SSZ 2016 SWI

Systémové programování

SP

Část: Paralelní programování

Seznam otázek

1. Flynnova taxonomie – architektury a jejich urychlení.

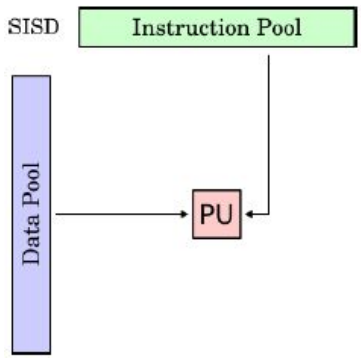
Status: nová otázka - zpracováno dle přednášek (1. Architektury) + hodně lehce doplněno z arcao (SP.pdf)

Čtyři základní klasifikace podle Flynna:

	Single Instruction	**Multiple Instruction**
Single data	SISD	MISD
Multiple data	SIMD	MIMD

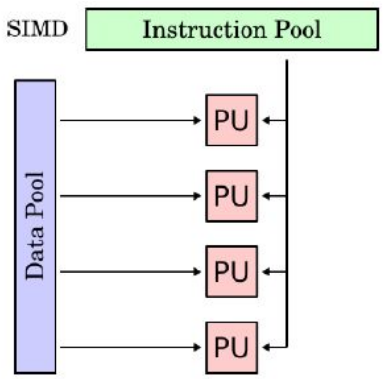
- počet konkurenčně prováděných instrukcí (tj. programů a někdy používanému počeštěnému označování např. SPMD, což ale není formálně správně právě podle Flynna)
- MIMD – dále se dělí na:
  - SPMD – Single Program, Multiple Data Streams
  - MPMD – Multiple Program, Multiple Data Streams

1.1. SISD (SPMD)



- Single Instruction, Single Data Stream
- Sekvenční výpočet, žádný paralelismus
- Např. i386 s MS-DOSem

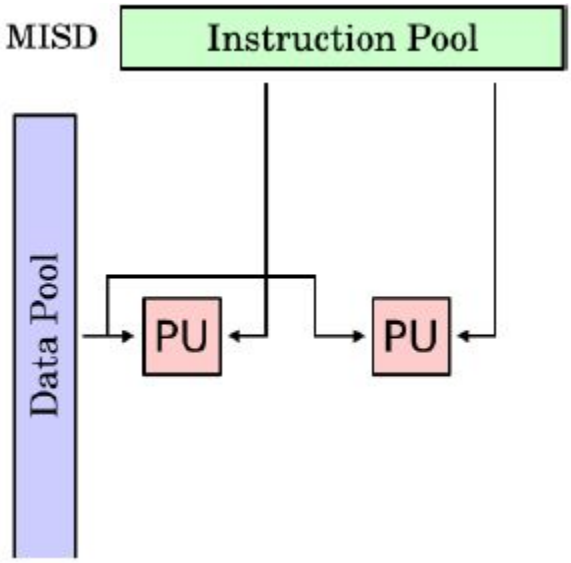
1.2. SIMD



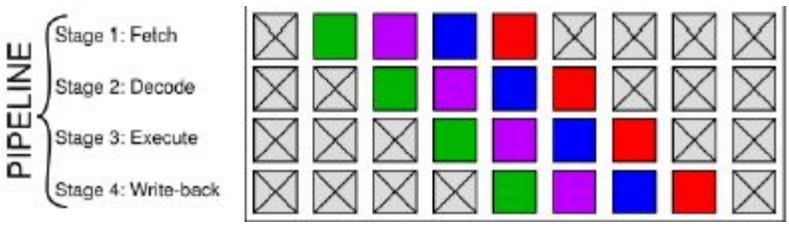
- Single Instruction, Multiple Data Streams
- Vektorový paralelní počítač (Vector Processor, Array Processor)
- Mohou provádět operace s vektory čísel na úrovni instrukcí strojového kódu
- př. MMX (Intel) – Matrix Math eXtension, SSE1-5 (Intel, AMD) – Streaming SIMD Extensions
- Paralelizaci tohoto typu využívá překladač

- Některé (např. GCC v4) umí tzv. auto-vektorizaci, kdy je část kódu rovnou převedena na vektorové instrukce
- Programátor může překladači pomoci správným zápisem kódu

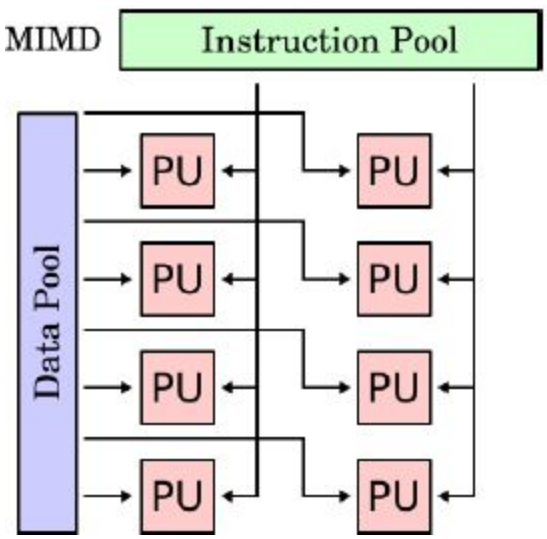
1.3. MISD (MPSD)



- Multiple Instruction (Program), Single Data Stream
- Používáno pro výpočty odolné proti poruchám (Fault Tolerant)
  - Několik různých systémů zpracovává ty samá data a musejí se shodnout na výsledku – např. řízení letu raketoplánu, letadla, atd.
- Používá se v tzv. Pipeline architektuře
  - několik procesů zpracovává data v jednom datovém proudu
  - analogií je montážní linka v továrně
  - Např. instrukční pipeline procesoru



1.4. MIMD



- Multiple Instruction, Multiple Data Streams
- Několik procesorů zároveň vykonává různé instrukce nad několika různými daty
- Procesory pracují asynchronně a nezávisle na sobě
- Procesory buď mají sdílenou paměť,
    - Programátorovi snáze srozumitelný přístup
    - O zajištění integrity dat se stará OS
- nebo distribuovanou
    - má lepší škálovatelnost
    - distribuovaný systém (např. cluster)
- MIMD programy lze odladit i na jednom počítači
    - Virtualizace jednoho procesoru pro sdílenou paměť
    - localhost a pro distribuovanou paměť
    - Samozřejmě, některé chyby takto mohou uniknout, protože se neprojeví díky jinému komunikačnímu zpoždění, nebo díky virtualizaci jednoho procesoru, kdy žádná dvě vlákna neběží současně

1.5. SPMD

- Single Program, Multiple Data Streams
- Několik procesorů autonomně vykonává jeden program nad různými daty
- Bod vykonávání programu nemusí být na všech procesorech stejný
- Označuje se též jako dekompozice dat
- Používá se ke zpracovávání velkých objemů dat (násobení matic, iterační numerické řešení parciálních diferenciálních rovnic)
    - *k* procesů běžících podle stejného programu zpracovává strukturně stejné, ale hodnotově různé části dat
    - např. násobení matic – každý prvek, řádek, či sloupec matice lze spočítat jedním procesem/vláknem
- Sleduje se čistě výkonnostní hledisko
- Předpokládá se, že každý z procesorů je schopen vykonávat ten samý program

1.6. MPMD

- Multiple Program, Multiple Data Streams
- Několik procesorů autonomně vykonává více než jeden program nad různými daty
- Např. farmer-worker, kdy jeden proces úkoluje ostatní
- Nemusí jít nutně pouze o urychlení výpočtu
    - Může jít o aplikaci, kdy se každý proces stará „o to svoje“ a zároveň spolupracuje s ostatními
    - Např. Cisco IOS – síť A s OSPF, síť B s EIGRP, border router vyplní směrovací tabulku podle obou
- Dist. Simulace – systém spolupracujících komponent
- provedenou dekompozici lze obvykle vyjádřit precedenčním grafem, ve kterém hrany symbolizují činnosti a uzly potřebu synchronizace
- Programový model MPMD lze implementovat jak na jednoprocesorovém počítači, tak na multiprocesorech různého typu.
- Lze například vytvořit paralelní program např. v jazyce Ada na jednoprocesorovém počítači, odladit ho a pak přenést beze změny kódu (vše zařídí překladač) na symetrický multiprocesor. Na něm budou jednotlivé procesy probíhat fyzicky paralelně a výpočet bude rychlejší

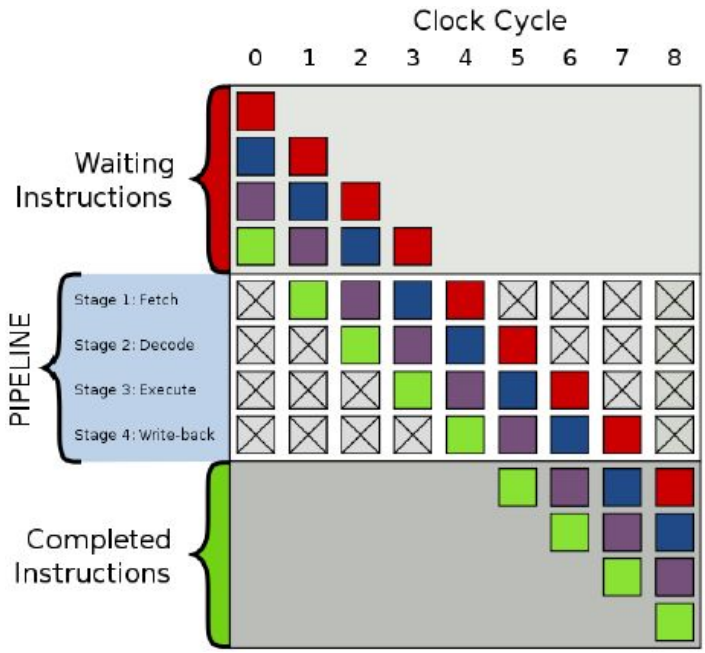
2. Paralelismus na úrovni instrukcí, predikce skoků, paměťová závislost, falešné sdílení a transakční paměť (Intel TSX) – jejich princip a význam pro urychlení sekvenčních a konkurenčních částí algoritmů.

Status: nová otázka a pěkně zkurvená - zpracováno z přednášek (2a. Urychlení vlákna)

2.1. Paralelismus na úrovni instrukcí

2.1.1. Pipeline

- Zpracování jedné instrukce lze rozdělit do několika fází jako je načtení, dekódování, vykonání, přístup do paměti
  - Počet fází závisí na konkrétním procesoru
  - Instrukce se skládá z několika mikroinstrukcí => mikroarchitektura, kterou programátor nevidí
- procesor bez pipeline vykonává jednu instrukci několik cyklů – např. 386 (byť ne že by neměla nic jako pipeline)
- pipeline procesoru (486) má sama o sobě možnost vykonávat instrukce paralelně – tj. nezávisle na sobě se snaží vykonávat různé mikroinstrukce
  - => **paralelismus na úrovni instrukcí**
  - ve špičkovém výkonu lze vykonat jednu instrukci za jeden cyklu



- moderní superskalární procesor (586+) používá další techniky, které mu umožní vykonat více než jednu instrukci za jeden cyklus
  - a pak se projeví znatelný rozdíl mezi kódem, který byl a který nebyl optimalizován
    - velkou část umí dobrý překladač, takže platí že, optimalizovat se vyplatí tak dlouho, dokud to není na úkor čitelnosti kódu
    - v kódu se pak programátor stará o to, aby překladač a procesor měli „volné ruce“ k provádění svých optimalizací
- je-li cílem vysoký výpočetní výkon, zapomeňte na JIT překladače – stojí další čas a výpočetní výkon, aby se nakonec možná dobrali ke kódu, který lze získat už statickým překladem
- finální optimalizace, které mají na svědomí celkový výkon, se totiž odehrávají v procesoru

2.1.2. Spekulativní multithreading

- při zvyšování počtu instrukcí za cyklus už může docházet k hazardům, takže se jejich vykonávání někdy uměle zpožďuje
  - stejně jako už dříve nešlo donekonečna zvyšovat pracovní frekvenci procesoru
- možným dalším směrem vývoje je analýza nezávislosti instrukcí procesorem, zda by nešly vykonávat ve vlastních vláknech (pokročilejší out-of-order execution)
  - tj. jakási hw autoparalelizace sériového kódu

2.2. Predikce skoků

- vykonává-li procesor instrukce paralelně, pak se při podmíněném větvení programu dostane do bodu, kdy:
  - buď počká na dokončení ostatních instrukcí, aby mohl vypočítat podmínku větvení
    - trvá to dlouho, ale nesplete se
  - anebo podmínku větvení odhadne
    - trvá to krátce, ale je tu penalizace, když se splete
- když procesor špatně odhadne podmínku větvení, pak musí zahodit všechny výpočty, které udělal na základě chybné predikce a vyprázdnit pipeline – tj. udělat rollback
  - je to drahé
- moderní procesor si udržuje historii (Branch Target Buffer - BTB), kolikrát na dané podmínce skočil, a podle toho dělá predikci, zda opět skočí
  - tím minimalizuje riziko penalizace
- Nemá-li procesor pro podmínku záznam v Branch Target Buffer, pak použije Static Branch Prediction Algorithm
  - `if (cond){<procesor předpokládá cond==true>};` - předpokládá, že podmínka bude vždy splněna
  - `do { <procesor předpokládá opakování smyčky> } while (cond);` - předpokládá alespoň jedno opakování cyklu
- Pokud procesor nezná příslušnou podmínku, tak se jí naučí
  - U podmínky řídící proměnné cyklu se tedy může splést úvodem
  - Při mnoha špatně navržených podmínkách, může docházet ke zdržení, když budou podmínky špatně vyhodnocované (špatně napsaný program)
- Programátor může podmínky navrhnout tak, aby vyhovovali Static Branch Prediction Algorithm
- Z tohoto vychází domněnka, že JVM může profilací kódu adaptivně přeuspořádat podmínky za běhu programu tak, aby program běžel co nejrychleji a programátor se s tím nemusel zatěžovat
- Skutečnost je ovšem taková, že s podmínkami se lze vypořádat elegantnějším způsobem, který už JVM nemůže zlepšit (JVM tedy dokáže překonat C++ překladač tehdy, jedná-li se o špatně napsaný program!)

2.2.1. Vyhodnocování podmínek

- Překladač obvykle vyhodnocuje komplexní podmínku ve zkrácené formě, je-li to možné
- Seřadit podmínky tak, aby podle pořadí vyhodnocování
  - Byly co nejmenší nároky na získání hodnot potřebných k vyhodnocení podmínky
    - proměnné v registru vs. proměnné v paměti
    - proměnné vs. volání funkcí
    - O() funkcí vracejících porovnávané hodnoty
  - Došlo co nejrychleji k vyřazení co největšího počtu možností
    - např. v databázi pacientů by podmínka pohlaví s ~50% úspěšností vyřadila nehledaný subjekt
- Opakování matka moudrosti
  - Sekvence AND končí s prvním false

- Sekvence OR končí s prvním true

2.2.2. Ternární operátor a CMOV

- Rodina instrukcí x86
  - Konkrétně u cmov jde o přiřazení, pokud je splněna podmínka
- Zejména v x86-64 kódu se používá u překladu ternárního operátoru – viz eliminace skoků
- Je třeba si uvědomit, kdy je rychlejší skok a kdy ternární operátor
- Pokud je podmínka snadno predikovatelná, např. řídící proměnná cyklu, pak se vyplatí dělat if
  - Procesor se snadno naučí správně odhadnout, kdy bude splněna a plně využije svého výkonu
- Pokud je ale podmínka závislá na vstupních datech, pak
  - Pokud jsou data predikovatelná, např. nějak uspořádaná, pak je lepší použít if
    - Např. budeme-li hledat v poli prvků hodnot nejmenší prvek
    - Např. budeme-li v uspořádaném poli bytů určovat, zda je byte větší než 0x80 nebo menší
  - Pokud jsou ale vstupní data taková, že daná podmínka bude ne/splněna s takovým rozložení pravděpodobnosti, které se blíží rovnoměrnému, pak je efektivnější použít ternární operátor – cmov
    - Např. budeme-li v neuspořádaném poli bytů určovat, zda je byte větší než 0x80 nebo menší
- Datová závislost ternárního operátoru/cmov
  - x <- cmov (x, y) - vzniká datová závislost, tj. zpoždění a je lepší použít if
  - z <- cmov (x, y) – nevzniká datová závislost a co použít závisí na pravděpodobnosti splnění podmínky
- U if lze pravděpodobnost specifikovat
  - GCC: __builtin_expect
  - MSVC: __assume
- Skoky lze sice eliminovat, ale pozor na to, že
  - může dojít k nárůstu kódu za skok, který lze snadno predikovat, což se projeví zpomalením
  - nebo může vzniknout datová závislost, která se opět projeví nežádoucím zpomalením
  - => **tj. skoky se neeliminují za každou cenu**

2.3. Eliminace skoků

Se skoky	S jedním skokem

`for i:=1 to 100 do`
`  begin`
`    if i mod 2=0 then`
`      a[i]:=0`
`    else a[i]:=1;`
`end;`
	`i:=100;`
	`repeat`
	`  a[i]:=0;`
	`  a[i-1]:=1;`
	`  dec(i, 2);`
	`until i=0;`

- Využití cmov – bytecode pro to nemá ekvivalent

Naivně (se skokem)	Lépe (bez skoku)

`int abs(int x) {`
`  if (x<0) x-=x;`
`    return x;`
`}`
	`mov ecx, [x] ; Load value.`
	`mov ebx, ecx ; Save value.`
	`neg ecx ; Negate value.`
	`cmovs ecx, ebx ; If negated value`
	`; is negative,`
	`; select value.`
	`mov [x], ecx ; Save labs result.`

- Při rekurzivním, podmíněném volání podprogramu, od hloubky 12 může dojít ke špatnému odhadu adresy, tj. vyhodnocení, kam se skočí při návratu z podprogramu.

