přesun kódu, zpracování polí)

# OPTIMALIZACE CYKLŮ

* Cíle:
  + redukce režie
  + zlepšení přístupu do paměti (čím více load/store, tím hůř; chceme vše do registrů)
  + zvýšení paralelismu (rozvití závislostí, ne na úrovni paralelismu procesorů)
* krátké cykly se unrollujou na sekvenci několika příkazů
* dlouhé se rozsekají a jednotlivé úseky se řeší paralelně
* INLINING – optimalizace, která rozvine funkce v místě jejich volání, snižuje režii nepoužitím volání funkce a return, snižuje počet prováděných skoků, nevýhoda je, že navyšuje objem kódu (nemusí se vlézt do cache = capacity miss a snížení výkonu)
* DATOVÉ ZÁVISLOSTI
  + máme cyklus, pracujeme s poli, snažíme se rozvíjet závislosti mezi jednotlivými iteracemi cyklu
  + FLOW/BACKWARD DEPENDENCIES – zpětná, hodnota i-té iterace, závisí na i-1. iterace, rozepíšeme přiřazení do dvou po sobě následujících přiřazení = rozvineme cyklus
  + ANTI-DEPENDENCIES – přejmenování proměnných, rozvineme do několika cyklů
  + OUTPUT-DEPENDENCIES – bude potřeba jenom některé hodnoty, zanalyzujeme, co není potřeba, nebudeme vykonávat, zjistím, co je opravdu na konci
  + optimalizace je obtížná, jsou-li například použity globální proměnné, dynamické pole nebo se používají ukazatele

# ROZVOJ CYKLŮ

* tělo se několikrát zkopíruje
* hlavním smyslem je snížení režie (snížení počtu průchodu cyklem) a zvýšení paralelizace
* adaptace na skutečný počet průchodů
* nevhodné cykly:
  + malý počet iterací – malé „n“ – úplný rozvoj cyků
  + tlusté cykly – plno instrukcí, dostatek příležitostí k vlastní paralelizaci
  + cykly s voláním procedur – lepší rozvinout, abychom nemuseli skákat
  + cykly s podmíněnými výrazy – mají-li Andes, nevadí nám
  + rekurzivní cykly (s vnitřní závislostí a[i] = a[i] + 2)
* problémy:
  + rozvoj špatným počtem iterací,
  + zahlcení registrů (vše máme optimálně až na registry)
  + výpadky cache instrukcí
  + HW problémy, jsou potlačeny ostatními, ale je to těžké odchytit
  + SPOJOVÁNÍ CYKLŮ – opakované použití dat, větší tělo cyklu, kompilátor to zvládá sám, pokud mezi cykly není jiný kód, jedná se o nezávislé cykly
  + ROZVOJ VNĚJŠÍCH CYKLŮ

## OPTIMALIZACE PŘÍSTUPU K PAMĚTI

* nejlépe je mít vše v registrech
* optimální je dělat jeden krok – pracovat s cache
* pracuje se s maticemi
  + C ukládá po řádcích, nejrychleji se mění pravý index
  + Fortran ukládá po sloupcích, nejrychleji se mění levý index
* obrácení indexu („j“ vyměním za „i“)
* skákání do bloků
* nepřímá adresace
* použití ukazatelů (komplikujeme situaci kompilátoru)
* nedostatečná kapacita paměti (přetečení paměti je moc drahé)
  + ruční zpracování
  + virtuální paměť – dovolíme virtualizovat?