Naivně rekurzivně	**Lépe (bez skoku)**

`long fac(long a) {`
`  if (a == 0) { return 1; }`
`  else {`
`  myp(a); //Can cause`
`  //returns to be`
`  //mispredicted`
`  return (a * fac(a - 1));`
`  }`
`}`
	`mov ecx, [x] ; Load value.`
	`mov ebx, ecx ; Save value.`
	`neg ecx ; Negate value.`
	`cmovs ecx, ebx ; If negated value`
	`; is negative,`
	`; select value.`
	`mov [x], ecx ; Save labs result.`

- Při vytváření switch/case záleží, jestli jsou možnosti sousední hodnoty, nebo jestli jsou mezi nimi mezery
  - Pokud jsou mezery, překladače switch převedou na sérii porovnávání
  - Snažit se eliminovat mezery
- Lze totiž sestavit tabulku ofsetů, kam se má skočit v konkrétních case a z podmínky lze potom vypočítat index to této tabulky
- Pokud to nejde, switch bude jedna velká série podmíněných skoků

S mezerami	**Bez mezer**

`switch (grade) {`
`  case 'A': ... break;`
`  case 'B': ... break;`
`  case 'C': ... break;`
`  case 'D': ... break;`
`  case 'F': ... break;`
`}`
	`switch (grade) {`
	`  case 'A': ... break;`
	`  case 'B': ... break;`
	`  case 'C': ... break;`
	`  case 'D': ... break;`
	`  default: ... break;`
	`}`

2.4. Mapování a inkrementování pointerů

- Než statické přiřazení nebo tohle

```
if (param = 1) or (param = 7) then proc1
else if param in [4..5] then proc2;
```

- je efektivnější udělat case/switch (záleží na hodnotách, a jak překladač case zrealizuje)

```
case param of
  1, 7: proc1;
  4..5: proc2;
end;
```

- anebo (zejména když by byl více než jeden parametr)

```
var procs:array[1..7] of TProc;
begin
  FillChar(procs, sizeof(procs), #0);
  procs[1]:=@proc1;
  procs[4]:=@proc2;
  procs[5]:=@proc2;
  procs[7]:=@proc1;
  ...
```

```
  //a namísto case
  procs[param];
end;
```

- Co bude rychlejší, to už záleží na konkrétním kódu (a co vlastně budou větve case/switch obsahovat)

2.5. Náročnost operací

- Díky optimalizacím mikroarchitektury se zdá, že provedení instrukce trvá kratší dobu, než tomu ve skutečnosti je, ale stále každá instrukce trvá 1 až několik cyklů, jako u 386
- Proto je třeba vyhýbat se drahým/dlouhotrvajícím instrukcím
- Má-li procesor vykonat drahou instrukci, musí zpozdit ostatní instrukce v pipeline
- proto se např. opakované dělení provádí násobením inverzním číslem
  - ix:=1.0/x; y:=a*ix; z:=b*ix;
  - U integerů, m:=i div (j*k) namísto m:=(i div j)div k
- Dělení hard-coded číslem se už ale nechává na překladači, který pro to má vlastní triky

2.6. Vektorizace a Loop Unrolling

- Pro procesor s vektorovými instrukcemi lze loop unrolling chápat jako nápovědu pro překladač s auto-vektorizací (v nejhorším dojde k většímu využití pipelines procesoru)
- Použití i++ v a[...] by mohlo vnést závislost, tj. zhoršit výkon při využití pipelines

Naivně	**Lépe**

`double a[100], sum;`
`int i;`
 	
`sum = 0.0f;`
 	
`for (i = 0;`
`  i < 100; i++) {`
`  sum += a[i];`
`}`
`//Překladač to může,`
`//ale i také nemusí`
`//správně pochopit.`
 	
`//Loop-unrolling`
`//také snižuje počet`
`//porovnávání, tj. i`
`//případných skoků.`
	`double a[100], sum;`
	`double sum1, sum2, sum3, sum4;`
	`int i;`
	
	`sum1 = 0.0f;`
	`sum2 = 0.0f;`
	`sum3 = 0.0f;`
	`sum4 = 0.0f;`
	
	`for (i = 0; i < 100; i + 4) {`
	`  sum1 += a[i];`
	`  sum2 += a[i+1];`
	`  sum3 += a[i+2];`
	`  sum4 += a[i+3];`
	`}`
	
	`sum = (sum4 + sum3) +`
	`(sum1 + sum2);`

2.7. Počet iterací při Loop Unrolling

- Vektorové operace mají operandy o fixní velikosti
- Překladač tak potřebuje vědět, kolik potenciálně vektorizovatelných operací jde po sobě (aby věděl, jestli může použít vektorové instrukce)

Tohle lze vektorizovat	**Tohle nelze vektorizovat**	**Tohle lze, pokud známe data**

`for (i = 0; i < 100; i++)`
`{`
`  src[i] += dst[i];`
`}`
	`while (*dst) {`
	`  *src += *dst;`
	`  src++;`
	`  dst++;`
	`}`
		`while (*dst) {`
		`  *src += *dst;`
		`  *(src+1) += *(dst+1);`
		`  *(src+2) += *(dst+2);`
		`  *(src+3) += *(dst+3);`
		`  src += 4;`
		`  dst += 4;`
		`}`

2.8. Out-of-Order Execution & Registry Renaming

Tohle nelze paralelizovat (stejný registr)	Tohle ano, použili jsme jiný registr

`R1 = mem[addr1]`
`R1 = R1 + 4`
`mem[addr1] = R1`
 	
`R1 = mem[addr2]`
`R1 = R1 + 8`
`mem[addr2] = R1`
	`R1 = mem[addr1]`
	`R1 = R1 + 4`
	`mem[addr1] = R1`
	
	`R2 = mem[addr2]`
	`R2 = R2 + 8`
	`[addr2] = R2`

- Ve specifikaci architektury registrového procesoru jako je x86 existuje konečný počet pojmenovaných registrů
- Překladač se sám snaží využít registry tak, aby se co nejvíce instrukcí dalo provést paralelně
- Procesor se snaží o další dvě optimalizace
  - Out-of-order execution – procesor sice načítá instrukce v pořadí daném překladačem, ale prochází je, a určuje jim nové pořadí, ve kterém je vykoná
    - Snaží se detekovat jejich vzájemné závislosti tak, aby je mohl čekání na dokončení jedné instrukce překrýt vykonáváním jiné (nezávislé instrukce)
  - Registry renaming – procesor má více registrů, než kolik jich je pojmenovaných na úrovni architektury
    - V případě, kdy překladači dojdou volné registry, procesor je stále schopný paralelně vykonávat instrukce, které zdánlivě pracují se stejným registrem – viz případ vlevo
    - Dalším pozitivem registry renaming je i to, že se může vyplatit loop-unrolling i tehdy, když se nejedná o kód převeditelný do vektorových instrukcí
    - Nicméně bude stále platit, že čím méně proměnných ve funkci, tím více proměnných dokáže překladač uchovat pouze v registrech (Což zrychlí běh aplikace a vyplatí se „roztrhat“ např. náročnější tělo cyklu na několik cyklů)

2.9. Paměťová závislost

- x <- cmov (x, y) - datová závislost
- paměťová závislost - instrukce load (from memory) musí čekat na instrukci store (to memory)
  - Přičemž přístup k paměti se může stát úzkým hrdlem
- Řešením je eliminace nepotřebných sekvencí Store-and-Load používání dočasných proměnných, které mohou být realizovány pomocí registru
  - Programátor by měl pomoci překladači jejím použitím – namísto co nejkratšího zápisu (viz př.)

Naivně	Lépe

`double x[VECLEN], y[VECLEN];`
`unsigned int k;`
 	
`for (k = 1; k < VECLEN; k++) {`
`  x[k] = x[k-1] + y[k];`
`}`
	`double x[VECLEN], y[VECLEN];`
	`unsigned int k;`
	`double t;`
	
	`t = x[0];`
	`for (k = 1; k < VECLEN; k++) {`
	`  t = t + y[k];`
	`  x[k] = t;`
	`}`

2.10. Falešná závislost

- Registrů na úrovni mikroarchitektury je konečný počet
- A nakonec se musí výsledek zapsat někam, kde je viditelný z pohledu architektury
- Příkladem, na x86 existují čtyři způsoby, jak zapsat do registru nulu
    - registr eax se běžně používá pro předávání návratových hodnot a parametrů funkcí

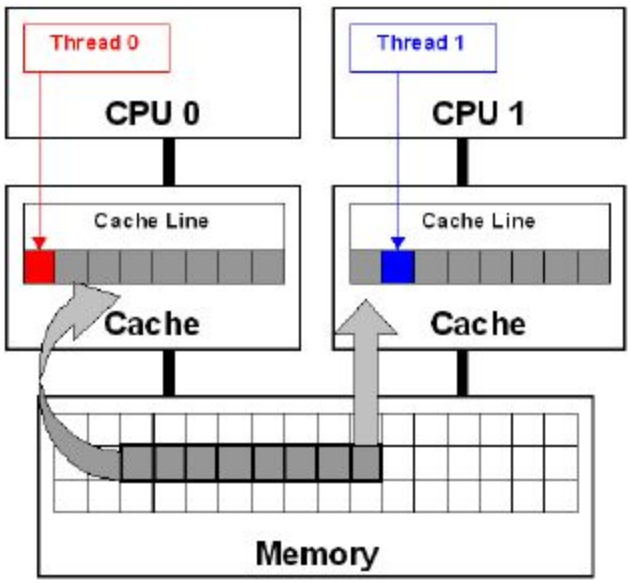
Instrukce	**Strojový kód**

`mov eax,`
`and eax,`
`sub eax, eax`
`xor eax, eax`
	`b8 00 00 00 00`
	`83 e0 00`
	`29 c0`
	`31 c0`

- Krátký kód je dobrý, protože se ho pak více vejde do cache => vyřadíme mov a and
- sub a xor představují datovou závislost
    - jenže u xor si je procesor vědom, že jde o falešnou závislost (a překladač to ví)

2.11. Falešné sdílení

- False Sharing
- Každý procesor má svou vlastní cache, ale paměťový systém musí zajistit tzv. cache-coherence – konzistenci dat
- False sharing nastane tehdy, když různé procesory (a každém z nich běžící jiné vlákno) modifikují různé proměnné, které ale sdílejí stejnou cache-line
    - Ačkoliv se může jednat o proměnné, které nejsou sdílené mezi vlákny, přesto je daná cache-line zneplatněna
- Následně je vynucena její aktualizace => a dojde ke zpomalení
- Paradoxem je, že sériovému programu se to nestane a může tak být i rychlejší, než špatně napsaný paralelní program.



Flašné sdílení	Odstranění flašného sdílení

`double sum=0.0, sum_local[NUM_THREADS];`
`#pragma omp parallel num_threads(NUM_THREADS)`
`{`
`  int me = omp_get_thread_num();`
`  sum_local[me] = 0.0;`
`  #pragma omp for`
`  for (i = 0; i < N; i++)`
`  sum_local[me] += x[i] * y[i];`
`  #pragma omp atomic`
`  sum += sum_local[me];`
`}`
	`void threadFunc(void *param) {`
	`  ThreadParams *p = (ThreadParams*)`
	`  param;`
	`  auto local = p->variable;`
	
	`  for(local=p->start; local<p->end;`
	`  local++) {`
	`  // Function computation`
	`  }`
	`  // Update only once`
	`  p->variable = local;`
	`}`

- U sum_local může dojít k false sharing, protože pole může být tak malé, aby se celé vešlo do jedné cache-line
- Možným, ale špatným, řešením by bylo zvýšit velikost proměnné na tolik bytů, aby se do cache-line vešla vždy jen jedna proměnná (to vyžaduje znalost cílového procesoru)
- Velikost cache-line (tj. velikosti bloku dat) může být 32, 64, ale i 128
- Lepším řešením je, aby si každý thread vytvořil svou pracovní kopii data, která pak zapíšou pouze jedno, až na závěr svého výpočtu

2.12. Transakční paměť

- Mějme datovou strukturu naplněnou např. 1024 prvky, kterou chceme modifikovat z více vláken
- Nejjednodušší je použít jeden zámek, protože vzniká čitelný kód a je nejmenší riziko chyby synchronizace
- Jenomže bude-li chtít např. první vlákno modifikovat desátý prvek a současně druhé vlákno 600. prvek, jedno z nich bude muset počkat, ačkoliv nehrozí riziko poškození dat
- Softwarovým řešením je použít více zámků (rozdělit prvky do několika regionů, každý s vlastním zámkem), takže se sníží pravděpodobnost, že na sebe budou dvě vlákna čekat, budou-li modifikovat nezávislé prvky
    - Riziko ovšem nelze eliminovat úplně a navíc se tím zvyšuje složitost synchronizačního kódu => zvyšuje se riziko chyby
- Hardwarovým řešením je transakční paměť
    - Procesor ji nedělá automaticky, ale poskytuje speciální instrukce
- Programátor pak buď využije knihovny svého jazyka, které pracují s transakční pamětí např. gnu libc, Intel TBB nebo v jazycích jako je C++ ji může použít sám
- Cílem transakční paměti je poskytnou výhody jednoho zámku při rychlosti sw řešení s několika zámky
- Programátor v kódu označí instrukci, kterou začíná kritická sekce
    - Touto instrukcí zároveň označí část kódu, kam procesor skočí v případě, že se transakci nepodařilo uskutečnit – tzv. fallback path
- Procesor zjistí začátek kritické sekce a od té chvíle jsou veškeré zápisy do paměti lokální, dokud:
    - Není transakce úspěšně dokončena a pak se provede commit
    - Anebo dojde ke konkurenčnímu zápisu do hlídané oblasti paměti
        - Pak procesor skočí na první instrukci fallback path
        - Velikost hlídané paměti je záležitostí procesoru, např. 32kB L1 cache u Intel Haswell
- x86-64 umí dva režimy
    - HLE (hardware lock ellision) – kompatibilní se starým kódem, znovupoužití prefixů instrukcí
    - RTM (restricted transaction mode) – nové instrukce

2.12.1. HLE - modifikace spinlocku

```
      mov rdx, qword (-1);hodnota zamčeno
      xor rax, rax ; hodnota odemčeno
spin: xacquire lock cmpxchg8b [zámek], rdx
      jz spin ;if (rax == [zámek]) ZF=1
      ;zápis do kritické sekce dle potřeby
      xrelease mov qword ptr[zámek], 0
```

- fallback path začíná instrukcí cmpxchg – ta díky xacquire napoprvé proběhne, jako kdyby nebylo zamčeno

2.12.2. RTM - modifikace spinlocku

```
     mov rdx, qword (-1);hodnota zamčeno
     xor rax, rax ; hodnota odemčeno
     xbegin spin
     jmp locked
spin: lock cmpxchg8b [zámek], rdx
     jz spin ;if (rax == [zámek]) ZF=1
locked:
     ;zápis do kritické sekce dle potřeby
     mov qword ptr[lock], 0 ; odemknout
     xend
```

- Instrukce xbegin říká, kde začíná fallback path
- Na rozdíl od HLE jsme ušetřili lock cmpxchg8b
- Existuje i instrukce xabort
- Registr eax je při skoku do spin nastaven na hodnotu kódu, proč transakce neprošla

3. Paralelizace výpočtu součtů prefixů - charakteristika a řešení.

Status: nová otázka, pěknej ojeb - zpracováno dle přednášky (3b. Shared SPMD) a lehce doplněno z arcao (SP.pdf)

- Při analýze možné paralelizace cyklu je třeba určit jaké typy proměnných v cyklu jsou:
  - lokální - inicializovány uvnitř smyčky `//a`
  - sdílené - hodnoty se přenáší mezi iteracemi
    - nezávislé – když se pouze čtou `//items (kromě v pravo dole)`
    - závislé
      - redukční – nejprve čtena, následně zapsána – ve stejné iteraci `//sum`
      - uzamykané – čtena i zapisována v několika iteracích nebo několikrát v jedné - pokud by se iterace neprováděly sekvenčně, výsledek by byl stále OK `//min`
      - uspořádané – správný výsledek, jen když jsou iterace provedeny ve správném pořadí - již nelze urychlit tak, že vlákna současně vykonají operace nad svou částí pole (a pak jedno z nich zpracuje mezivýsledky) `//items vpravo dole`

```
sum:=0;
for i:=0 to High(Items) do
  begin
    a:=random(Items[i]);
    sum:=sum+a;
  end;
```

```
min:=Items[0];
for i:=1 to High(Items) do
  if min<Items[i] then
    min:=Items[i];
```

```
for i:=1 to High(Items) do
  Items[i]:=Items[i] + Items[i-1];
```

- ***Paralelizaci součtu pole*** lze provést snadno, protože se v cyklu vyskytuje nejvýše redukční proměnná, kterou stačí jen ošetřit mutexem.

- ***Paralelizace součtu prefixů*** (výsledek pole součtů) již nelze provést jednoduchým rozdělením iterací vláknům, protože se v cyklu nachází uspořádaná proměnná.

3.1. Paralelizace výpočtu součtů prefixů

- Problém popisuje následující kód, kde máme uspořádanou proměnnou a vypadá jako beznadějně sekvenční

```
for i:=1 to High(Items) do
  Items[i]:=Items[i] + Items[i-1];
```

```
[a0, a1, … , an-1] //matematicky
[(a0), (a0+a1), …, (a0+a1+…+an-1)]
```

```
[4, 5, 3, 1, 6] //pro lidi
[4, 9, 12, 13, 19]
```

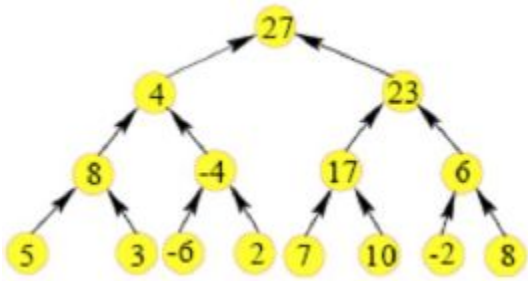
- Princip paralelizace výpočtu součtů prefixů se dá aplikovat i na některé další sekvenční výpočty, např.
  - Třídění
  - Lexikální analýza
  - Histogramy
  - Teorie grafů
  - Práce s řetězci

3.1.1. Postup

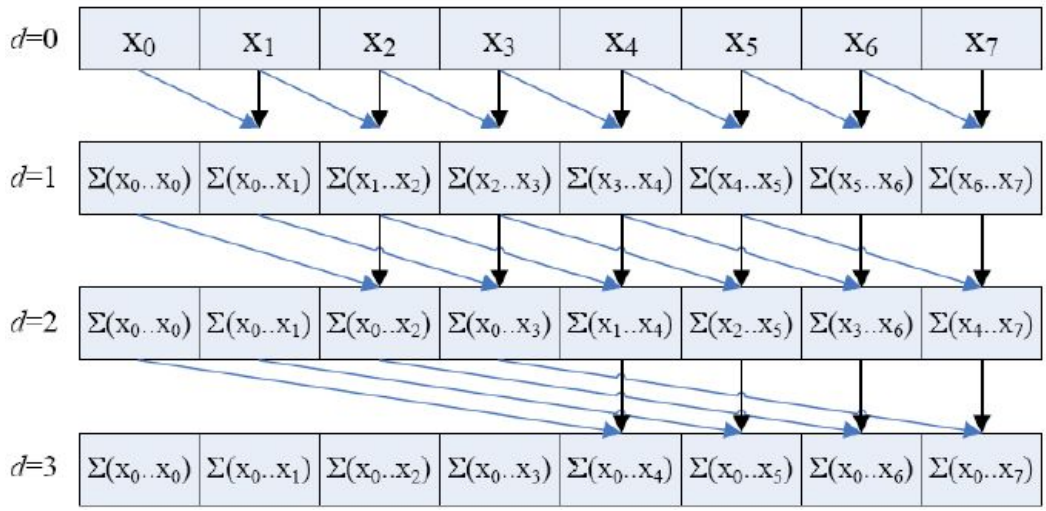
1. První krok - zavést kopii pole
   a. Uspořádaná proměnná nám zabraňuje zapisovat do pole v jiném, než určeném pořadí => **zavedeme ještě jednu kopii pole**
   b. S `i-1` už přistupujeme do jiného pole, do kterého se v cyklu nezapisuje => **zrušeno uspořádání**

```
Move(Temp, Items, Length(Items)); //kopirovani puvodniho pole do Temp
for i:=1 to High(Items) do
  Items[i]:= Temp[i-1] + Items[i]; //samozřejmě, že „něco" chybí k úplnosti
```

2. Krok druhý - výpočet n-tého prvku rozložit do jiné posloupnosti
    a. V sekvenční verzi jsme s každým součtem získali jeden prefix
    b. Výpočet `n-tého` prefixu se skládá z `n-1` součtů
    c. Protože už jsme se ale zbavili uspořádanosti, můžeme **součet `n-tého` prvku rozložit do jiné posloupnosti součtů** než v sekvenční verzi
    d. Pokud bychom měli k dispozici dva procesory, uvedený strom ukazuje, že výpočet posledního prvku můžeme paralelizovat



3. Krok třetí - mezisoučty počítat v krocích
    a. Jestliže lze paralelizovat výpočet posledního prvku, pak lze paralelizovat i výpočet ostatních prvků
    b. **Mezisoučty vznikají postupně v krocích – `d`**
    c. Významná část mezisoučtů je využita jako mezihodnota dva dalších mezisoučtů



4. Čtvrtý krok - určit každému vláknu rozmezí (hranice odkud kam)
    a. Programový kód pro jedno vlákno
    b. **Hranice dat každého vlákna je daná rozmezím `Offset` a `Size`**

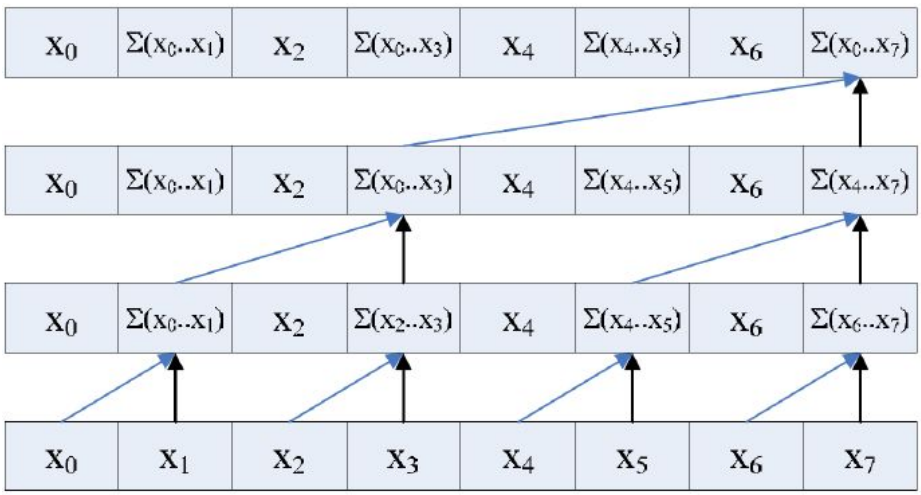
```
step:=1;
while step<High(Items) do
  begin
    Move(Temp^, @Items[Offset], Size);

    Barrier;
    for i:=max(step, Offset) to Offset+Size do
      Items[i]:= Temp[i-step] + Items[i];

    Barrier;
    step:=step shl 1; //*2
  end;
```

- S rostoucím krokem, for cyklus u některých vláken nebude probíhat
  - `Step` by mohla být sdílená proměnná, kterou by zapisovalo např. výhradně první vlákno
  - Pak poslední bariéra v poslední iteraci není třeba
- Uspořádanost
  - Zbavili jsme se původní uspořádanosti
  - Nicméně, původní uspořádanou proměnnou jsme jenom nahradili jinou uspořádanou proměnnou
    - `Step, d`
  - V paralelizované podobě
    - Hlavní cyklus řízený uspořádanou proměnnou má nyní méně iterací
    - V každé iteraci se udělá více práce, kterou už lze paralelizovat
    - V původní verzi byl pouze hlavní cyklus
- Urychlení
  - Sekvenčně *O(n)*
  - Paralelní verze vykonaná pouze jedním vláknem ~ $O(0{,}5\ n\ log_2 n)$
  - Část celého pole hodnot, se kterými se bude počítat, nám "utíká" doprava
    - Další urychlení dynamicky přerozdělit meze, ve kterých jednotlivá vlákna počítají tak, aby všechny počítaly stejný objem práce
  - Paralelizace má smysl tehdy, běží-li všechna vlákna paralelně
    - Každá dvě vlákna, která se střídají o stejný procesor znamenají pouze nárůst instrukcí k vykonání, bez urychlení
  - Splníme-li podmínky vhodné paralelizace, pak s *m* procesory dostáváme ~ $O((0{,}5\ n\ log_2 n)/m)$
- Algoritmus je evidentně závislý na počtu procesorů a objemu zpracovávaných dat
- Hodí se spíše k HW akceleraci

3.1.2. Optimalizace

- Další úprava algoritmu, kterou se s paralelní verzí dostaneme na složitost *O(n)*
  - Tj. stejně jako u sekvenční verze, jenomže už ji můžeme spustit paralelně)
    - Pak *O(n/m)*
  - Navíc to celé provedeme in-place, stejně jako sekv.