# BENCHMARKING

* související je PROFILLING – identifikace míst programu, kde to trvá nejdéle, zaměřujeme na ně pozornost
* je to snaha o porovnání systémů, hardware i software společně
* neexistuje žádné universální řešení
* průmyslové a privátní (můj program) benchmarky
* MIPS A MFLOPS
  + srovnání na základě vykonaných počtů instrukcí za sekundu
  + MIPS – počet celočíselných instrukcí za sekundu
  + MFLOPS – počet operací s pohyblivou řádovou čárkou za sekundu
  + problém s tím, že nevíme jaká instrukce a v jaké posloupnosti
  + je to umělé a nevypovídající
* CELOČÍSELNÉ
  + VAX MIPS (minipočítač)
  + DHRYSTONES
* S POHYBLIVOU ŘÁDOVOU ČÁRKOU
  + WHETSTONE (nepracuje se s ním)
  + LINPACK (lineární algebra)
* SPEC BENCHMARKS
  + nezávislá organizace, standardizované benchmarky pro různé architektury
  + vychází z kernel kódů (ne z umělého kódu), využívá celý program nebo jeho část, programy jsou dostupné ve zdrojovém kódu, každý si může přizpůsobit, využívá je, protože – spotřební koš ČSÚ – průměrná inflace, vezmeme co jsme koupili (blatník, 4 jogurty, 1 noviny), poskládáme, podíváme se, jak se ceny pohnuly oproti minulému období = provede se analýza často používaných složitých programů a z nich se poskládá mix (celé nebo část) a hodnota toho SPEC B. se spočítá jako vážená funkce rychlosti vykonání těchto programů
  + musí dávat stejné výsledky i po optimalizaci (kernel) kódu
* TRANSAKČNÍ BENCHMARKY
  + výkon databází
  + TCP-A – databáze zákazníků, test s bankomaty, kolik bankomatů je systém schopen obsloužit za sekundu
  + TCP-B – jako TCP-A, ale ne po síti
  + TCP-C – nejčastější, komplexní test, transakce objednávek a pod.
* SÍŤOVÉ BENCHMARKY
  + netperf, iperf, End to end měření (vezmeme koncové a změříme mezi)
  + pozor na to, co skutečně měříme (důležitá je izolace)
* VLASTNÍ BENCHMARKY
  + specifické požadavky
  + co testujeme, jak dlouho to budeme testovat, jaké jsou požadavky na velikost paměti
  + čím menší program, tím víc musíme přemýšlet
* KONTROLY
  + měříme opravdu to, co chceme?
  + příčiny ovlivnění jsou v použité optimalizaci, velikosti paměti a přítomnosti jiných procesorů
  + je třeba kontrolovat výsledky a srovnávat se „známým“ standardem