1. Krok první
   a. Redukční fáze – od zdola nahoru



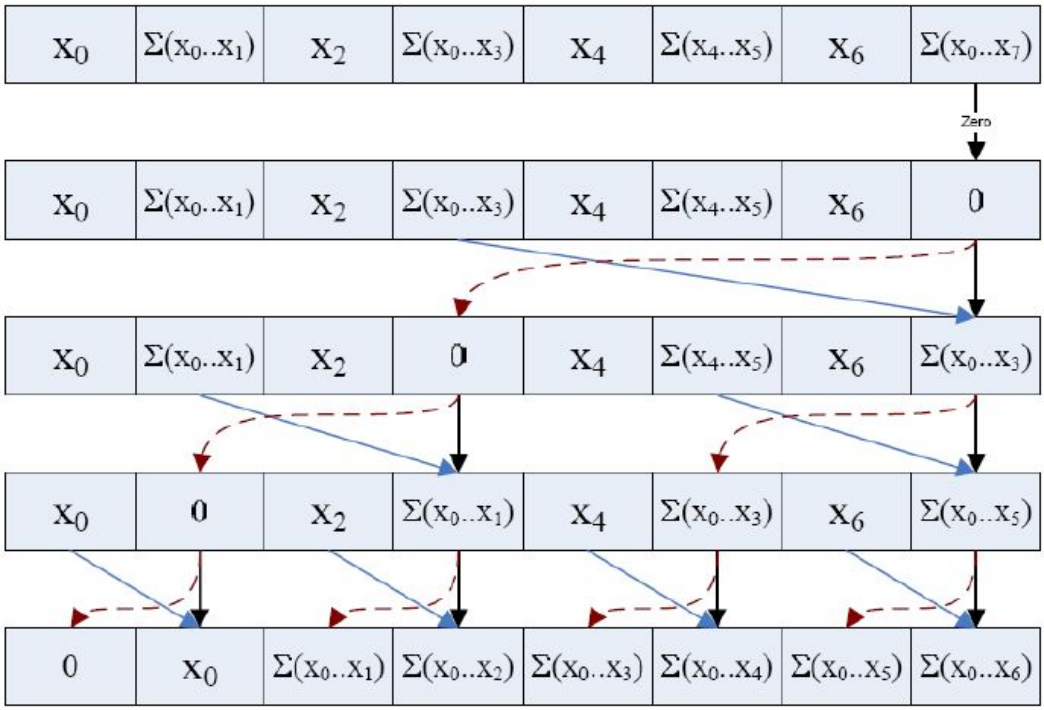

```
d:=1;
while d<High(Items) do
  begin
```

```
    for i:=max(d, Offset) to Offset+Size by d do
      Items[i]:= Items[i-d] + Items[i];

    Barrier;
    d:=d shl 1;
  end;
```

2. Krok druhý
    a. Záloha součtu posledního prvku
    b. Vynulování posledního prvku pole (i když to ve skutečnosti dělat nemusíme)

```
SavedSum:=Items[High(Items)];
Items[High(Items)]:=0;
```

3. Krok třetí
    a. Fáze smetení (down sweep), od shora dolů




```
d:=d shr 2; //d zůstala jeho hodnota
while d>1 do
  begin
    for i:=max(d, Offset) to Offset+Size by d do
      begin
        temp:=Items[i];
        Items[i]:=Items[i+d];
        Items[i+d]:=Items[i+d]+temp;
      end;
  Barier;
  d:=d shr 1;
end;
```

4. Krok čtvrtý
    a. Posunutí všech součtů o jeden doleva
    b. Obnovení součtu posledního prvku

```
Temp:=Items[Offset];
for i:=Offset to Offset+Size-1 do
  Items[i]:=Items[i+1];
Barrier;
if Offset>0 then //kromě prvního úseku
  Items[Offset-1]:=Temp; //zápis do cizích dat
if Offset+Size=High(Items) then //poslední úsek
  Items[Offset+Size]:=SavedSum;
```

- Závěrem o paralelních součtech prefixů – alias „scan“, má dvě varianty:
  - Inclusive – první verze
  - Exclusive – nula z optimalizace na začátku, neobsahuje finální součet
- Podporováno např. v MPI (později) fcí MPI_Scan
- Nemusí se jenom sčítat (hledání maxima, minima apod.)
- o Aneb, i některé úlohy s „defaultně“ uspořádanými proměnnými lze paralelizovat – jen vymyslet jak:-)

4. Spurious wakeup – charakteristika a ošetření.

Status: nová otázka - zpracováno dle přednášek (4b Java)

- Ačkoliv má být JVM vysokoúrovňovou abstrakcí od OS, např. Spurious Wakeup je takový scénář, kdy před OS „nelze utéct“
- **Jednodušše řečeno jde o to, že spící vlákno se může probudit, aniž by byla splněna programátorem zapsaná podmínka pro jeho probuzení**
  - Ještě jednodušeji: když zavoláte wait(), tak se vlákno může probudit, aniž by někdo zavolal notify()
- Bariéru špatně a správně

```
class Bariera {
  private int pocetVlaken;
  private int citac;
  private boolean muzeVstat;

  Bariera (int pocetVlaken) {
    this.citac = 0;
    this.muzeVstat = false;
    this.pocetVlaken = pocetVlaken;
  }

  public synchronized void synchronizuj_spatne() throws InterruptedException {
    citac++;
    if (citac < pocetVlaken) { wait(); }
    else { citac = 0; notifyAll();}
    //Když se jedno vlákno vzbudí předčasně, tak //také bariéru opustí předčasně =>
    //je třeba zavést další podmínky pro čekání vlákna.
  }

  //takhle to řeší tzv. cyklická bariéra
  public synchronized void synchronizuj_dobre() throws InterruptedException {
    while (muzeVstat == true) {wait();}
    citac++;

    if (citac == pocetVlaken) {
      muzeVstat = true;
      notifyAll();
    }

    while (muzeVstat == false) {wait();}
    citac--;

    if (citac == 0) {
      muzeVstat = false;
      notifyAll();
    }
  }
} //class Bariera
```

- Uvedený kód demonstruje SpuriousWakeup
  - Jinak se dá použí třída `CyclicBarrier`
- Linux
  - Mapuje-li JVM svá vlákna 1:1 na vlákna `pthreads`, pak wait odpovídá funkci `pthread_cond_wait()`, která je implementovaná pomocí futexu – tj. pomocí systémového volání
  - Jenomže, blokující systémové volání je „násilně“ přerušeno, `EINTR`, obdrží-li proces signál
  - `pthread_cond_wait` nelze restartovat, protože za tu dobu, co byl proces „vzhůru“, mohl nastat skutečný wakeup a ten by se mohl restartování „promeškat“

        - => spurious wakeup, a ať si to hlídá programátor, protože jinak by to bylo příliš komplikované na úrovni jádra
- Futex == Fast User Space Mutex
    - Fronta čekajících procesů je na úrovni jádra
    - Ale počítadlo je v uživatelském adresovém prostoru
    - Vlákna se jeho použitím pokoušejí vyhnout drahým systémovým voláním
    - Ve Windows se takhle chovají slim locks a kritická sekce – viz spincount
- Vedle Spurious Wakeup ještě existuje tzv. Stolen Wakeup
    - Jiné vlákno se spustí dříve než vzbuzené vlákno

5. Amdahlův a Gustafsonův zákon, Karp-Flattova metrika.

Status: potřebuje revizi, mírně se liší od původní otázky - zpracováno dle přednášky (1 Architektury)

5.1. Výkonnost

- **Složitost**
  - Complexity, worst-case complexity
  - Maximální doba výpočtu algoritmu pro všechny možné kombinace vstupních dat
  - *O(n) = n*
    - složitost sekvenčního algoritmu, která je lineárně závislá na *n* počtu prvků
    - např. součet čísel
  - Paralelní součet na p = n/2 procesorech má složitost O(log n)
    - Logaritmus s libovolným základem
- **Cena (paralelního algoritmu)**
  - Cost
  - Složitost vzhledem k počtu použitých procesorů
    - Kolik celkového strojového času, tj. všech použitých procesorů, výpočet spotřebuje
    - Např. *(log n)*n/2*
  - Je úměrná celkovému strojovému času všech použitých procesorů
- **Urychlení**
  - Speedup
  - Existuje několik způsobů, jak ho vyjádřit, následující je nejrozšířenější/nejznámější způsob
  - Poměr doby výpočtu referenčního algoritmu a porovnávaného algoritmu
    - Např. nejlepšího známého sekvenčního algoritmu a paralelního algoritmu na témže (paralelním) počítači
      - Lze ovšem porovnat i urychlení referenčního sekvenčního algoritmu oproti provedené optimalizaci nebo porovnat dva různé paralelní algoritmy
  - *S(p) = T(1) / T(p)*
    - doba běhu na *1* procesoru ku době běhu na *p* procesorech
    - >1 znamená urychlení
  - Perfektní urychlení - poměr je přesně roven počtu procesorů (těžko ho dosáhnete)
- **Účinnost**
  - Efficiency
  - Urychlení dělené počtem procesorů
  - *E = S / p*
  - Uvažujeme urychlení proti sekvenčnímu algoritmu
  - Sekvenční výpočet trvá 10s, paralelní algoritmus na 4 procesorech trvá 5s
  - Urychlení: 2
  - Účinnost: 0,5
- **Anomální urychlení**
  - Distribuce rozsáhlých dat u distribuované aplikace může omezit nutnost stránkovat RAM
  - S dostatečně rychlými komunikačními kanály pak dojde k rychlejšímu vykonávání programového kódu, protože odpadá čekání na zpomalující I/O operace provázející stránkování, včetně obsluhy příslušných přerušení
  - Např. paralelizované vyhledávací algoritmy mohou mít větší než lineární urychlení
  - Lze rychleji upřesnit výběrová kritéria a zparalelizovaný algoritmus má pak urychlení větší než lineární
- **Superlineární urychlení**

- v některých případech je skutečně možné, aby S vyšlo lépe než je perfektní urychlení
  - V těchto případech se nejedná o chybu ve výpočtu
  - Zejména, porovnáváme-li oproti nejlepšímu sekvenčnímu algoritmu
- **Cache procesoru -** pokud se program vejde do cache, dojde k superlineárnímu urychlení
  - Mnohem častější cache-hit než je obvyklé
  - Výpočet je pak prováděn mnohem rychleji, než když se programový kód dostává k procesoru z pomalejší paměti
  - Záleží na konkrétním programovém kódu, zda se ho bude dostatečné množství nalézat právě v lokálních cache jednotlivých procesorů
- **Analogicky datová cache -** pokud se data vejdou do cache, dojde k superlineárnímu urychlení

- superlinární a anomální urychlení (dle mě je to stejný) je možné dosáhnout pouze u Amdahla (alespoň co říkal tomáš, páč Gustafson s tím už má počítat)

5.2. Amdahlův zákon

- Nelze dosáhnout perfektního urychlení, protože vždy bude nějaká část výpočtu provedena sekvenčně
  - *G, H* – čas strávený sekvenčně provedeným výpočtem
  - *G* - čas strávený vykonáváním neparalelizovatelného kódu, tj. sekvenčně (Unavoidably Serial)
  - *H* - čas strávený vykonáváním paralelizovaného kódu, ale sekvenčně (Serialized-Parallel)
  - $E(p) = G + H/p$
  - $S(p) = (G+H)/(G+H/p)$
- Pro velké p platí $S(p) = 1+H/G$
  - To by znamenalo, že dosažení perfektního urychlení je možné, pokud se můžeme vyhnout kódu, který nelze paralelizovat
  - Má vliv na *H*
  - Ale – čím větší počet procesorů, tím např. větší režie jejich komunikace => takže přece jenom nepůjde + režie OS (plánování, I/O, atd.) se o to také postará

$S \leq \frac{1}{f+\frac{1-f}{p}} \leq \frac{1}{f}$ kde f je 0<f<1	$max\ speedup \leq \frac{p}{1+f\times(p-1)}$

5.3. Gustafsonův zákon

- $S(P) = P - \alpha \cdot (P-1)$ //tenhle je nejdůležitější
  - *S* – urychlení
  - *P* – počet procesorů
  - α - část procesu, kterou nelze paralelizovat
- Co když budeme mít k dispozici hodně procesorů s distribuovanou pamětí a výpočet složitý tak, že sériově prováděná část bude zanedbatelná?
  - Amdahl si s tím neporadí…
- Nerozdílíme výpočet podle paralelizovatelnosti kódu, ale podle podílů času – sériově vs. paralelně
- $S = \frac{a(n)+p\cdot b(n)}{a(n)+b(n)}$
- $a(n)+b(n) = 1$
  - *1* – čas výpočtu na p procesorech
  - $a(n)$ – sériově
  - $b(n)$ – paralelně
- $a(n) + p \cdot b(n)$ - čas výpočtu na jednom procesoru
- Ideální urychlení
  - Do S se dosadí a(n)+b(n) = 1

- $p \to P$ => $S(P) = P - \alpha \cdot (P-1)$
- A lze vyhodnotit pro *a(n) aka* $\alpha = \lim_{x \to 0} x$

5.4. Karp-Flattova metrika

- $e = \frac{\frac{1}{\psi} - \frac{1}{p}}{1 - \frac{1}{p}}$ //tohle je nejdůležitější
- e – metrika, podíl, kolik kódu se provedlo sériově
- p – počet procesorů
- ψ - urychlení na p procesorech (určí se z naměřených časů)
- $T(p) = Ts + \frac{Tp}{p}$ //odsud dolu je pravděpodobně popis jak vznikl ten vzorec nahoře
- Čas výpočtu na p procesorech
  - Ts – čas po který kód běžel sériově
  - Tp – čas po který kód běžel paralelně
- $T(1) = Ts+Tp$
- $e = \frac{Ts}{T(1)}$
  - Dosadí se do T(p)
- $\psi = \frac{T(1)}{T(p)}$
- $\frac{1}{\psi} = e + \frac{1-e}{p}$
- A úpravou dostaneme e

6. Programové prostředky pro multithreading – POSIX, WinAPI, C++11.

Status: nová otázka (bombová) - zpracováno z přednášek (4c cpp 0x, 5a Posix, 5b Win Threads) a SP.pdf z arcaa

6.1. C++11

- C++ ve standardu ++0x dočkal podpory vláken
- Výhodou je, že standardní knihovna je platformě nezávislá na úrovni zdrojového kódu
- Základním kamenem je třída `std::thread`

```
#include <thread>
#include <iostream>

void my_thread_func() {
  std::cout<<"hello"<<std::endl;
}

void write_sum(int x,int y){
  std::cout<<x<<" + "<<y<<" = " <<(x+y)<<std::endl;
}

int main() {
  std::thread t1(my_thread_func);
  std::thread t2(write_sum,123,456);
  t1.join();
  t2.join();
}
```

- Třída bere funkci, i např. přetížený operátor () objektu, jako první argument
  - A pak následují další argumenty
  - Argumenty se kopírují do interního úložiště threadu
- Je-li cílem spustit jinou funkci než funkční operátor (), předá se pointer na funkci (pointer, který se použije jako hodnota this) a případné argumenty

```
#include <thread>
#include <iostream>

class CSayHello {
  public:
    void greeting(std::string const& message) const {
      std::cout<<message<<std::endl;
    }
};


int main() {
  CSayHello x;
  std::thread t(&CSayHello::greeting,&x,"goodbye");
  t.join();
}
```

- pochopitelně se musí zajistit, že objekt přežije vlákno, např. lze takto:

```
int main() {
  std::shared_ptr<CSayHello> p(new CSayHello);
  std::thread t(&CSayHello::greeting,p,"goodbye");
  t.join();
}
```

- Je-li cílem předat referenci na existující objekt - tj. chceme-li zavolat () operátor přímo na konkrétním objektu, ne na jeho kopii
- `std::ref` se dá použít i pro argumenty

```
#include <thread>
#include <iostream>
#include <functional>

class CPrintThis {
  public:
    void operator()() const {
      std::cout<<"this="<<this<<std::endl;
    }
};

int main() {
  CPrintThis x;
  x();
  std::thread t(std::ref(x));
  t.join();
}
```

6.1.1. Synchronizace

- **std:mutex**
  - má lock, try_lock a unlock
  - není rekurzivní
- **std::recursive_mutex**
- **timed_mutex** a **recursive_timed_mutex** umožňují u `trylock` specifikovat dobu čekání
  - `try_lock_for` a `try_lock_until`
- důležitá je šablona **std::lock_guard**
  - zajistí, že daný blok kódu bude hlídán mutexem a že bude kritická sekce opuštěna vždy, když vykonávaný kód opustí daný blok kódu a to včetně zpracování vyjímek
  - => nemusí se tedy na konec `try` a do `catch` dávat `unlock`

```
std::mutex m;
unsigned counter=0;

unsigned increment() {
  std::lock_guard<std::mutex> lk(m);
  return ++counter;
}

unsigned query(){
  std::lock_guard<std::mutex> lk(m);
  return counter;
}
```

- Šablona **std::unique_lock** - umožňuje vytvořit zámek, který se zamkne, až mu řekneme

```
#include <mutex>
#include <thread>
#include <chrono>

struct Box {
  explicit Box(int num) : num_things{num} { }
  int num_things;
  std::mutex m;
};

void transfer(Box &from, Box &to, int num){
  // don't actually take the locks yet
  std::unique_lock<std::mutex> lock1(from.m, std::defer_lock);
  std::unique_lock<std::mutex> lock2(to.m, std::defer_lock);

  // lock both unique_locks without deadlock
  std::lock(lock1, lock2);

  from.num_things -= num;
  to.num_things += num;

  lock1.unlock();
  lock2.unlock();
}

int main() {
  Box acc1(100);
  Box acc2(50);

  std::thread t1(transfer, std::ref(acc1), std::ref(acc2), 10);
  std::thread t2(transfer, std::ref(acc2), std::ref(acc1), 5);

  t1.join();
  t2.join();
}
```

- **Podmínkové proměnné**
  - `Condition_variable`
  - `notify/All` a `wait/for/until`
  - `wait` bere jako parametr zámek a zámek musí být zamčený v době volání `wait`

```
std::condition_variable cv;
std::mutex cv_m;
int i = 0;


void waits() {
  std::unique_lock<std::mutex> lk(cv_m);
  std::cerr << "Waiting... \n";
  cv.wait(lk, [](){return i == 1;});
  std::cerr << "...finished waiting. i == 1\n";
}
void signals() {

  std::this_thread::sleep_for(std::chrono::seconds(1));
  std::cerr << "Notifying...\n";
  cv.notify_all();

  std::this_thread::sleep_for(std::chrono::seconds(1));
  i = 1;
```

```
  std::cerr << "Notifying again...\n";
  cv.notify_all();
}
```

- **Atomické operace a inicializace**
  - `call_once` zavolá funkci jenom jednou, i kdyby byla volána z více vláken
  - šablony `std::atomic*`
- **Vyhnutí se deadlocku** - std::lock – viz příklad výše
- **Asynchronní volání**
  - `std::async` vykoná kód funkce nezávisle na vláknu, ze kterého byla `std::async` zavolána
  - nezaručuje, že se vykoná v jiném vláknu
  - `std::future` vrací hodnotu `std::async` výpočtu

```
#include <future>
#include <iostream>

int calculate_the_answer_to_LtUaE();
void do_stuff();

int main(){
  std::future<int> the_answer=std::async(calculate_the_answer_to_LtUaE);
  do_stuff();

  std::cout<<"The answer to life, the universe and everything is "
  <<the_answer.get()<<std::endl;
}
```

- Komplikovaněji, ale zato s větší kontrolou nad vlákny, je možnost použití `std::promise`
  - Např. pro I/O operace
  - Promise je svázán s future
- Jeden thread pomocí `set_value` zapíše výsledek a druhý thread si ho pak vyzvedne
- Promise je v podstatě úložiště hodnoty, dokud si ji někdo nevyzvedne přes svázanou future
  - Producent vloží hodnotu do promise
  - Konzument si vyzvedne hodnotu z future

```
typedef int (*calculate)(void);

void func2promise(calculate f, promise<int> &p)
{
  p.set_value(f());
}
int main(int argc, char *argv[]) {
  getUserData();
  promise<int> p1, p2;
  future<int> f1 = p1.get_future(),
  f2 = p2.get_future();

  thread t1(&func2promise, calculateB, std::ref(p1)),
         t2(&func2promise, calculateC, std::ref(p2));

  c = (calculateA() + f1.get()) * f2.get();
  t1.join();  t2.join();

  showResult();
}
```

6.2. POSIX - PThreads

- Portable Operating System Interface
- API kompatibilní s UNIX-like operačními systémy (obecně ho může implementovat kterýkoliv OS - např. MS Windows)
- Aktuálně se dělí na tři části:
  - API jádra OS (Real-Time, vlákna, bezpečnost, IPC)
  - Příkazová řádka a utility
  - Validace
- jde o knihovnu pro jazyk C umožňující práci s vlákny
- nejsou vysokoúrovňový prostředek
- obsahuje tři základní typy objektů
  - vlákna (`pthread_t`)
  - mutexy (`pthread_mutex_t`)
  - podmínkové proměnné (`pthread_cond_t`), které jsou reprezentovány pomocí tzv. handle (zobecněný ukazatel)
- kromě handlů existují ještě tzv. atributované objekty (k popisu vlastností vláken, mutexů, podmínkových proměnných) `pthread_attr_t`, `pthread_mutexattr_t`, `pthread_condattr_t`
- vytváření a rušení objektů je dynamické, každé vlákno má svůj zásobník, tj. proměnné definované v programu vlákna jsou lokální.
- proměnné definované v hlavním programu jsou globální (sdílené a musejí se zamykat)

6.2.1. Vlákna

- má svůj zásobník
- má svůj kontext a prioritu
- program vlákna je vlastně funkce C s typem `void* fce_vlakna(void* arg)`
- vlákno se vytvoří následujícím způsobem:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void
*(*start_routine)(void*), void *arg);

int status;  //0 uspech, jinak chyba
status = pthread_create(&worker, NULL, fce_vlakna, args);
```

- `NULL` znamená ukazatel na atributy objektu, pokud je to `NULL`, vytvoří se automaticky atributový objekt s implicitním nastavením
- vlákno běží hned po vytvoření
- stavy vlákna jsou **ready**, **running**, **waiting** a **terminated** (handle je sice stále platný, ale použít lze už jenom `pthread_join`)
- vlákno se ruší pomocí `pthread_detach` – zneplatní handle vlákna a uvolní jeho kontext
- **Atributy**:
  - *způsob plánování*:
    - **FIFO** – nejprioritnější kategorie (nejdříve se plánují vlákna této kategorie)
    - **RR** – round-robin
      - vlákna běží podle priority v pořadí FIFO
      - plánují se po časových kvantech – střídají se ve výpočtu
      - když dojdou plánovatelná vlákna s nejvyšší prioritou, začnou se plánovat ty s nižší prioritou
      - až dojdou všechna, přepne se na **Other (FG, BG)**
    - **FG** – foreground, implicitní kategorie, střídání vláken, ty s vyšší prioritou mají více času
    - **BG** – background, všechna vlákna se střídají, ale dostávají méně času než FG
  - *priorita*
  - *rozměr zásobníku*
  - *hlídač zásobníku (Guardsize)* – jak daleko se můžeme od konce zásobníku dostat, jinak vznikne výjimka

- *konec vlákna*
  - **vlákno dojde na konec svého programu** (`return`, `pthread_exit`) - na toto ukončení se lze synchronizovat z jiného vlákna pomocí `pthread_join(handle, ret_val)` - u volání `pthread_exit` se návratová hodnota nevrací, není komu; při volání return se stejně volá `pthread_exit`
  - **zabito z vnějšku** – pokud možno nepoužívat, základní funkce pro likvidaci `pthread_cancel(handle);` oběť se může bránit `pthread_setcancelstate(…)`, k likvidaci nemůže dojít kdekoliv v kódu vlákna, ale jen v předem připravených místech (volání blokující funkce, volání `pthread_testcancel()`), popisovaný způsob je synchronní, existuje i asynchronní (`PTHREAD_CANCEL_ASYNCHRONOUS`) - ma cancel je možné využít destruktorů (pro úklid) `pthread_cleanup_push`, `pthread_cleanup_pop`

6.2.2. Mutexy

- existují k ochraně globálních dat, ne pro synchronizaci - k synchronizaci lze vyrobit semafory
- většinou implementovány jako busy-wait (je-li zámek zamčen vlákno se na něm zacyklí, tj. spotřebovává prostředky a je běžící či připravené k běhu)
- hodí se pro krátké kritické sekce z výše uvedeného důvodu
- vytvoření mutexu - `pthread_mutex_t zamek = PTHREAD_MUTEX_INITIALIZE;`
- **tři typy zámků:**
  - ***normální*** – konkrétní vlákno ho může zamknout jen 1x (`PTHREAD_MUTEX_NORMAL`, `PTHREAD_MUTEX_DEFAULT`)
  - ***rekurzivní*** – konkrétní vlákno ho může zamknout vícekrát, ale také ho musí tolikrát odemknout (`PTHREAD_MUTEX_RECURSIVE`) - pro rekurzivní zpracování globálních dat
  - ***ladící*** – funguje jako normální, ale zná vlastníky a pozná opakované zamčení (`PTHREAD_MUTEX_ERRORCHECK`)
- Základní funkce:
  - *`pthread_mutex_lock(&zamek);`*
  - *`pthread_mutex_unlock(&zamek);`*
  - *`pthread_mutex_try_lock()`* – neblokující zamykání
  - *`pthread_mutexattr_settype()`* - nastavení atributů

6.2.3. Podmínkové proměnné

- implementují frontu, kde vlákna *pasivně* čekají (spí) na splnění podmínky, neimplementují test podmínky (ten musím být v kódu vlákna).
- aby test podmínky a případná změna proměnné byla atomická akce, je třeba sdružit podmínkovou proměnnou s mutexem
- typy podmínkových proměnných:
  - *`pthread_cond_init(&podm_prom, &attr);`* - inicializace
  - *`pthread_cond_wait(&podm_prom, &mutex);`* - čekající vlákno musí odemknout mutex
  - *`pthread_cond_timedwait(&podm_prom, &mutex, &time)`* - čeká danou dobu, pak vrací error
  - *`pthread_cond_signal(&podm_pom);`* - jako *notify()*
  - *`pthread_cond_broadcast(&podm_prom);`* - jako *notifyAll()*
  - *`pthread_cond_destroy(&podm_prom);`* - destruktor

6.2.4. Semafor - dle přednášek tam patří taky

- `#include <semaphore.h>`
- `int sem_init(sem_t *sem, int pshared, unsigned int value);`
- `int sem_wait(sem_t *sem); /* P(sem), wait(sem) */`
- `int sem_post(sem_t *sem); /* V(sem), signal(sem) */`
- `int sem_getvalue(sem_t *sem, int *sval);`
- `int sem_trywait(sem_t *sem);`
- `int sem_destroy(sem_t *sem); /* undo sem_init() */`

6.2.5. Bariéra pomocí POSIX - trpí spurious wakeup

```
pthread_mutex_lock (&barrier_lock); // uzamknutí dat podmínky
barrier_cnt++; // změna dat podmínky, dělají všichni

if (barrier_cnt < N) { /* dělají všichni až na posledního */
  pthread_cond_wait(&cond_barrier, &barrier_lock);

/* blokující operace, tj. vlákno přejde do stavu waiting, ale odemkne se zámek -
kvůli tomu se do operace dává jeho adresa */

/* tady se vlákno probudí, ale už zase s uzamčeným barrier_lock */
}
else { /* poslední vlákno */
  barrier_cnt = 0;

/* nějaká změna dat a dále vyslání signálu "probuzení" pro všechna vlákna čekající v
podmínkové proměnné cond_barrier */

  pthread_cond_broadcast(&cond_barrier);
}
// sem už přijdou všichni, a pokaždé je zde zamčený zámek, musí se tudíž odemknout
pthread_mutex_unlock (&barrier_lock);
```

6.3. WinAPI

- preemptivní multitasking a preemptivní multithreading
- podpora kooperativních fibres jádrem

6.3.1. Proces

- bezpečnostní kontext = kdokoliv nemůže provádět cokoliv, jedinečný identifikátor, spustitelný kód
- prioritní třída – vlákna mohou mít prioritu pouze v rozsahu prioritní třídy procesu
- má alespoň jedno, primární, vlákno, které může vytvářet vlákna – threads a fibers

6.3.2. Thread

- entita v rámci procesu, kterou plánuje OS
- priorita
  - OS zajišťuje
    - Inverze priorit
    - Dynamic boost
    - Foreground/Input
  - včetně real-time
- jedinečný identifikátor
- TLS – Thread Local Storage
  - Data, která jsou specifická/lokální pro daný thread - `__declspec(thread) int number;`
  - Možnost, jak si thread může zabezpečit jedinečný přístup ke svým datům
  - Každý proces má k dispozici několik TLS slotů, které mohou být použity jeho thready
  - Možnost využití ke zpracování vyjímek

```
program ExplainTLS;

var       global:integer;
threadvar local:integer; //TLS

procedure X;
  var stack:integer;//také local,ale ne TLS
begin
end;
```

    - tři typy (viz kód výše):
        - **global** – jednu proměnnou sdílí všechna vlákna
        - **local** – každé vlákno má svou vlastní, využito TLS
        - **stack** – každé volání rutiny má vlastní proměnnou
- všechny vlákna sdílí paměťový prostor a zdroje svého procesu
- má kontext a zásobník
- může mít vlastní bezpečnostní kontext – možnost převzetí role někoho jiného (impersonating)
- Lze spustit na dostupných procesorech (`Get/SetProcessAffinityMask`)

6.3.3. Job

- možnost sloučit několik procesů dohromady a spravovat je jako celek
- operace provedená na jednom objektu ovlivňuje ostatní

6.3.4. Fiber

- běží v kontextu vlákna, plánuje ho thread procesu, jeden thread může naplánovat několik fibers
- FLS – Fibre Local Storage
    - Analogie k TLS
    - FLS je asociováno se threadem (Lazy init)
- má menší kontext (v porovnání s kontextem threadu) - zásobník, podmnožinu registrů a inicializační data
- má přístup do TLS threadu v jehož kontextu běží
- nemá prioritu

6.3.5. Stavy vlákna

- ke zjednodušení lze použít použít čtyř stavový model **ready, running, waiting, terminated**
- dle WMI:
    - ThreadState:
        - Initialized — sjelo z výrobní linky jádra OS
        - Ready — připraveno ke spuštění na některém z procesorů
        - Running — běží
        - Standby — bude spuštěno, vždy pouze jen jedno vlákno
        - Terminated — RIP
        - Waiting — není připraveno běžet, až bude, bude naplánováno
        - Transition — vlákno čeká na něco jiného než je procesor
        - Unknown — (klientu WMI) Neznámý stav
    - ExecutionState
    - Status
    - ThreadWaitReason

6.3.6. Funkce řízení běhu vlákna

- `CreateThread`
- `CreateRemoteThread` - v adresovém prostoru jiného procesu
- `Suspend, Resume`
- `Kill, Terminate, Exit`
- `Sleep`
- `Get/SetPriority`
- `SetThreadStackGuarantee`

6.3.7. ThreadPool

- množina vláken, která zpracovává asynchronní události pro process, místo vytváření a rušení vláken, která běží jen krátkou dobu
- farmer-worker model
- asociován s
    - frontou úkolů ke zpracování
    - waitable handles

    - časovačem
    - I/O
- výhodné např. pro aplikace pro distribuované vyhledávání v síti nebo místo vytváření a rušení vláken, která běží jen krátkou dobu

6.3.8. User-Mode Scheduling

- light-weight mechanismus jako fibres - možnost pro aplikaci, jak si plánovat svoje vlákna nezávisle na systémovém plánovači
    - ušetří se režie potřebná k přepínání mezi uživatelským režimem a režimem jádra
    - jsou efektivnější než ThreadPool, protože ten vyžaduje přepnutí
- na rozdíl od fibres má každý UMS thread svůj kontext
- Doporučeno jen pro aplikace, u kterých se předpokládají vysoké nároky na výpočetní výkon
- Dostupné od Windows 7 a Server 2008 R2 a pouze pro 64-bitové verze
- UMS aplikace
    - Má svůj *UMS plánovač*, který vytvoří jeden *UMS scheduler thread* pro každý procesor, na kterém může běžet *UMS thread*
    - Vytvoří *UMS thready* - konverzí z „normálních“ threadů
    - Spravuje frontu threadů, které dokončily činnost v režimu jádra - OS posílá oznámení o takových threadech, aby si je UMS aplikace mohla naplánovat podle svého
    - Provádí úklid po *UMS threadech*
    - UMS thread by neměl předpokládat, kdo ho plánuje
    - APC zamyká kontext UMS threadu a pak ho nelze naplánovat
- Pozor na sdílení systémových zámků mezi *UMS work threadem* a *UMS Schedulerem* –např. loader LoadLibrary - Worker ho uzamkne a Scheduler se už nevzbudí

6.3.9. Synchronizace

- probíhá pomocí synchronizačních objektů
- **Event** - může být buď pulsní, tj. překlopí se do *nonsignaled*, jakmile je zpracována `wait` funkcí nebo zůstane *signaled*
- **Mutex**
- **Semafor**
- **Časovač waitable timer**
- **Oznámení o změně** – Change notification
- **Standardní vstup**
- **Job**
- **Memory resource notification** - zápis do fyzické paměti
- **Proces**
- **Thread**
- Lze čekat i na handle **souboru, roury, komunikačního zařízení**, ale není doporučováno, místo toho se pro I/O používá asynchronní události – `OVERLAPPED`
- Objekt je buď *signaled*, nebo *nonsignaled* - čeká se pomocí tzv. `wait` funkcí
    - **Single-Object**
        - `SignalObjectAndWait`
        - `WaitForSingleObject/Ex`
    - **Multiple-Object**
        - `WaitForMultipleObjects/Ex`
        - `MsgWaitForMultipleObjects/Ex`
    - **Alertable Wait**

        - Čekání pomocí `SignalObjectAndWait` a `*Ex` se ukončí i v případě, že systém dokončil I/O operaci, nebo má nastat APC pro dané vlákno
    - **Registered Wait**
        - `RegisterWaitForSingleObject`
        - `UnregisterWaitEx`
        - Určeno pro thread pool
- `Wait` funkce se
    - *buď vrátí hned* - testovala se jen podmínka, ale nečekalo se nebo se mělo čekat, ale podmínka je splněna
    - *vrátí se po splnění podmínky*
    - *vypršel timout, lze čekat i do nekonečna*
- operace `ReadFile` jsou buď **blokující** (synchronní) nebo jsou **neblokující** (asynchronní)
    - v případě asynchronní může vlákno buď pomocí alertable wait funkce zjišťovat konec operce nebo se může periodicky dotazovat (fce `GetOverlappedResult, HasOverlappedIoCompleted`), případně může celou operaci zrušit (`CancelIo`)

6.3.10. APC

- Asynchronous Procedure Call - rutina, která se provede v kontextu daného threadu
- Každý thread má svou frontu APC, jednotlivé APC jsou plánovány plánovačem namísto normálního pokračování non-APC kódu vlákna
- Provedou se pouze tehdy, když je thread v alertable state - na základě volání `alertable wait` funkce, nebo `SleepEx`
- Jsou dva druhy
    - User APC
    - Kernel APC
- `ReadFileEx, SetWaitableTimer` a `WriteFileEx` jsou realizovány pomocí APC

6.3.11. Další možnosti synchronizace

- **Kritická sekce** - `Enter, TryEnter, Leave`
- **Podmínkové proměnné** - nelze je sdílet mezi procesy (lze však vytvořit pojmenovanou proměnnou, otevřít ji pod jménem v jiném procesu =>2 synch. objetky, se stejným stavem (sdílejí ho)
    - Jsou asociované buď s kritickou sekcí, nebo se slim lock
    - Podporují vzbuzení jednoho, nebo všech čekajících vláken (vhodné pro vlastní implementaci bariéry)
    - Čekáním na podmínkovou proměnnou se vlákno vzdá zámku kritické sekce a při vzbuzení ho znovu získá
- **Slim Locks** - Slim Reader/Writer (SRW) Locks
    - Optimalizované pro případy, kdy kritická sekce, nebo mutex, představují příliš velkou režii
    - U vstupu do kritické sekce nevím co se tam bude dít (čtení, zápis) => u slim locku se to dá poznat podle volané funkce a systém tak optimalizuje
    - Vyplatí se v případech, kdy se chráněná data více čtou než zapisují
    - 2 režimy ve kterých vlákno může přistupovat k datům
        - *Sdílený* - read-only; `AcquireSRWLockShared`
        - *Exkluzivní* – jenom jedno vlákno může zapisovat; `AcquireSRWLockExclusive`
- **Jednorázová inicializace** - jedna datová struktura, kterou se může pokoušet inicializovat několik vláken, ale smí být inicializována pouze jednou (pouze jedním vláknem)
    - *Synchronní* - `InitOnceBeginInitialize, InitOnceComplete, InitOnceExecuteOnce, InitOnceCallback`
    - *Asynchronní* - `INIT_ONCE_ASYNC`

- **Interlocked Variable Access** - atomické operace nad 32-bitovými proměnnými pod 32-bitovým systémem; stejně pro 64-bitů
  - Operace nad proměnnou větší než registr nejsou atomické – např. 64 bitů na 32-bitovém procesoru
  - `Increment, Decrement`
  - `Exchange, ExchangeAdd, CompareExchange`
- **Interlocked Single Linked List** - atomické operace nad jednosměrným seznamem
- **Fronta časovače** - umožňuje naplánovat vyvolání dané callback funkce v zadanou dobu (`CreateTimerQueue`)

6.3.12. Rizika synchronizace na víceprocesorových systémech

- cíl : uspořádání přístupu do paměti //popis jak se dělá volatile
- zápis do paměti je cachován pro zvýšení výkonu
- čtení dat jde z cache
- procesor provádí optimalizaci přístupu k datům spekulativním načítáním do cache, když předpokládá, že budou žádána
- paměťové operace mohou být prováděny mimo pořadí pro zvýšení výkonu
- řešením je paměťová bariéra, podpora ze strany procesoru – jak říci, kdy jsou která data k dispozici
  - *acquire* – před další instrukcí v zapsaném kódu
  - *release* – po další instrukci v zapsaném kódu
  - *fence* – dostupné v době provedení instrukce
  - funkce WinAPI se o to postarají

6.3.13. Zprávy

- možnost zasílání zpráv do front, asynchronní doručení z hlediska příjemce
- je zajištěna synchronizace, konzistence, při doručení do fronty zpráv příjemce
- `PostMessage` – neblokující odeslání zprávy
- `SendMessage` – odesílající je blokován do té doby, než je zpráva zpracována a vrácena návratová hodnota (`InSendMessage, SendMessageTimeout`)
- `Get/PeekMessage` – vyzvednutí/přečtení zprávy z fronty
- `WaitMessage` – vlákno se naplánuje až po přijetí zprávy
- `PostThreadMessage` – doručení do fronty zpráv konkrétního threadu

6.3.14. IPC - Inter Proces Communication

- lze synchronizovat i ta vlákna, která se nacházejí v různých procesech
- **DuplicateHandle**
  - duplikuje stávající handle objektu tak, aby ho mohl použít i jiný proces (např. `OpenSemaphore`)
  - výměna informací vyžaduje nějakou formu interakce mezi procesy
    - pojmenované objekty (roury, sdílená paměť – `CreateFileMapping`)
    - zaslání zprávy
- **Dědičnost -** `CreateProcess` – při vytváření dalšího procesu lze specifikovat, zda má zdědit platné handles rodičovského procesu
- **Sdílený segment DLL**
  - Jednu DLL může načíst více procesů
  - Vytvoří se nová sekce v DLL (vedle .text, .bss., atd.)
  - Podle flagů OS pozná, že pro ni má vytvořit sdílený segment
  - Data v tomto segmentu jsou pak sdílená pro všechny procesy, které danou DLL načtou (pochopitelně bez synchronizace, tu je třeba dodat)

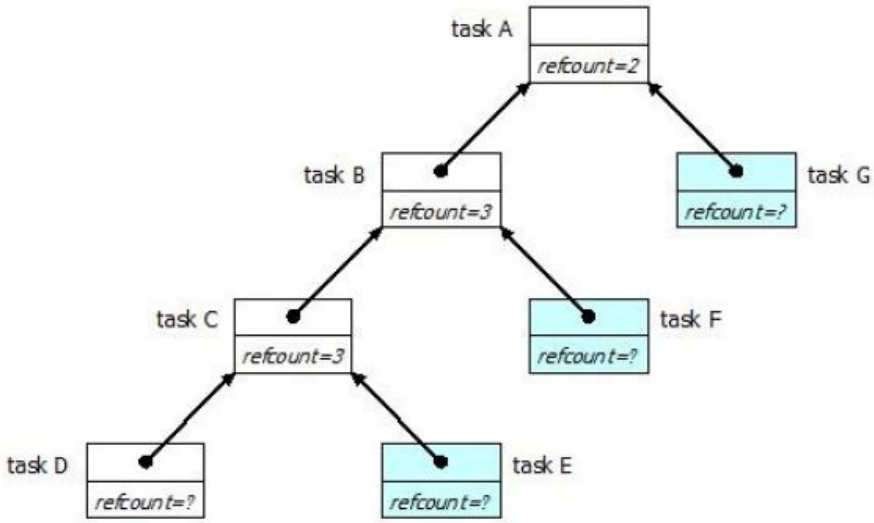
6.3.15. Rendez-Vous s WinAPI

- Je možné implementovat se všemi vymoženostmi
- Za pomoci dual-interfaces, variants, vlastního skriptu a RTTI je možné vytvořit implementaci, která bude ještě věrněji imitovat zápis v Adě

7. Intel Threading Building Blocks.

Status: nová otázka - zpracováno dle přednášek (6a Threadless)

- Open Source, podpora více platforem - stejně jako OpenMP
- Ačkoliv si je TBB vědoma vláken poskytovaných OS, myšlenka je taková, že programátor už s vlákny nepracuje
- Namísto vláken specifikuje úlohy, tasks, a jejich návaznosti
- Důvodem je mj. **redukce efektu cache-cooling**
  - Uvažujeme tradičních *m* vláken na *n* procesorů, kde $m > n$
  - Jak vlákno běží, jeho pracovní data se dostávají do cache procesoru (tzv. hot data)
  - Jak se vlákna přepínají, do cache procesoru se dostávají pracovní data nového vlákna a pracovní data předchozích vláken jsou z cache odstraňována
  - Takže až takové vlákna přijde opět na řadu, každý cache-miss ho bude stát stovky cyklů čekání, než se jeho pracovní data znovu nahrají do cache (protože byly tzv. cooled)
  - S TBB programátor nepíše vlákna, ale úlohy
  - Vlákna poskytuje TBB tak, aby m<=n
  - Vlákno TBB pracuje na jedné úloze, dokud není dokončena => redukuje efekt cache-cooling
  - Tradiční vlákna ani thread-pool nic takového neumí! =>tradiční vlákna jsou dobrá na GUI a I/O, ale ne na výpočty
- **Task Stealing** je způsob plánování úloh



- Úlohy A, B, C vytvořily podúlohy, na které čekají
- Úloha D běží
- Úlohy E, F, G ještě neběží
- Nejhlouběji zanořené úlohy jsou nejvíce cache-hot
- Breadth-first co nejdříve rozbalí celý strom, takže maximalizuje paralelismus
- Depth-first zase udržuje omezený počet uzlů stromu
- Takže je to v praxi kompromis, protože počet procesorových jader je omezený
- Na každý procesor je zásobník úloh a pokud jeden procesor dokončí svůj zásobník, může zkusit „ukrást“ cool data z vrcholu zásobníku jiného procesoru aneb FIFO ani priority nejsou vždy ta nejlepší strategie
- TBB vykonává úlohy paralelně, jak nejlépe to se současným know-how jde
  - Obsahuje různé další optimalizace v jakém pořadí úlohy vykonávat
  - Lze si napsat vlastní plánovač
- Stejně jako STL, i TBB intenzivně používá C++ šablony (tj. nelze ji použít v C jako OpenMP)
- Základní konstrukce TBB jsou:
  - `parallel_for, parallel_reduce, parallel_scan`

    - parallel_while, parallel_do,pipeline, parallel_sort
    - paralelní kontainery, které jsou v STL
    - mutex, spin_mutex, queuing_mutex, spin_rw_mutex, queuing_rw_mutex, recursive mutex
    - fetch_and_add, fetch_and_increment, fetch_and_decrement, compare_and_swap, fetch_and_store
- TBB dokáže použít vlastní paměťový manažer, který je optimalizovaný na paradigma výpočtu úloh
- TBB je náročnější se naučit, ale zase se to vyplatí na výkonu, pokud se začíná psát nová aplikace, než např. paralelizovat s OpenMP
- Navíc TBB se nespoléhá na direktivy pro překladač, takže podmínka if OpenMP se dá realizovat libovolně složitá, nebo se dají udělat konstrukce, které by šli s OpenMP udělat jen velmi obtížně – např. cancellation výpočtu v OpenMP není
- Např. chceme-li spustit na pozadí několik výpočetně náročných úloh paralelně