# MPI

* DATA PARALELISMUS
  + stejné instrukce na různých procesorech zpracovávají různá data
  + odpovídá SIMD modelu
  + zpracovává se celý vektor dat současně
* TASK PARALELISMUS
  + paralelně prováděné nezávislé bloky
  + MIMD model, různé instrukce, různá data
* SPMD
  + není synchronizován na úrovni jednotlivých instrukcí
  + jeden stejný program přes více uzlů
  + ekvivalent MIMD
* určeno pro MIMD a SPMD
* zaměřuje se na to, jakým způsobem spolu uzly komunikují
* standardizováno
* řada nezávislých implementací – optimalizace pro konkrétní HW, problémy s interoperabilitou
* VÝVOJ MPI
  + pomalý, ale kvalitní, aby nepadala letadla
  + Verze 1.0 nebyla implementována
  + Verze 1.1 byla implementována
  + Verze 1.2 je nejčastější
  + Verze 2.0 je experimentální, paralelní rozšíření I/O (komunikace mezi běžícími paralelními procesy), jednosměrné operace (put, get), manipulace s procesy (přidat i zaniknout)
  + Verze 3.0 aktuální standard, obohacení 2.0
* CÍLE NÁVRHU
  + přenositelnost – definice standardu (API), vazba na různé programovací jazyky a na nezávislé (i exotické) platformy
  + výkon – nezávislá optimalizace pro konkrétní HW
  + funkcionalita – snaha pokrýt všechny aspekty meziprocesorové komunikace
  + specifikace knihoven pro podporu předávání zpráv
  + zpřístupnění paralelního HW pro uživatele, autory knihoven a vývojáře nástrojů a aplikací
* INICIALIZACE MPI
  + vytvoření prostředí
  + definice, že program bude používat MPI knihovny
  + nemanipuluje explicitně s procesy
* IDENTIFIKACE PROSTŘEDÍ
  + kolik procesů se výpočtů účastní (MPI\_Comm\_size – vrací počet procesů sledující komunikátor MPI\_COMM\_WORLD)
  + jaká je moje identifikace (MIP\_Comm\_rank – vrací číslo mého procesu, čísluje se od nuly)
* PRÁCE SE ZPRÁVAMI
  + proces A posílá zprávu: operace send
  + proces B přijímá zprávu: operace receive
  + send i receive jsou blokující
  + Jak vhodně popsat data? Co to posíláme?
  + Jak specidfikovat proces? Komu to posíláme?
  + Jak přijímající pozná, že data patří jemu? Jsou to moje data?
  + Jak poznáme (úspěšné) dokončení těchto operací?
* KLASICKÝ PŘÍSTUP
  + data posíláme jako proud bytů (úkolem vysílajícího je správně nastavit přijímajícího, ten je musí správně rozpoznat)
  + jedinečné identifikátory
  + specifikace příznaku (tag)
  + synchronizace (přijímající může říct, od koho chce data )
  + send (buffer, len, destination, tag) – buffer – data, len – délka, destination – identifikace přijímajícího
  + recv(buffer, maxlen, source, tag, actlen) – maxlen – délka, actlen – skutečná délka, source – vysílající
  + NEDOSTATKY - nedostačná úroveň definice dat (nekopmatibilní zdroj a cíl, mnoho kopírování, vysoká zátěž programátora), globální tags (nezávislé knihovny mohou mít stejné označení jako závislé), příliš mnoho send/receive – broadcast
* ROZŠÍŘENÍ
  + procesy uspořádány do skupin (např. uzly knihovny)
  + každá zpráva má definovaný kontext (ne pouze příznak, např „moje knihovna“)
  + skupina a kontext definují komunikátor
  + skupina se všemi procesy MPI programu se nazývá MPI\_COMM\_WORLD
  + rank procesu je vždy uvnitř kontextu

# PVM

* předchůdce MPI, ale jinak orientovaný, důraz na to, jak sestavit virtuální paralelní počítač
* podpora programů kooperujících úloh
  + samostatné procesy na různých procesorech
  + komunikace přes zprávy
* přirozená podpora Task paralelismu (vhodný pro MPMD)
* C, C++, Fortran
* ŘÍZENÍ PROCESŮ
  + každý procesor má vlastní jedinečný TID, ten nese informaci o úloze a o jejím umístění na uzlu v PVM
  + TID je vydáván master PVM démonem
  + každé PVM má jeden master démon, je to potenciálně slabé místo
* POSÍLÁNÍ A PŘIJÍMÁNÍ ZPRÁV
  + za spolehlivý přenos odpovídají pvmd démoni
  + zprávu vždy přebírá lokální pvmd démon
  + z TID cílového procesu se určí umístění a zpráva se zašle pvmd démonovi
  + zajištění převodu dat mezi různými architekturami
* SKUPINOVÉ OPERACE
  + PVM podporuje tvorbu skupin procesů, proces se kdykoliv může ke skupině přidat nebo ji opustit
  + není nutná synchronizace

# MOORŮV ZÁKON

* Počet tranzistorů se na jednom čipu přibližně každých 18 měsíců zdvojnásobí.
* dříve bylo zvyšování frekvence zajištěno instrukčním paralelismem, out-of-order spouštěním instrukcí (zpracování instrukcí mimo programem stanovené pořadí tak, aby se co nejvíce využívalo výpočetního výkonu), vyrovnávacími pamětmi
* nyní vektorovými instrukcemi, zmnožováním jader

# GPU

* je datově paralelní, má vyšší koncentraci ALU, tím pádem vyšší teoretický výkon
* předdefinované funkce a programovatelné funkce (specifické grafické efekty např.)
* jsou vysoce výkonné, rozšiřitelné