```
tbb::task* CMasterCalculationTBBLogic::execute() {
  tbb::task_list list;

  for (int i=gaiFirst; i<=gaiLast; i++)
    list.push_back(*new(allocate_child()) CTask(i));

  set_ref_count(gaiLast-gaiFirst+2); //počet úloh +1
  spawn_and_wait_for_all(list);

  return NULL; //non-NULL by byla úloha, která by měla být spuštěna jako
}              //další/závislá na téhle
```

- násobení vektoru

```
class CMulVect {
  floattype *mA, *mB;
  int mLen;

  public:
    floattype mProduct;
    CMulVect(floattype *a, floattype *b, int len) :
             mA(a), mB(b), mLen(len), mProduct(0.0) {}
    CMulVect(CMulVect& x, tbb::split) : mA(x.mA),
             mB(x.mB), mLen(x.mLen), mProduct(0.0) {}

    void join(const CMulVect& y) {
      mProduct += y.mProduct;
    }

    void operator()(const tbb::blocked_range<size_t>& r ) {
      int r_end = r.end();
      floattype *a = mA;
      floattype *b = mB;
      floattype sum = 0.0;

      for (int i=r.begin(); i!=r_end; ++i)
        sum += a[i]*b[i];

      mProduct += sum;
    }
};

floattype MulVect(floattype *a, floattype *b, int len) {
```

```
  floattype result = 0.0;

  //Jsou data tak velká, aby se je vůbec
  //vyplatilo počítat paralelně?
  if (len<=rmSerialThreshold) {
    for (int i=0; i<len; i++)
      result += a[i]*b[i];

  } else {
    CMulVect mul(a, b, len);
    tbb::parallel_reduce(tbb::blocked_range<size_t>(0,len), mul);
    result = mul.mProduct;
  }

  return result;
}
```

- U redukčních operací je třeba při inicializaci mezivýsledku rozlišovat mezi konstruktorem a funkčním operátorem ()
- Konstruktory jsou dva - normální, který bere parametry výpočtu a split konstruktor, který bere referenci na druhý objekt a na tbb::split
- V konstruktorech se vždy inicializují dílčí mezivýsledky celé operace
- Návrhově to totiž může svádět k tomu, aby se mezivýsledek inicializoval v těle funkčního operátoru jenže funkční operátor se může volat několikrát na stejné instanci, a pak by to dávalo špatné výsledky

7.1. Případová studie použití TBB - TBB vs vlákna

- Uvažujme výpočetně náročnou aplikaci nad rozsáhlými daty s GUI
- Jedno vlákno bude obsluhovat GUI
- Jedno vlákno bude obsluhovat I/O
  - Naivní přístup je jedno vlákno pro zápis a jedno pro čtení
  - Lepší je jenom jedno vlákno a použití asynchronních I/O operací (OS si je uspořádá sám pro lepší výkon - např. disky mají Native Command Queuing)
  - Ale co s výpočetní částí? Určitě v tom bude alespoň jedno vlákno, aby výpočet běžel na pozadí a GUI bylo responsive
- Přístup s psaním vláken by znamenal postarat se
  - Vytváření a rušení vláken, což je další režie pro OS
  - Jejich správnou synchronizaci, což je náročné, jakmile se program stává složitější
  - Výkonnostně je rozdíl, jestli se synchronizuje v kernel-mode, nebo v user-mode address space
  - Výkonnostně hraje roli, zda se k datům přistupuje ze stejného procesoru – thread affinity
  - Optimální počet vláken odpovídá počtu procesorových jader v systému, což ale není přístup, který je vždy možný při psaní vláken, např. počet vláken může odpovídat tomu, jaká je metoda výpočtu – problém škálovatelnosti (s TBB se taková vlákna nahradí úlohami)
- S TBB
  - O výše uvedené problémy se TBB postará
  - Programátor napíše jedno vlákno, ve kterém pak spustí výpočet úloh TBB
  - Nicméně programátor si stále musí být vědom toho, jak se píší paralelní programy s vlákny, aby mohl správně používat synchronizační primitiva TBB
    - Optimálně se naprogramují pouze úlohy s tím, že každá úloha po dokončení vrátí další úlohu, která se má vykonat jako navazující
    - Jako bariéra na dokončení více úloh se pak použije task list

- Úlohy však mohou sdílet proměnné, a proto je třeba znát teorii o psaní vláken, aby byla představa, že 2 úlohy mohou, ale i nemusí, běžet paralelně, např. když se má správně použít mutex, podmínka…

8. OpenCL.

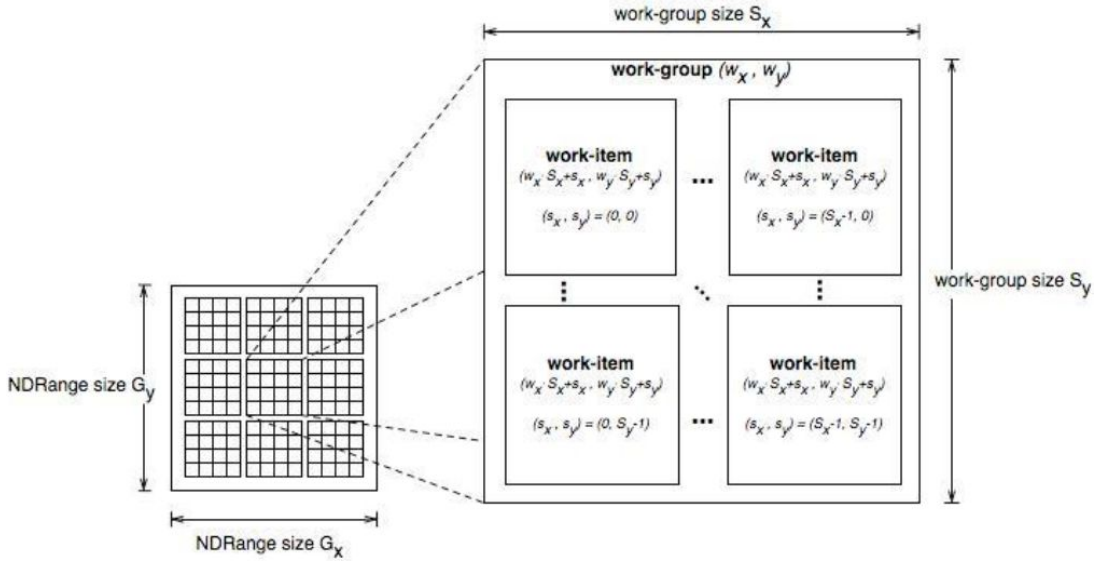
Status: nová otázka - zpracováno dle přednášek (6b GPU)

- OpenCL je původně od Apple, ale dnes už otevřený standard
- OpenCL zase běží i na hardware i ostatních výrobců oproti CUDA (pouze pro NVIDIA)
- OpenCL umí „fallback to CPU“, není-li k dispozici GPU
    - Jeden kód v programu, lépe se udržuje, ale nejspíš bude pomalejší, než dobře napsaný kód pro CPU – tj. běží všude, ale někde pomalu
- je třeba myslet jako GPU, ne jako CPU, jde-li nám o efektivní využití GPU
- **Device** – zařízení, kde se vykoná GPU-kód, tj. GPU, v případě fallbacku CPU
- **Kernel** – funkce v GPU-kódu, která se vykoná na Device aneb zkompilovaný program zapsaný v OpenCL
- **Host** – aka CPU
- **Platform** – Host + všechny dostupné Devices

Součet vektorů na CPU	Součet vektorů na GPU

`void vector_add_cpu (const float* src_a,`
`  const float* src_b,`
`  float* res,`
`  const size_t num) {`
 	
`  for (size_t i = 0; i < num; i++)`
`  res[i] = src_a[i] + src_b[i];`
`}`
	`__kernel void vector_add_gpu (`
	`  __global const float* src_a,`
	`  __global const float* src_b,`
	`  __global float* res,`
	`  const int num) {`
	
	`/* get_global_id(0) vrátí ID aktuálního threadu. Jelikož jich na Device běží několik zároveň, každý sečte jenom svůj prvek vektoru. */`
	
	`const int idx = get_global_id(0);`
	
	`/* Podmínka je nutná, protože může být více threadů než prvků vektoru. Pokud by bylo více prvků vektoru než threadů, tak jeden thread může sečíst více prvků ala modulo. */`
	`  if (idx < num)`
	`  res[idx] = src_a[idx] + src_b[idx];`
	`}`

- kernel musí vracet void a být deklarován s `__kernel`
- pointer na paměť, se kterou se pracuje, musí být deklarován s `__global`
- prvek vektoru je tzv. *work-item*, které se sdružují do tzv. *work-group*, které se sdružují do tzv. *ND-Range*
- kernel se vykoná právě jednou pro každý *work-item*



- Ačkoliv lze mít jednu work-group, která obsáhne všechny work-items, není to dobrý nápad
  - Jedna workgroup se totiž celá počítá na jedné compute unit, ale jedna compute unit může vykonat více work-groups, takže v případě více compute units by ostatní compute units zůstaly nevyužité při jedné velké work-group, zatímco při několika work-groups se mohou využít všechny dostupné compute-units
  - Optimální počty závisí na použitím hardwaru, ale velikost work-group by měla být dělitelná 32 (nVidia warp) nebo 64 (AMD wavefront)
  - Maximální velikost work-group je `DEVICE_MAX_WORK_GROUP_SIZE`, případně `KERNEL_WORK_GROUP_SIZE`, kterou vrací fce `GetKernelWorkGroupInfo`, a která může být menší než `DEVICE_MAX_WORK_GROUP_SIZE`
  - Zároveň musí platit, že velikost workgroup beze zbytku dělí počet work-items
- OpenCL program se překládá až na cílovém počítači přímo pro konkrétní hardware, který je tam k dispozici
- Z pohledu programu jde o tzv. kontext a frontu příkazů

```
cl_int error = 0;
cl_platform_id platform;
cl_context context;
cl_command_queue queue;
cl_device_id device;

// Platform
error = oclGetPlatformID(&platform);
if (error != CL_SUCCESS) {...}

// Device - CL_DEVICE_TYPE_GPU je typ zařízení,
//1 znamená, že chceme deskriptor 1x GPU
error = clGetDeviceIDs(platform,
CL_DEVICE_TYPE_GPU, 1, &device, NULL);
if (err != CL_SUCCESS) {...}

// Context – zjistili jsme 1x GPU, tak vytvoříme
// kontext 1x GPU, ale lze dát i více devices
context = clCreateContext(0, 1, &device, NULL, NULL, &error);
if (error != CL_SUCCESS) {...}

// Command-queue – fronta, do které budeme zadávat příkazy, co se má vypočítat
queue = clCreateCommandQueue(context, device, 0, &error);
if (error != CL_SUCCESS) {...}
```

- Ale než něco spočítáme, je třeba alokovat paměť tak, aby ji Device viděl a mohl ji číst a zapisovat

```
float* src_a_h=new float[count];
```

```
cl_mem src_a_d = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                                mem_size, src_a_h, &error);
…
//Žádný garbage collector, paměť se uvolňuje manuálně!
delete[] src_a_h;
clReleaseKernel(vector_add_k);
clReleaseCommandQueue(queue);
clReleaseContext(context);
clReleaseMemObject(src_a_d);
```

- Program -> kompilace -> Kernel -> výpočet
    - Pochopitelně stále platí, že co se dá udělat jednou, to se udělá jenom jednou a na začátku běhu programu, aby se to dalo opakovaně využívat a nezdržovalo to vlastní výpočet

```
//Kromě .c kódu taky existují C++ wrappery, se kterými
//může být život o něco jednodušší
//buď načteme .cl soubor, nebo to může být statický řetězec v kódu programu
ifstream file(kernelFile);
string prog(istreambuf_iterator<char>(file),(istreambuf_iterator<char>()));

cl::Program::Sources source( 1, make_pair(prog.c_str(), prog.length()+1));
cl::Program program(context, source);
file.close();
//zkompilujeme kernel pro dané zařízení
program.build(devices);

//a získáme kernel, kterému můžeme předat parametry a spustit ho
cl::Kernel kernel = cl::Kernel(program, "kernel_function_name");
```

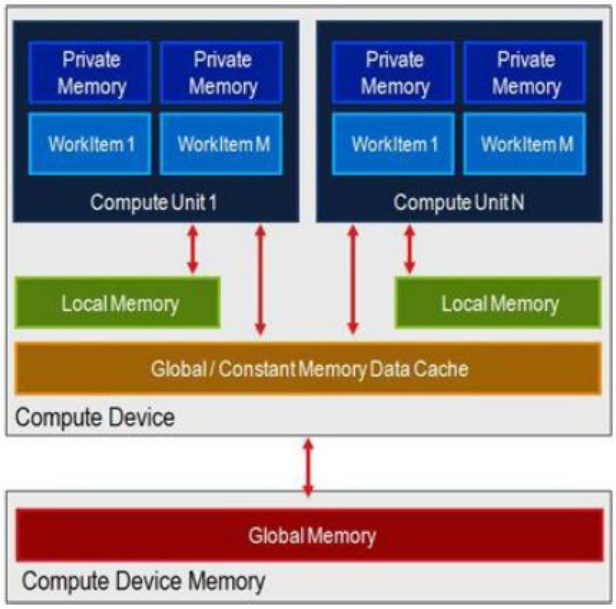
- teď už v chybí jenom předat parametry a zařadit kernel do fronty ke spuštění

```
error = clSetKernelArg(vector_add_k, 0, sizeof(cl_mem), &src_a_d);
error |= clSetKernelArg(vector_add_k, 1, sizeof(cl_mem), &src_b_d);
error |= clSetKernelArg(vector_add_k, 2, sizeof(cl_mem), &res_d);
error |= clSetKernelArg(vector_add_k, 3, sizeof(size_t), &size);

assert(error == CL_SUCCESS);
// Launching kernel
const size_t local_ws = 512; // Number of work-items per work-group

// shrRoundUp returns the smallest multiple of local_ws bigger than size
const size_t global_ws = shrRoundUp(local_ws, size); // Total number of work-items

error = clEnqueueNDRangeKernel(queue, vector_add_k, 1, NULL, &global_ws, &local_ws,
                               0, NULL, NULL);
```

- global work-size je počet všech work-items
- local size je počet work-items v jedné work-group
- než se kernel dopočítá, můžeme mezitím dělat i jinou, užitečnou činnost
    - Poslední parametr je `clEnqueueNDRangeKernel` totiž event, který je signalizován v době dokončení kernelu
    - A podobně jsou předposlední dva parametry eventy, které musí být signalizovány, než se může daný kernel spustit
- Rekurze
    - Ne každý GPU hw to umí, a rozhodně to není dobrý nápad používat
    - GPU dosahuje ve výpočtech urychlení nad CPU právě proto, že GPU nemusí řešit stack a s ním související branch-prediction
    - A rozhodně není praktické kopírovat velké registry GPU na zásobník a zpět – push/pop
- lze použít vlastní datové typy - `typedef struct`
    - nutno definovat kromě v .c/.cpp kódu definovat i v .cl kódu

```
typedef char my_char[8];
typedef struct tag_my_struct {
  long int id;
  my_char chars[2];
  int numerics[4]
  float decimals[4];
} my_struct;

__kernel void foo(__global my_struct * input, __global int * output) {
  int gid = get_global_id(0);
  output[gid] = input[gid].numerics[3]== 2 ? 1 : 0;
}
```

- Plánování vs. vykonání
  - work-items v jedné work-group budou naplánovány dohromady, ale už nikdo nezaručuje, že příslušné kernely budou vykonány zároveň - to závisí na ovladačích a hw, který se snaží zamaskovat zpoždění při práci s pamětí
- Paměť
  - **Global**
    - Nejpomalejší paměť celého GPU subsystému
    - Výkon závisí na tom, jak se s pamětí pracuje
    - Což závisí na konkrétním kusu hw a tudíž se vyplatí si pamatovat, že se má omezit konkurenční přístup ke stejné paměti z několik různých work-items, které jsou vykonávány (tj. mohly by být vykonávány) současně (tzv. bank-conflict)
  - **Private**
    - Rychlá paměť pro použití při výpočtu jednoho work-item (tj. threadu GPU)
    - Její velikost, ani minimální ani maximální, není definována žádným standardem, takže závisí na konkrétním hw
    - Takže použít jí co nejméně, protože jakmile nároky na privátní paměť překročí limit hw, tak se to uloží do globální paměti a to degraduje výkon
  - **Local**
    - Opět je rychlejší než globální paměť
    - Slouží ke sdílení dat mezi jednotlivými work-items v jedné work-group
  - **Constant**
    - Read-only paměť
    - Dává hw možnost, aby optimalizoval konkurenční přístup k paměti, která je jenom ke čtení (tj. aby se vyhnul onomu bank-conflict)



8.1. Násobení vektorů

- Nejprve by se spustil kernel, který vynásobí prvky vektorů

```
__kernel void dot_mul_kernel(__global const double * x, // input vector
                             __global const double * y, // input vector
                             __global double * r,       // result vector
                             uint n)                    // input vector size
{
  uint id = get_global_id(0);
  if ( id < n ) {
    r[id] = x[id] * y[id]; // multiply elements, store product
  }
}
```

- Aby se pak pustil další kernel, který je sečte anebo se to dá udělat ještě takhle

```
#define LOCAL_GROUP_XDIM 256
__kernel __attribute__((reqd_work_group_size(LOCAL_GROUP_XDIM, 1, 1)))
  void dot_local_reduce_kernel(__global const double * x, // input vector
                               __global const double * y, // input vector
                               __global double * r,       // result vector
                               uint n)                    // input vector size
{
  uint id = get_global_id(0);
  uint lcl_id = get_local_id(0);
  uint grp_id = get_group_id(0);

  double priv_acc = 0; // accumulator in private memory
  __local double lcl_acc[LOCAL_GROUP_XDIM]; // accumulators in local memory

  if ( id < n ) {
    priv_acc = lcl_acc[lcl_id] = x[id] * y[id]; // multiply elements, store product
  }
  barrier(CLK_LOCAL_MEM_FENCE);

  // Find the sum of the accumulators.
  uint dist = LOCAL_GROUP_XDIM;

  // i.e., get_local_size(0);
  while ( dist > 1 ) {
    dist >>= 1;

    if ( lcl_id < dist ){
      // Private memory accumulator avoids extra local memory read.
      priv_acc += lcl_acc[lcl_id + dist];
      lcl_acc[lcl_id] = priv_acc;
    }

    barrier(CLK_LOCAL_MEM_FENCE);
  }

  if ( lcl_id == 0 ) { //Store the result (the sum for the local work group).
    r[grp_id] = priv_acc;
  }
} //konec kernelu
```

- Barrier vs. Fence
  - **Bariéra** slouží k synchronizaci vláken vykonávajících kernel na jednotlivých work-items v jedné work-group
  - Různé work-groups mezi sebou nelze synchronizovat

- **Fence** zajistí, že se všechny load/store instrukce dokončí před fence, tj. předtím, než se začnou vykonávat další load/store instrukce, které jsou v programovém kódu po fence
- Fence se vztahuje jenom na jeden work-item
- Bariéra může volat fence pro všechny work-items

9. Rendez-Vous, vč. select v Adě, a jeho porovnání s monitorem Javy.

Status: potřebuje revizi, mírně se liší od původní otázky - zpracováno dle přednášek (4d Ada) a dle PPR_NOVE_2014.pdf

- Ada je objektově orientovaný jazyk se silnou verifikací typů(nelze implicitně přetypovat datové typy –např. void pointer)
- nepoužívá interpretr
- nepodporuje fibres
- překladače musejí projít testem na soulad se standardem jazyka
- paralelní části výpočtu se označují jako tasky (ne thready)
- mohou být prováděny na jednom procesoru, více procesorech nebo více počítačích, není to však vyjádřeno v kódu programu
- pro synchronizaci tasků se používá asymetrické synchronní rendezvous (zasílání zpráv)

Tasky

Konstrukce task představuje program paralelně proveditelného procesu, schopného komunikace s ostatními procesy.
Deklarace je následující:

```
task [type] jméno is
  deklarace jmen komunikačních typů
end jméno;

task body jméno is
  lokální deklarace a příkazy
end jméno;
```

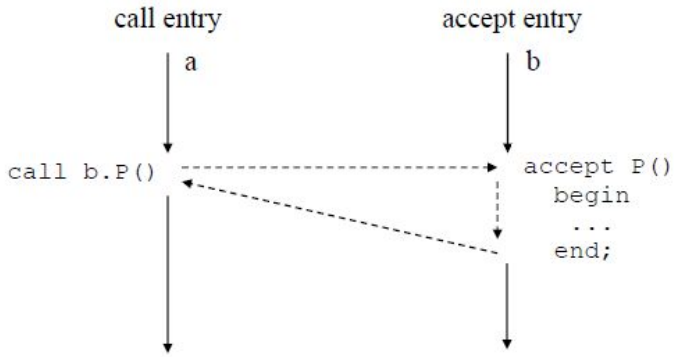
- pro ukončení procesu (násilné ukončení tasku) lze použít příkaz `abort jméno;`, ale jeho použití by mělo být výjimečné
- k ukončení tasku se používá příkaz `terminate`

```
select
   accept e
or
   terminate
end select
```

Rendez-vous

Pro interakci procesů používá ADA principu asymetrického (1 vlákno v roli klient, druhé v roli server – akceptuje rendezvous, klient musí znát (mít odkaz na) server, naopak ne) rendezvous, kterým eliminuje potřebu semaforů (umí je nahradit), je synchronní (vlákna se při rendezvous synchronizují). Dovoluje tak elegantní konstrukci monitorů (protože může být aktivní pouze jeden volající).

- prostředek pro synchronizaci úkolů(tasks)
- dva úkoly spolu komunikují pomocí rendez-vous -Meeting point, entry calls
- task je uspán do té doby, než se dostaví druhý task, který s ním chce komunikovat



- Synchronní – call a accept jsou blokující
  - sdružuje se synchronizace a výměna dat
  - Náročná implementace v distribuovaném prostředí
  - Accept entry je atomická operace se vzájemným vyloučením

```
task Simple_Task is
   entry Start(Num : in Integer);
   entry Report(Num : out Integer);
end Simple_Task;
task body Simple_Task is
   Local_Num : Integer;
   begin
      //čeká na vložení čísla –entry call
      accept Start(Num : in Integer) do
         Local_Num := Num;
      end Start;
      //normálně pokračuje v běhu
      Local_Num := Local_Num * 2;
      //čeká na vyzvednutí spočítané hodnoty
      accept Report(Num : out Integer) do
         Num := Local_Num;
      end Report;
end Simple_Task;
```

- uvedený příklad stačí, pouze pokud potřebujeme jen jedno vlákno běžící podle uvedeného kódu - pro více instancí je potřeba deklarovat s task s klauzulí `type`
- task je vytvořen v okamžiku, kdy je nadeklarovaná instance
- průběh dostaveníčka - accept:
  - klient zavolá server
  - server si převezme parametry
  - server provede výpočet, klient spí
  - server předá výsledky

Select

- může být nezbytné, aby úkol mohl reagovat na několik vstupních volání (entry calls) –pokaždé na jiné dle okolností –tj. ne v předem určeném pořadí
- pokud nelze splnit podmínku select je možné v else např. vyhodit vyjímku

```
//Vynutíme si inicializaci a další se
//už pak může vykonávat v libovolném
//pořadí.
accept Init(Item : in Integer) do
   Local_Item := Item;
end Init;
loop
```

```
   select
      accept Stop;
         exit;
      or
      when podminka => //může i nemusí být
         accept Put(Item : in Integer) do
            Local_Item := Item;
         end Put;
         Local_Item := Local_Item * 2;
      else
         Put_Line("No entry call at this time");
   end select;
   delay 0.01;
end loop;
```

Protected Objects, Protected Types

- tasky mohou sdílet objekty
- objekt je instance typu –klíčové slovo *type*
- klíčové slovo *protected* zajistí exkluzivní přístup k chráněnému objektu
- jsou tři operace nad chráněnými objekty:
  - **Procedury** – mění stav objektu, aniž by pro to musela být splněna podmínka; překladač se stará, aby měly exkluzivní přístup k objektu
  - **Entry calls** – stejné jako procedury, ale pro vykonání entry call je třeba navíc splnit podmínku
  - **Funkce** – pouze vrací stav a nic nemění, a proto nemusí mít exkluzivní přístup k objektu

```
protected type Counting_Semaphore is
  entry Acquire;
  procedure Release;
  function Count return Natural;

private
  Holding_Count : Natural := 0;
end Counting_Semaphore;

protected body Counting_Semaphore is
  entry Acquire when Holding_Count < 5 is
  begin
    Holding_Count := Holding_Count + 1;
  end Acquire;

  procedure Release is
  begin
    if Holding_Count > 0 then
      Holding_Count := Holding_Count - 1;
    end if;
  end Release;

  function Count return Natural is
  begin
    return Holding_Count;
  end Count;
end Counting_Semaphore;
```

Porovnání s Javou

- Java nemá chráněné objekty ve smyslu Ady, ale pouze jeden globální zámek na objekt/statickou třídu (chráněný objekt je mnohem více než javovský monitor)
- Je třeba vytvořit EntryCall objekt, který bude spravovat frontu klientů – bude ji však používat i server

    - Je nutné použít wait a notify a tím se zaručí nedeterministické chování, protože nelze se 100% spolehlivostí dopředu určit, kdy se vzbudí které vlákno – pořadí není garantované
    - Bylo by nutné kompletně přepsat mechanismus uspání a vzbuzení vlákna
- Další problémy na obzoru, které souvisejí se samotnou implementací konkrétního JVM a k nimž nemusí existovat řešení, protože do plánovače JVM a správy paměti (chráněný objekt) se interpretovaný kód nedostane
- Rendez-vous je zajímavé hlavně ve spolupráci se select a Java nemá ekvivalent pro select Ady
- Oba poskytují strukturovaný způsob vzájemného vyloučení (v obou způsobech je vzájemné vyloučení implicitní).
- Monitory jsou **pasivní** objekty.
    - Existují pouze klientské thready.
    - Tyto thready pro sebe provádí službu uvnitř monitoru.
    - Stav serveru se nemění samovolně.
- Rendez-Vouz moduly obsahující serverové thready jsou **aktivní** objekty.
    - Thready serveru se v modulu chovají podle klientských threadů po dobu trvání rendez-vous.
    - Thready serveru mohou změnit stav modulu mezi voláními od klientů.

10. Výpočetní prostředí s distribuovanou pamětí.

Status: potřebuje revizi, mírně se liší od původní otázky - přednáška (7a Distributed)

MPMD

- Systém pro paralelní výpočet s distribuovanou pamětí se skládá z výpočetních uzlů a komunikačních kanálů.
- Tři základní možnosti realizace
  - Univerzální počítačová síť (software umožňuje využívat počítač jako uzel, SETI)
  - Univerzální paralelní počítač (hlavní úlohou je výpočet paralelní aplikace, Cluster)
  - Jednoúčelový paralelní počítač (počítač postavený pro jednu konkrétní aplikaci s maximální optimalizací, TWINKLE)
- protože neexistuje sdílená paměť, používá se pro komunikaci mezi procesy především zasílání zpráv
- protože systémy s distribuovanou pamětí nemají žádný úzký profil ve formě sběrnice, přes kterou by procesory přistupovaly ke sdílené paměti, hodí se pro úlohy vyžadující tzv. masivní paralelismus (stovky až tisíce procesorů)
- Obecně systém s distribuovanou pamětí umožňuje větší urychlení než systém se sdílenou pamětí díky paralelizaci komunikace
  - Zatímco se data přenášejí kanálem, uzel může počítat
  - Urychlení závisí na
    - Objemu interakce
    - Celkovém objemu zpracovávaných dat
    - Konkrétní hw architektura
    - Jak dalece je použitý programový kód optimální pro danou architekturu

HW pohled

Topologie obecně

- Pravidelná - síť má strukturu popsatelnou grafem pravidelné struktury (kruh, mřížka, krychle, ...)
- Nepravidelná – např. Internet

Směrování

- Pevná topologie – můžete poslat vzkaz pouze sousedovi
- Podle příjemce – směrování jak ho známe např. z IP
- Podle odesílajícího – odesílající si určí cestu
  - IP „strict source and record route“ (SSRR)
  - IP „loose source and record route“ (LSRR) - (často blokováno z bezpečnostních důvodů – address spoofing (podvrhnutí adresy))

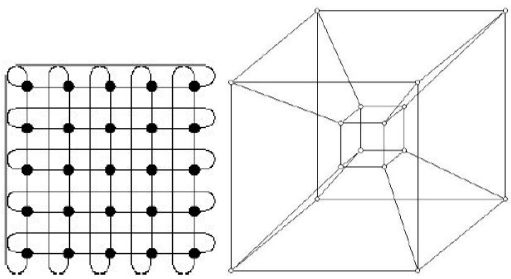
Fyzická topologie

- Pevná – procesory jsou spojeny komunikačním kanálem
- Flexibilní
  - Circuit switching
  - Packet switching - sériová linka s CSMA

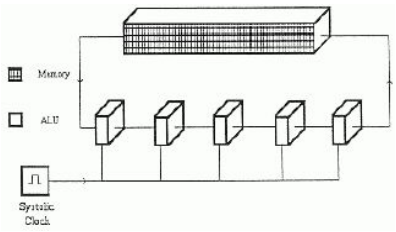
Pevná topologie

- každý s každým
- 2D/3D mřížka
- Toroid

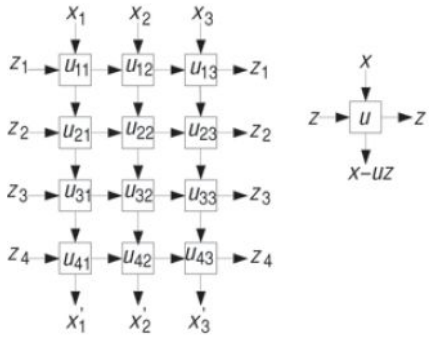
- n-rozměrná krychle
- Systolické pole
  - Několik vzájemně propojených uzlů, které si postupně s každým taktem hodin předávají data k dalšímu výpočtu - něco jako sériová linka
  - Synchronní, lock-stepped
  - Protějšek von Neumannovy architektury - výpočet řídí data, ne program counter
  - Komplexnější pipeline
  - Komunikační kanály používají simplex


Toroid

4-rozměrná krychle



Systolické pole



Systolické pole - ve dvou směrech

Parametry

- N - Celkový počet uzlů v síti
- d_{ij} - vzdálenost mezi dvěma uzly (sousedé mají 1)
- d_{max} - nejhorší varianta, kolika uzly musí projít zpráva, než je doručena (nejdelší cesta zprávy v celém systému) - 2D mřížka dmax = 2(n – 1), toroid dmax = n-1
- počet sousedů - s kolika dalšími uzly je daný uzel spojen přímo
- přenosová kapacita
  - agregovaně – kolik uzlů může najednou posílat zprávu
  - odolnost proti chybám – kolik komunikačních kanálů musí selhat, než se z jedné sítě stanou dvě

Snahou je dosáhnout

- Co největšího počtu uzlů v síti - škálovatelnost
- Co nejmenší komunikační vzdálenosti (d_{max}) - tj. omezit komunikační zpoždění
- Co nejmenší počet sousedů - i komunikační kanál něco stojí
- Dosáhnout co největší přenosové rychlosti

Fyzická adresa uzlu

- Sekvence bytů s nějakým významem - např. 2x integer pro mřížku (`struct addr {int x, y;}`)
- měla by souviset s polohou uzlu v síti
- mělo by z ní být možné odvodit fyzické adresy sousedů
- komunikační operace - pokud nelze docílit, aby spolu komunikovali pouze sousedé, pak
  - **Store & Forward** – přijme se celý paket, aby se poté poslal dál
  - **Wormhole Switching/aka Routing**
    - paket je rozdělen na menší jednotky - flit (flow unit)
    - první flit obsahuje adresu, podle které se určí rozhraní odchozího komunikačního kanálu, kam jsou automaticky poslány zbývající flits daného paketu
    - zmenšuje komunikační zpoždění, ale vytváří nové možnosti k uvíznutí

        - velikost bufferů rozhraní vs. rychlost zpracování vs. počet simultánních flitů z různých paketů, atd.
- pokud sestavujeme vlastní paralelní počítač, např. jednoúčelový, je to věc našeho návrhu
- ale co když máme využít např. IP - tj. univerzální počítačová síť a adresa, ze které lze odvodit sousední uzly, jak je má vidět výpočet
- Vlastní adresní rozsah, kdy číslo počítače budou zakódované souřadnice uzlu - měly by také odpovídat umístění počítače v síti
- Prezentovaná síťová topologie
    - Fyzická topologie, tj. jak je nataženo vedení, je odlišná od síťové topologie, kterou vidí výpočet
    - Musí existuje bijektivní funkce, která mapuje mezi fyzickou a síťovou topologií (např. tabulka)

SW pohled

Alokování uzlů

- 1 proces na 1 uzel
    - Např. pevně daná u paralelního počítače
    - 1 proces dokáže plně využít celý uzel, takže nemá smysl jich na jednom uzlu spouštět několik
    - OS uzlu neumí spustit více jak jeden proces najednou
- Potenciálně nula až několik procesů na jeden uzel
- Přidělení celé sítě pro jeden výpočet - celkový čas výpočtu je pak dán:
    - Dobou k zavedení programů, spuštění procesů a distribuce dat do uzlů
    - Vlastním výpočtem
    - Získáním výsledků z uzlů
- Přidělení části sítě jednomu výpočtu
- Několik paralelně běžících výpočtů
    - Na jednom uzlu může běžet několik procesů
    - Nelze se spoléhat na odvozená urychlení, protože ta nepočítala se zátěží, kterou vygeneruje neznámý kód
    - Nehodí se pro synchronní/lock-stepped algoritmy – na společném uzlu by dva spolupracující procesy na sebe musely čekat dobu výpočtu jednoho kroku

Identifikace procesů

- Jedinečná ID procesů - může mít další význam (např. určení farmera)
- Interakce send/receive (vše ostatní je na nich postaveno)
- Podle přidělení na uzly:
    - 1 uzel – 1 proces
    - Více procesů na uzlu
    - Více procesů na uzlu a procesy mohou migrovat (tabulka umístění procesů) - adresování může zajišťovat middleware

Komunikační schéma

- Fyzická topologie = jak je to sdrátováno
- Síťová topologie = jak vidí fyzickou topologii software
- Virtuální topologie = komunikační vazby procesů
- Ideálně 1:1 (aby docházelo k nejmenším zpožděním)

Virtuální topologie

- může **mít:**
    - pravidelnou strukturu - mřížka, hvězda (farmer-worker)...

    - nepravidelnou strukturu
- může **být:**
    - statická - např. mřížka
    - dynamická - procesy mohou procházet různými fázemi, mohou vznikat i zanikat
- popis orientovaným grafem

11. Rozdíly mezi PVM a MPI.

Status: potřebuje revizi, mírně se liší od původní otázky

PVM a MPI se používají v prostředí s distribuovanou pamětí

Hlavní rozdíly

- PVM – vznik v prostředí heterogenní sítě, MPI navržen spíše pro clustery – homogenní prostředí (u obou je ale běh možný v heteregonním prostředí)
- MPI – jednodušší a efektivnější abstrakce na vyšší úrovni
- MPI nabízí bohatší možnosti pro přenos zpráv (např. plně duplexní `sendrecv`, perzistentní komunikace, každý pošle každému - `MPI_Alltoall`)
- MPI se už v návrhu snaží o omezení kopírování paměťových bloků
- PVM se nestará o topologii, MPI podporuje logické komunikační topologie
- MPI nemá žádný config jako PVM
- MPI se vyhýbá nízkoúrovňovým rutinám kvůli přenositelnosti
- MPI – možnost definovat vlastní datové typy pro snížení režie při posílání velkého množství dat (vs. PVM – vícenásobné volání `pvm_pack`)
- MPI – podpora souborových operací (soubor se rozdělí na 1-N bloků, čtení a zápis jako zasílání zpráv)

PVM (Parallel Virtual Machine)

- Univerzální výpočetní model pro heterogenní distribuované výpočetní prostředí
- v PVM se musí provádět operace `PVM_initsend`, PVM_PK* apod.
- PVM umožňuje především provádět distribuované výpočty v heterogenním prostředí (různé architektury)
- PVM se nestará o topologii, MPI podporuje logické komunikační topologie.
- Programátor PVM může využít funkci `PVM_Config` aby např. určil, kde spustit další proces – MPI nic takového nemá.
- PVM programátorovi umožňuje, aby se do systému napojil pomocí nízko úrovňových rutin, MPI se tomu vyhýbá kvůli vyšší přenositelnosti.

Základní funkce

- Probíhá pomocí zasílání zpráv

  `pvm_spawn(char *task, char **argv, int flag,char *where, int ntask, int *tids)`
- Spustí několik procesů podle zadaného programu
- Procesy se po vytvoření neznají -> proces, který je vytvořil je musí seznámit
  - numt –počet úspěšně spuštěných procesů(návratová hodnota)
  - task –spustitelný soubor
  - argv –parametry
    - PvmTaskDefault – PVM si rozhodne, kde spustit
    - PvmTaskHost – where bude obsahovat adresu, kde spustit
    - PvmTaskArch – where bude obsahovat typ architektury/platformy
    - Existují i další jako PvmTaskDebug, PvmTaskTrace, PvmMppFront, PvmHostCompl
  - flag – možnosti spuštění
  - where – kde spustit proces, ignoruje se, pokud nejsou příslušně nastaveny možnosti spuštění
  - ntask – počet procesů ke spuštění
  - tids – identifikátory spuštěných procesů

pvm_initsend(int encoding) – nastavení kódování (defaultně se používá PvmDataDefault)

`int pvm_pkint( int *ip, int nitem, int stride )` – vloží data do bufferu

`int pvm_send(int tid, int msgtag)` – pošle data procesu (TID)

`int pvm_recv(int tid, int msgtag)` – přijme data od procesu (TID)

`pvm_upkint( int *ip, int nitem, int stride)` – rozbalí data

MPI (Message Passing Interface)

- MPI je postaveno na PVM
- Knihovna pro podporu paralelních výpočtů v systémech s distribuovanou pamětí, vazby (interface) má pro programovací jazyky C a Fortran.
- Proti PVM se jedná o programovací prostředek vyšší úrovně, vztah je asi jako mezi jazykem symb. adres (odpovídá PVM) a vyšším programovacím jazykem (odpovídá MPI), tedy:
  - v PVM jde naprogramovat "skoro cokoliv", ale dá to velkou práci a program pak nejspíš nebude přenositelný,
  - dominantní aplikací pro MPI jsou numerické výpočty s regulárními daty (vektory či matice s číselnými prvky), přičemž pro takové aplikace je programování relativně pohodlné a program je dobře přenositelný do jiné instalace MPI
- MPI je primárně míněno (na rozdíl od PVM) pro homogenní výpočetní prostředí (tj. cluster stanic se společnou administrací, superpočítač jako N-Cube, ap.), takže poskytuje prostředky i pro "synchronní" algoritmy (tj. takové, kde se předpokládá přibližně stejná rychlost běhu jednotlivých procesů) a odpovídající statické rozdělení práce mezi procesy.
- MPI primárně využívá SPMD model paralelního výpočtu, tj. vyrobí se (na rozdíl od PVM) jen jeden "exe" soubor programu a ten se zavede do zvoleného počtu (dále N) procesorů. V každém procesoru běží jen jeden proces. Všech N procesů tudíž běží podle téhož programu, zjistí si své číselné ID a podle toho odliší svou činnost. V zásadě je tudíž realizovatelný i MPMD model výpočtu (třeba ve verzi farmer-workers, přičemž v programu je přepínač podle ID, jednu větev realizuje proces s číslem třeba 0–farmer, druhá větev (jiná čísla než 0) je pro procesy typu worker).
- Komunikace procesů je asynchronní message-passing s přímým adresováním přes číselné ID procesu. Existuje mechanismus skupin procesů (viz dále tzv. komunikátory) a možnost broadcastu zprávy ve skupině procesů. Zprávy není na rozdíl od PVM třeba pracně "pakovat". MPI má svoje "primitivní datové typy" a z nich lze skládat "strukturované typy", sloužící ovšem jen pro účely komunikace (tj. zjednodušený popis toho, co má přijít do zprávy – lze srovnat s náročností "pakování" zprávy v PVM).
- Existuje sada funkcí pro tzv. *"globální operace"*, tj. operace nad daty, jejichž instance jsou "rozprostřeny" ve všech procesech výpočtu. Realizace takových operací v PVM se musí pracně rozepsat do primitivnějších operací `pvm_send()` a `pvm_recv()`.

Základní funkce

Je jich 6, přičemž už umožňují napsat jednodušší aplikaci. První čtyři z nich je třeba použít v každém MPI programu. Všechny funkce vrací celočíselný kód úspěšnosti provedené operace, přičemž symbolická hodnota MPI_SUCCESS znamená úspěch. Přehled základních funkcí v C-syntaxi:

`int MPI_Init (int* argc, char*** argv)`

- Inicializace MPI výpočtu, argc a argv jsou argumenty hlavního programu.

`int MPI_Comm_size (MPI_Comm comm, int* adr_size)`

- Zjištění počtu procesů, počet se dosadí do proměnné odkazované parametrem adr_size. MPI_Comm je typ "komunikátor", pokud při volání dosadíme za parametr comm hodnotu MPI_COMM_WORLD, zjišťujeme počet všech vytvořených procesů aplikace N.

`int MPI_Comm_rank (MPI_Comm comm, int* adr_rank)`

- Zjištění čísla procesu v rámci "komunikačního světa" comm. Čísla jsou v rozmezí 0 až M-1, kde M je "rozměr komunikačního světa" reprezentovaného komunikátorem comm (M = N, pokud dosadíme za comm hodnotu MPI_COMM_WORLD).