# CPU vs. GPU

* jednotky jader vs. desítky multiprocesorů (ty se skládají z jader), GPU je více paralelní
* out of order (složité) vs. in order (jak přijde, tak se provede)
* MIMD, SIMD (jádra jsou na sobě nezávislá) pro krátké vektory vs. SIMT pro dlouhé vektory (není to vyloženě práce s vektory, ale s vlákny, stejná instrukce na více vláknech)
* velká cache vs. malá nebo vůbec žádná cache (přístup do paměti ve velkých blocích)
* GPU používá více tranzistorů pro výpočetní jednotky než pro cache a řízení běhu => vyšší výkon, méně univerzální algoritmy

# ARCHITEKTURA GPU

* koprocesor s dedikovanou (vlastní) pamětí
* asynchronní běh instrukcí
* připojen k systému přes PCI (sběrnicí, relativně pomalé)

# SROVNÁNÍ TEORETICKÉ RYCHLOSTI GPU A CPU

* GPU má cca 10x rychlejší aritmetiku
* GPU má cca 5x vyšší propustnost paměti
* důležité pro různé problematiky (chemie, simulace) – záleží nám, jestli se něco bude počítat rok nebo den, jestli video pojede 3fps nebo 30fps
* je-li uvedeno 100x i 1000x zrychlení, jde o produkční SW, který nemusí být perfektně optimalizován
* sériový CPU
  + ukazuje to širší možnosti
  + běh na jednom vlákně znamená až 16x zpomalení
  + absence vektorizace znamená až 4x zpomalení
  + oproti sériové implementaci můžeme kód paralelizací a vektorizací zrychlit
* vyšší odstup GPU
  + grafické funkce, SIMT pružnější než SIMD
* nižší odstup GPU
  + nedostatek paralelismu
  + vysoký overhead (zadání natlačíme, spustíme, stáhneme – je-li výpočet rychlý, tak většinu času tlačíme data přes sběrnici)
  + nevhodný algoritmus, chaos, divergence

# PARALELIZACE

* Sčítání vektorů (jen velké vektory, malé nemají smysl)
* Game of Life
* Redukce (např. součet všech prvků ve vektoru, může vypadat sekvenčně, ve skutečnosti

v log n krocích

* povodňové mapy

# DIVERGENCE KÓDU

* občas znemožní akceleraci kódu, musíme se zamyslet nad podstatou algoritmu
* nalezení divergujícího kódu může být snadné
* může i znemožnit akceleraci dobře paralelizovatelných algoritmů

# DIVERGENCE PŘÍSTUPU DO PAMĚTI

* přístup ve velkých blocích, jinak se snižuje propustnost
* složitě překonatelný problém (průchod obecného grafu)
* řídké matice

# LATENCE GPU

* propojení přes PCI-E
* kopírování vstupů a výstupů je pomalé (sběrnice)
* je třeba dostatečné množství aritmetiky na přenášená data
  + násobení matic – přenášení n^2 dat, počítání n^3 , přenášení zpět = nevadí, ztratí se
  + sčítání má přenášení i počítání n^2, zbytečné, pokud to není součást bloku

# CUDA

* architektura pro paralelní výpočty od NVIDIA
* poskytuje nový programovací model, který umožňuje efektivní implementaci obecných výpočtů pro GPU
* je možno použít ji s více programovacími jazyky

# C for CUDA – rozšíření jazyka C

* HIERARCHIE VLÁKEN
  + sdílená paměť, která je lokální pro daný multiprocesor
  + vlákna organizována do bloků, ty tvoří mřížku
  + dekompozice problému na podproblémy, ty mohou být prováděny nezávisle paralelně
  + podproblémy rozděleny do malých částí, ty mohou být prováděny kooperativně paralelně
  + dobrá škálovatelnost
* HIERARCHIE PAMĚTÍ
  + rozdílná viditelnost
  + rozdílný čas života
  + rozdílné rychlosti a chování
  + dobrá škálovatelnost