```
int MPI_Finalize (void)
```

- Ukončení výpočtu v MPI, provádí každý proces.

```
int MPI_Send (void* adr_buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm)
```

- Odeslání zprávy s typem tag v komunikačním světě comm procesu dest. Odesílá se zpráva z bufferu buf obsahující count položek typu datatype.
- Čili buffer pro zprávu je na rozdíl od PVM kdekoliv v datech programu (zadá se adresa příslušného pole). Za datatype se dosazují buď primitivní typy MPI, například MPI_INT, MPI_DOUBLE, MPI_CHAR, nebo strukturované typy vytvořené z primitivních (viz dále).

```
int MPI_Recv (void* adr_buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm, MPI_Status *adr_status)
```

- Blokující příjem zprávy. Parametry mají analogický význam jako u MPI_Send. Navíc je parametr adr_status, odkazující kam se má uložit status příjmu zprávy (výstupní parametr). Status obsahuje položky: status. MPI_SOURCE -od koho zpráva přišla a status.MPI_TAG -jakého typu zpráva přišla. Dosadí-li se za source hodnota MPI_ANY_SOURCE a za tag MPI_ANY_TAG, přijme se jakákoliv zpráva a z uvedených položek výstupního parametru status se dá zjistit co to vlastně přišlo.

Globální operace MPI

- Globální operace jsou operace, do kterých jsou zapojeny všechny procesy patřící do téhož "komunikačního světa" (tj. jedním parametrem funkcí pro globální operace je příslušný komunikátor). Funkci globální operace volají všechny zúčastněné procesy (což je pochopitelné, protože typicky procesy běží podle téhož programu), přičemž jejich činnost se v obecném případě liší podle čísla procesu.
- Globální operace v PVM nejsou (kromě broadcastu) a musely by se pracně rozepsat do posloupnosti operací send() a receive().
- Globální operace MPI lze rozdělit na tři skupiny – synchronizace, přesuny dat a redukční operace.

Synchronizace

Každý komunikátor mj. realizuje bariéru, na které se mohou všechny jeho procesy synchronizovat voláním funkce
`int MPI_Barrier (MPI_Comm comm)`

- Volání této funkce je tedy blokující a výpočet každého procesu pokračuje následujícím příkazem až poté, kdy se na bariéře "sejdou" všechny procesy patřící do "komunikačního světa" comm.

Přesuny dat

- Jedná se o operace s charakterem broadcastu, shromáždění či rozptýlení dat a tzv. redukční operace.

```
int MPI_Bcast (void* adr_buf, int count, MPI_Datatype datatype,
               int root, MPI_Comm comm)
```

- Operace broadcast. Má v zásadě stejné parametry jako MPI_Send(). Proces s číslem root vysílá, ostatní zprávu přijímají (čili root používá buffer jako vysílací, ostatní jako přijímací). Proti send() chybí tag – pro *"globální"* komunikační operaci nemá význam – všichni aktéři komunikace prochází tímtéž bodem programu a tudíž vědí, *"o co při komunikaci jde"*.

```
int MPI_Gather (void* adr_inbuf, int incnt, MPI_Datatype intype,
                void* adr_outbuf, int outcnt, MPI_Datatype outtype,
                int root, MPI_Comm comm)
```

- Proces s číslem root shromažďuje data od všech procesů včetně sebe. Čili všechny procesy vysílají data (count položek typu intype) z bufferu inbuf a proces root je zapisuje do bufferu outbuf -samozřejmě je zapisuje v pořadí podle čísel vysílajících procesů(tj. nikoliv tak, jak zprávy došly). Parametr outbuf (a též outcnt a outtype) využije jen proces root. Za incnt a intype se normálně dosadí stejné hodnoty jako za outcnt a outtype.

```
int MPI_Scatter (void* adr_inbuf, int incnt, MPI_Datatype intype,
                 void* adr_outbuf, int outcnt, MPI_Datatype outtype,
                 int root, MPI_Comm comm)
```

- Inverzní operace k Gather(), tj. proces s číslem root "rozptyluje" data z bufferu inbuf všem ostatním procesům včetně sebe. Vstupní buffer musí obsahovat M částí dat (obecně různých, jedna část je pole count položek typu intype), přičemž i-tá část se pošle do bufferu outbuf procesu s číslem i. Parametr inbuf (a též incnt a intype) využije jen proces root. Za incnt a intype se normálně dosadí stejné hodnoty jako za outcnt a outtype.

Redukční operace

Redukční operace má M operandů umístěných ve vstupních bufferech komunikujících procesů. Výsledek operace se zapisuje do výstupního bufferu buď jednoho určeného procesu (root) nebo všech procesů.

```
int MPI_Reduce (void* adr_inbuf, void* adr_outbuf, int count, MPI_Datatype datatype,
                MPI_Op op, int root, MPI_Comm comm)
```

Za parametr ***op*** se dosazuje typ realizované operace, kde použitelné symbolické hodnoty typu jsou

- `MPI_MAX`, `MPI_MIN` (maximum, minimum)
- `MPI_SUM`, `MPI_PROD` (součet, součin)
- `MPI_LAND`, `MPI_LOR`, `MPI_LXOR` (logické operace)
- `MPI_BAND`, `MPI_BOR`, `MPI_BXOR` (logické operace se všemi bity)

Operandy jsou ve vstupních bufferech procesů, výsledek je ve výstupním bufferu procesu root.

```
int MPI_Allreduce (void* adr_inbuf, void* adr_outbuf, int count,
                   MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

- Funguje jako předchozí operace, ale výsledek dostanou do výstupního bufferu všechny zúčastněné procesy, tudíž chybí parametr root, který pak nemá smysl.

Komunikátor

Komunikátor je interní objekt MPI (typ MPI_comm, jedná se o typ tzv. *"handle"*, čili jakéhosi zobecněného ukazatele reprezentujícího komunikátor).

Komunikátor reprezentuje skupinu komunikujících procesů a komunikační kontext této skupiny.

Dále komunikátor slouží pro paralelní členění programu, či jinak řečeno pro realizaci modelu MPMD (funkční paralelismus), kdy se procesy aplikace rozdělí na skupiny (reprezentované různými komunikátory) a každá skupina *"dělá něco jiného"* a její procesy *"se vybavují jen mezi sebou"*. Čili uvnitř skupiny procesů (komunikátor) funguje datový paralelismus, kdežto mezi skupinami procesů funguje funkční paralelismus.

Základní funkce nad komunikátory jsou (podrobnosti viz literatura):

`MPI_Comm_rank()`
- vrací počet procesů komunikátoru, základní funkce MPI – viz dříve v části 2

`MPI_Comm_dup()`
- vytváří duplikát komunikátoru

`MPI_Comm_split()`
- *"štěpí"* kpomunikátor na dva jiné, čili rozděluje jednu skupinu procesů na dvě jiné

`MPI_Comm_free()`
- uvolňuje (likviduje) komunikátor (samozřejmě nikoliv procesy, které komunikátor reprezentuje)

`MPI_Intercomm_create()`
- vytváří tzv. *"interkomunikátor"* jakožto prostředek komunikace mezi dvěma skupinami procesů.

Příklad PVM

```
#include <stdio.h>
#include <pvm3.h>

int main() {
  int mytid;
  mytid = pvm_mytid();
  printf("My TID is %d\n", mytid);
  pvm_exit();
  return 0;
}
```

Příklad MPI

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[]){
  int pocet = 0;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(COMM_WORLD, &pocet);
  MPI_Comm_rank(COMM_WORLD, &moje_id);
  if (moje_id == 0)
    printf("Pocet procesu: %d",pocet);
  MPI_Finalize();
}
```

12. Přidělování práce v prostředí s distribuovanou pamětí, možnosti urychlení výpočtu a přiřazení procesů na jednotlivé uzly.

Status: potřebuje revizi, mírně se liší od původní otázky - zpracováno dle přednášek (B Advanced a 7b Distributed Models) a dle PPR_NOVE_2014.pdf

Přidělování práce v prostředí s distribuovanou pamětí

- Faktory ovlivňující rychlost výpočtu
  - Virtuální topologie, komunikační schéma, distribuované aplikace
  - Prezentovaná síťová topologie
  - Fyzická topologie sítě
  - Výkonnost jednotlivých uzlů v síti
  - Výpočetní model distribuované aplikace - každý proces může běžet podle vlastního programu
- Řešení obecně
  - Umístit procesy na uzly sítě takovým způsobem, aby běžely co nejrychleji a zároveň bylo
  - komunikační zpoždění co nejmenší - může se jednat i o protichůdné požadavky
  - Zatížení a dostupnost uzlů se může měnit v čase
  - Jsou tři typy úloh
    - **S konečným časem výpočtu** – distribuovaná aplikace má jenom něco spočítat a pak skončí
    - **S teoreticky nekonečným časem výpočtu** – distribuovaná aplikace poskytuje službu (neplatí, že uzel musí být zcela vytížen výpočtem po celou dobu)
    - **Processing While in Transit** – výpočet se nad daty provádí během jejich přenosu např. Active Network

Možnosti urychlení

MPMD

- Ve všech uzlech není stejný program
- Každý proces má svoje data

SPMD

- Do všech uzlů se zavede stejný program
- Do uzlů se zavedou specifická data procesů
- Každý proces dokáže zjistit svoje ID a z něj určí sousedy a svůj díl dat
- Jeden z procesů je řídící - obvykle ten první vytvořený
  - Mívá ID 0, ale konkrétně to závisí na sw
  - Zpracovává dílčí mezivýsledky
- V homogenním distribuovaném prostředí se zjistí celkový objem dat, počet spuštěných procesů a práce se přidělí najednou - např. cluster z identických stanic a MPI
- V heterogenním prostředí je lepší využít dynamické přidělování práce
  - Uzly nemusejí být stejně výkonné
  - Ale i v případě, kdy uzly nejsou využívány jednouživatelsky
  - Např. model farmer-workers, kdy farmáři udělíme čestný titul identický program -> bude k tomu všemu ještě makat jako worker
- Urychlení u Farmer-Worker v distribuovaném prostředí, pokud farmer jen kompletuje dílo od workerů a pokud zanedbáme odeslání a příjem zprávy je přibližně S ~ N - 1, tj. přibližně lineární
- F-W se hodí tehdy, když

    - Datový objem zpráv je malý
    - Výpočet je v porovnání s objemem zprávy náročný
    - Např. nemá smysl posílat pole integerů k součtu na jiný uzel, protože u dnešních i86/64 je doba operace mov zhruba stejně časově náročná jako add, muselo by se tedy vykonat spoustu operací navíc => zpomalení
- V heterogenním prostředí se realizuje automatický loadbalancing a urychlení pak závisí na velikosti objemu dat ke zpracování jedním procesem, je třeba najít ideální objem dat, který příliš nezpomaluje a zároveň poskytuje dobré urychlení
    - Příliš malé objemy dat nevedou k urychlení
    - Příliš velké objemy dat jsou sice rychlejší než příliš malé, ale zase se zbytečně čeká na procesy, které z nějakého důvodu počítaly pomaleji např. vlivem hw, nebo jiného sw, či režie OS, ...
- Ideální velikost posílaných dat?
    - Závisí na výpočetním výkonu uzlů
    - Měla by uzel na zaměstnat tak dlouho, aby bylo možné úkolovat uzly v době, kdy ostatní počítají
        - V první „vlně“ přidělit náhodné velikosti dat od jednoho až dvou násobků objemu zprávy - tím se na nějakou dobu zabrání konvergenci, kdy bude komunikační linka používána všemi procesy a dojde k eliminace komunikačního zpoždění překrytím výpočtem
            - Kromě neblokujících komunikačních operací
    - Data by se měla spočítat v nějakém přiměřeném čase, aby se na poslední proces nečekalo „celou věčnost“

MPSD

- Step-locked
- „pásová“ výroba
- Části dat by měly být stejně velké ne podle objemu dat, ale výpočetně
- Problémem může být kapacita přenosových kanálů
    - Objem dat může být příliš velký a tak uzel může nějakou dobu čekat na data
    - Ideálně se v i-tém kroku
        - Počítají data
        - Data z i-1 kroku se posílají dalšímu uzlu v řadě
        - Přijímají se data, která se budou počítat v i+1 kroku => překrytí doby komunikace dobou výpočtu
        - => eliminace komunikačního zpoždění, pokud se data déle počítají, než přijímají - pokud ne, zmenšit objem posílaných dat
- N – počet úkolů
- Pošleme-li objem vstupních dat k nekonečnu, pak je urychlení N

Přiřazování uzlů na procesy

Load-Sharing

- Předpokládá se prostředí pracovních stanic, které nemusejí být vždy plně vytíženy
- Jeden uzel se vyhradí jako master, kde se spustí aplikace - ostatní uzly slave
- V okamžiku, kdy je master vytížen na maximum, zkusí se vyhledat nevytížený uzel

Červ

- jednotlivé procesy se mohou replikovat na nevytížených uzlech
- červ se skládá z několika segmentů – procesů
- počet segmentů je buď **pevně** stanoven, nebo se při pokročilejší implementaci stanovuje **dynamicky** podle okolností

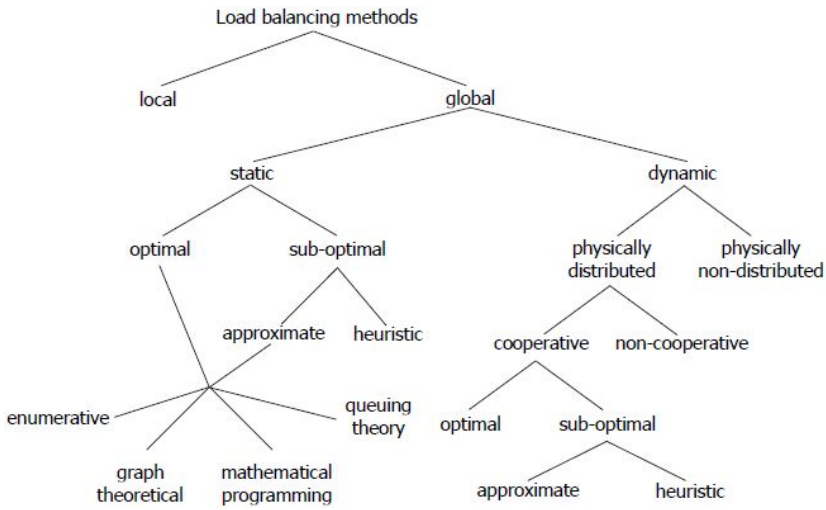
- nápadně připomíná šíření virů
- červ musí být věrohodný pro uživatele pracovních stanic
- komponenty červa
  - inicializační kód ke spuštění master
  - inicializační kód ke spuštění slave
  - výpočetní program
- životní cyklus červa
  - nalezení nevytíženého uzlu
    - žádný uzel nezná globální stav sítě a proto se každý segment musí postarat o nalezení volné stanice sám za sebe
    - lze hledat například pomocí broadcastu
    - hodila by se synchronizace, protože několik segmentů může soupeřit o stejný uzel
  - uvolnění uzlu
    - segment se musí postarat, aby uzel byl zase viditelný jako dostupný i pro ostatní
  - kontrola růstu
    - čím větší červ, tím vyšší rychlost výpočtu, ale vznikají další problémy (rychlost nemusí růst lineárně, synchronizace, stabilita)

Condor

- snaží se o fér využívání všech uzlů
- pokud některý uzel selže, proces se restartuje jinde
- arbitr, který rozhoduje o tom, kde se spustí proces
  - sám hledá použitelné uzly
  - centralizované místo - možnost selhání
- checkpoints
  - procesy lze relokovat za běhu distribuované aplikace - používá se ukládání obrazu procesu v paměti, ne rekonstrukce stavu
  - checkpoint se ukládá periodicky - pokud selže výpočet, použije se poslední checkpoint
  - pre-emptivnost procesů je implementována pomocí checkpointů

Load-Balancing

- na rozdíl od load-sharingu se předpokládá, že celá síť je dostupná pro výpočet
- existuje několik různých metod, které se liší použitelností, účinností, náročností na zdroje (paměť, procesor, ...), přesností, spolehlivostí, ...



- statické
  - výpočet přiřazení procesů na uzly je proveden ještě před spuštěním distribuované aplikace
    - výpočet může běžet libovolně dlouho, abychom dosáhli požadované přesnosti předpovědi

    - nelze reagovat na dynamické změny v prostředí
    - vyžadují předem spoustu informací o chování sítě a aplikace (např. kom. zpoždění, doby běhu procesů)
        - nereálné požadavky nelze splnit - vliv na přesnost a tedy i rychlost výpočtu
- dynamické
    - výpočet přiřazení procesů na uzly sítě se provádí za běhu distribuované aplikace
        - výpočet se odehrává v reálném čase a nemůže si proto dovolit konzumovat příliš mnoho zdrojů
    - umí se vyrovnat s dynamickými změnami - procesy musí umět pre-empci
    - potřebné informace lze zjistit až za běhu aplikace nebo si jich část vyžádat předem
- pre-emptivní
    - procesy lze přerušit během výpočtu a přemístit je na jiný uzel, aby bylo možné kompenzovat změny v síti - např. některý z procesů mohl skončit svoji činnost, nebo se odebral do dlouhodobého wait-stavu
    - pokud tuto vlastnost procesy nemají, na změny lze reagovat až při vytváření
- centralizované
    - mají jeden centrální prvek, arbitr, který rozhoduje o rozdělování zátěže na jednotlivé uzly
    - centrální prvek je slabé místo, co se stane, když selže?
    - centralizované správa vyžaduje komunikaci jednoho uzlu se všemi - možnost přetížení
    - arbitr má přehled o známé síti a proto lze očekávat, že dokáže zátěž rozdělovat celkem efektivně bez rizik, která jsou jinak spojena s distribuovaným principem
- distribuované
    - rozhodování o rozdělování zátěže provádí několik, až všechny procesy
        - mohou být distribuovány na několik uzlů
    - když jeden selže, nic se neděje, pokud ho výpočetní model aplikace nutně nepotřebuje k životu
    - procesy, které provádějí rozhodování mohou být buď specialisté, anebo to mají jako „vedlejšák“ ke své hlavní činnosti
- adaptivní
    - síť prochází změnami během výpočtu – mění se stav uzlů
    - adaptivní metody berou do úvahy i několik předchozích stavů při rozhodování o přidělení zátěže
- kooperativní
    - každý proces se může rozhodovat buď sám za sebe, nebo může na rozhodnutí spolupracovat s ostatními
    - přímo – procesy spolupracují nad konkrétním rozhodnutí
    - nepřímo – procesy dávají informace o svých rozhodnutích k dispozici ostatním a ty je použijí při svých rozhodnutích
- sender-initiated
    - v okamžiku, kdy je uzel zatížen přes určitou mez, začne vyhledávat jiné uzly, kam by přemístil část své zátěže
    - je to režie navíc, protože čas potřebný na vyhledávání nových uzlů mohl být použit na běh procesů
- receiver-initiated
    - v okamžiku, kdy zátěž uzlu klesne pod určitou mez, začne vyhledávat jiné uzly, odkud by mohl převzít jejich zátěž
    - režie se projeví zvýšenou komunikací, uzel má dost volného výpočetního času, který může alokovat pro vyhledávání zátěže

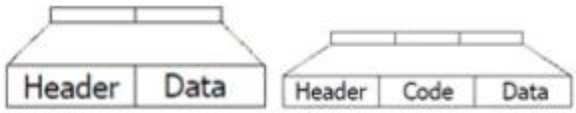
Load Redistribution

- load-balancing tradičně měří zátěž v počtu procesů, což není zrovna to nejlepší
- metoda vyžaduje pokročilou síťovou architekturu jako jsou Aktivní sítě

- metoda je naprosto nezávislá na konkrétní aplikaci a síťové topologie, výkonnosti uzlů, atd.
    - vše si dokáže zjistit sama za běhu
    - programátor ji jenom použije jako knihovnu a napíšu podporu pro migraci procesů
- Princip
    - Procesy se rozhodují samy za sebe,
    - Periodicky zkoumají své okolí
    - Nepřímo spolupracují => chovají se jako kolonie organismů, které mají jednoduchá pravidla, ale dohromady vytvářejí inteligentní chování a dokáží se adaptovat na změny
    - Metoda se nesnaží dosáhnout dobře vyváženého stavu hned na první pokus, ale postupně
- Síťové prostředí
    - Procesy injektují speciální capsules – PerformanceScouts
    - Taková capsule nejen, že zjistí, jak vypadá síťové okolí uzlu z hlediska topologie, ale také výkonnost, zatížení, komunikační zpoždění a další informace o navštívených uzlech
    - Takto zjištěné informace se pak využijí při hledání uzlu, kam by mohlo být vhodnější odmigrovat proces, protože by tam mohl běžet rychleji
- Komunikace
    - Pokud je distribuovaná aplikace dobře napsaná, pak procesy tvoří skupiny, které mezi sebou komunikují - Communication cluster
    - Může být tolik clusterů, kolik je procesů
    - Clustery jsou definovány virtuální topologií – komunikační schéma distribuované aplikace
    - Vyplatí se držet procesy z daného clusteru blízko sebe, aby se snížilo komunikační zpoždění
    - Čas od času může proces komunikovat i s jiným procesem, který do jeho clusteru nepatří - ideální cluster
    - Máme tři procesy s komunikací A – B, B – C
        - Existují tři clustery- A, B; B, C; A, B, C
    - Poskytnutím komunikačních funkcí lze trasovat komunikaci a odhalit tak komunikační clustery
    - Potencionálně se nemusí odhalit celá virtuální topologie, protože tok zpráv může být závislý na vstupních parametrech
    - Počítáme-li s migrací procesů, je výhodné začít používat adresování nezávislé na adresách uzlů a zjednodušit tak život programátorovi
    - Zajištění integrity, konzistentnosti a adresovatelnosti
- Výkon a rychlost běhu procesu
    - Lze změřit, kolik procesorového času proces konzumuje na lokálním uzlu a následně porovnat, zda by mu jiný uzel mohl poskytnout více procesorového času
    - Pomocí benchmarku se dá porovnávat výkon na různých typech procesorů
    - Lze identifikovat několik uzlů, kde by proces mohl běžet rychleji - vybere se ten, kde dojde k největší minimalizaci komunikačního zpoždění
- Synchronizace
    - Každý proces se rozhoduje sám za sebe, ale nepřímo spolupracuje s ostatními pomocí blackboards, které jsou podporovány přímo aktivním uzlem
    - Grade32, SAN - Smart Active Node - nástupce Grade32
- Rizika
    - Masová migrace
    - Oscilace - proces se může pohybovat po síti, aniž by cokoliv spočítal
        - Zavedení kreditů - dokud toho dost nenapočítá, nikam se migrovat nebude
    - Zbytečné migrace procesů
- Pokud mají dva uzly přibližně stejné množství procesorového času, který mohou nabídnout procesu, může dojít k migraci, aniž by došlo k urychlení výpočtu

Aktivní sítě

- stvořil Pentagon pro vyřešení nedostatků IP protokolu
- např. Any-Cast = metodologie pro adresování a routování, kdy jsou datagramy od jediného odesilatele routovány uzlu, který je topologicky nejblíže v dané skupině potenciálních příjemců (ačkoliv může být zasláno vícero uzlům, pokud mají stejnou adresu). Rozdíl od multicastu, který posílá všem ze skupiny najednou (a vždy), broadcastu, který zasílá jednoduše všem.
- Any-Cast je až v IPv6, aktivní sítě mají PAMcast – Programmable Any-Multicast – služba pro doručování zpráv, která generalizuje jak anycast tak multicast , která doručuje zprávy M z N příjemců, kde 1 <= M <= N
- IP
    - Dvojice vysílající a příjemce
    - Vysílající odešle paket na konkrétní adresu, včetně portu, kde předpokládá příjemce
    - Pokud tam není, paket se zahodí a vysílající zjistí chybu až timeoutem
- Aktivní síť
    - Paket, nazývaný capsule, je asociován s kódem, který se spustí na každém uzlu, kterým capsule prochází - vždy je přítomný příjemce
    - Kód může manipulovat s daty, vlastnostmi (cíl, TTL, atd.) a vykonávat další uživatelsky definované činnosti
    - Proces se označuje termínem aktivní aplikace
        - Distribuovaná aktivní aplikace se skládá z několika aktivních aplikací, které mohou injektovat capsule a zároveň capsule může injektovat aktivní aplikace



IP paket Paket aktivní sítě - capsule

- Komunikační model sám-sobě
    - Je možné injektovat capsuli do sítě, aby nasbírala potřebná data a pak je předala procesu, který ji injektoval
        - U IP by bylo nutné mít dopředu na každém uzlu, který by capsule mohla navštívit, spuštěný specializovaný proces
- Migrace procesů
    - Migrující proces změní svoji síťovou adresu, ale ještě ji nedal na vědomí ostatním procesům
    - Informaci o své nové síťové adrese zanechal na uzlu, odkud migroval
    - Capsule, která má doručit data, dorazí na uzel, odkud proces odmigroval, tam ho nenajde, ale použije svůj kód, aby si přečetla novou adresu a pouze změní svůj cíl
    - Ostatní procesy si mohou aktualizovat záznamy až později – lazy update, u IP je nutné vyřešit předem
- V aktivní síti je zapotřebí standardizace pouze dvou věcí
    - Programového kódu
        - Kód vykonává Execution Environment (EE) - na jednom uzlu může být několik EE
    - Code distribution protocol
    - Vše ostatní je pak už aplikačně specifické

13. Systémy reálného času.

Status: nová otázka - zpracováno dle přednášek (C Rela Time) a dle PPR_NOVE_2014.pdf

- **Non-Real Time**
  - Vlákna nemají stanovenu žádnou dobu, deadline, do kdy musí dokončit výpočet
  - A to ani tehdy, když od nich požadujeme rychlou odezvu
    - Jak napíšete slovo ve Wordu, už aby ho zároveň zkontroloval a odstranil překlepy
- **Real Time**
  - Dokončení výpočtu ve stanoveném termínu je kritické
  - Termín musí být dodržen bez ohledu na zátěž systému
  - Brzdy v autě, vojenské systémy, podpora života, jaderné zařízení, ...
    - Ale i řízení výroby a mobilní telefony
- **Hard Real Time**
  - Dokončení výpočtu po termínu se považuje za chybu a výsledek za bezcenný – strict deadline
  - Nedodržení termínu může vést k celkovému selhání systému - např airbag, řízení motoru, jaderné zařízení
  - Jsou vyžadovány tam, kde hrozí příliš velké škody v případě selhání systému
- **Soft Real Time**
  - Překročení termínu se toleruje, systém reaguje zhoršenou kvalitou poskytovaných služeb
    - Vypadne pár snímků, přílet letadla se dozvíte s několika sekundovým zpožděním
- Výkonnost
  - Nejde o to vypočítat co nejvíc, ale vypočítat to včas - real time systém nejsou vysokovýkonnostní
  - pokud je úloha schopna dodržet časové limity, není potřeba zvyšovat výkon
- Plánování
  - *Cyklický plánovač* - neosvědčil se, velká režie při přiřazování úloh, selhával
  - *Prioritní schéma* - RMA (odstraňuje nedostatky cyckl. plánovače)
- **Výběr algoritmu**
  - Je potřeba znát nejhorší možný čas výpočtu algoritmu (kdy vrátí výsledek)
    - Př. QuickSort - nejlepší O(N * logN) - nejhorší O(N^2)
    - oproti tomu HeapSort v praxi pomalejší, ale zaručeno O(N * logN), proto vhodnější
  - Další kritéria jako spotřeba paměti, možnost paralelizace nebo jiné speciální prodmínky

RMA

- Rate Monotonic Analysis
- Plánovací algoritmus systému reálného času – **Rate Monotonic Scheduling**
- vlákno se označuje jako úloha (task)
- Přiřazuje priority jednotlivým úlohám tak, aby stihly dokončit výpočet v termínu
  - Možným výsledkem je, že se zjistí, že to úloha nemůže stihnout
- **Periodická úloha**
  - Je opakovaně/periodicky runnable v pevně daných intervalech
    - Při korektní činnosti systému je spuštěna
      - Tj. systém neselhal – hard real time
      - Nedošlo ke zpoždění – soft real time
  - Perioda - časové rozmezí, kdy je úloha runnable
  - Deadline – začátek další periody
- **Priorita** - odvozena od délky periody
  - Čím kratší perioda, tím častěji běží => větší priorita úlohy

- Ratio grid - rozdělit interval <min_period, max_period> aby byl poměr mezi sousedy stejný: 1ms, 2ms, 4ms, 8ms...
- Ideálně tolik úrovní priorit, kolik je třeba - nemusí být vždy možné

- **Plánování**
  - Lui a Layland - přiřazování priorit je optimální v tom smyslu, že jeli množina úloh naplánovatelná tak, aby žádná z nich nepřekročila deadline, pak je naplánovatelná s rate-monotonic plánováním
  - V některých systémech mohou zajišťovat kritické úkoly úlohy s dlouhou periodou (nízkou prioritou) - řešením je rozdělit takovou úlohu do několika menších s kratší periodou
- **Naplánovatelný systém**
  - *Ci/Ti* je využití procesoru *i-tou* úlohou z *n* úloh
  - Lui a Layland dokázali, že

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n \cdot (\sqrt[n]{2} - 1)$$

  - *U* – využití procesoru
  - *n* – počet úloh
  - *Ci* – výpočetní čas úlohy
  - *Ti* – perioda úlohy

$$\lim_{n \to \infty} n \cdot (\sqrt[n]{2} - 1) = ln\ 2\ \approx 0,693147...$$

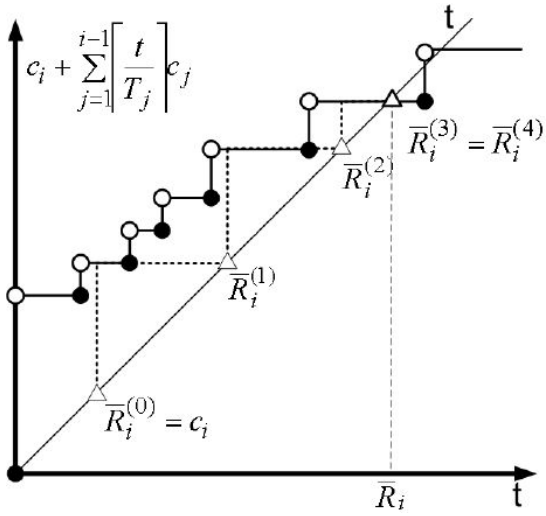
  - Udržíme-li zatížení procesoru n úlohami pod 70%, pak je možné naplánovat všechny úlohy tak, aby dodržely své deadlines
  - To neznamená, že není možné najít takové plánování, které bude mít stejnou vlastnost při větším zatížení systému
  - Zbývajících 30% mohou být non-real time threads
  - Zatížení 70% můžeme použít jako **rychlý test**, zda je systém **naplánovatelný** - pokud to s ním nevychází, pak lze použít Response Time Test
- **Response Time Test**
  - Lze použít, pokud nevychází test se 70%
  - Pro každou úlohu rekurzivně vypočítáme R_k (response time)• iteračně tak dlouho, dokud je rozdíl posledních dvou výsledků mimo nějaký rozsah

$$R_0 = \sum_{j=1}^{i} C_j$$

    - Úlohy jsou seřazené podle priorit, provede se součet všech úloh s vyšší prioritou

$$R_{k+1} = C_i \sum_{j=1}^{i-1} C_j \times ceil\left(\frac{R_k}{T_j}\right)$$

    - Algoritmus funguje tak, že se snaží navýšit výpočetní čas úlohy na základě času, který zaberou úlohy s vyšší prioritou
    - Iterace končí v okamžiku, kdy se výpočetní čas úlohy přestane navyšovat
    - jestliže R_k<*deadline* úlohy, pak úlohu lze naplánovat



- **Aperiodické a sporadické úlohy**
  - Zpracování událostí - např. hw přerušení
  - Inicializace
  - Zotavení se z chyby systému
  - Příkaz/požadavek operátora
  - **Soft deadline**
    - např. změna parametrů radaru
    - Dá se tolerovat zpoždění v reakci na příkaz operátora
    - Co nejmenší zpoždění je žádoucí, ale nikdy na úkor úloh s hard deadline
    - ***Aperiodická úloha*** - dokonce vůbec nemusí mít deadline
  - **Hard deadline**
    - např. pokyn pilota
    - Při některých manévrech může být zpoždění smrtelné
    - ***Sporadická úloha***
  - **Plánování**
    - V systému je několik serverů, které vykonávají aperiodické/sporadické úlohy
    - Servery jsou ***periodické úlohy*** a jsou plánovány podle pravidel ***RMA***
    - ***Sporadické úlohy*** jsou nejprve setříděny podle ***EDF*** (Earliest Deadline First)
- **Synchronizace – kritická sekce**
  - *Nebezpečí:* Úloha s nízkou prioritou může zablokovat úlohu s vyšší prioritou
  - Úloha s nízkou prioritou je v kritické sekci
  - Úloha s vysokou prioritou chce také do kritické sekce, ale je zablokována
  - Úloha se střední prioritou přeruší vykonávání úlohy s nízkou prioritou a tak zablokuje úlohu s vysokou prioritou => ***nestíhá se deadline, selhání systému***
  - **Inverze priorit**
    - Z úlohy s nízkou prioritou, která drží zámek, se dočasně stane úloha s vysokou prioritou
    - Úloha, která má normálně vysokou prioritou, dokončí výpočet o něco později
      - Většinou se to stihne, ale ne vždy a může to vést k závažným problémům
      - Např. úloha s vysokou prioritou může být kontrolní a pokud včas nezpracuje informace, může současný stav vyhodnotit jako selhání a resetovat celý systém (Mars Pathfinder)
  - **Zákaz všech přerušení**
    - Tj. i hodin
    - Jedna úloha je absolutním pánem systému
    - Nicméně, kritická sekce musí být krátká, jinak systém selže i tak
    - Používá se v malých systémech, nehodí se pro general-purpose

    - **Dědičnost priorit**
        - Úloha s nižší prioritou vykonává kritickou sekci
        - Úloha s vyšší prioritou se pokusí o vstup do kritické sekce
        - Úloha v kritické sekci dostane prioritu čekající úlohy s nejvyšší prioritou => žádná úloha se střední prioritou nezablokuje úlohu s nejvyšší prioritou
        - Může dojít k uvíznutí
        - Může vzniknout řetěz postupně blokovaných úloh
    - **Priority Ceiling**
        - Ví se, jaké zdroje chráněné kritickými sekcemi budou úlohy požadovat
        - Každá úloha běží s přidělenou prioritou, dokud nechce vstoupit do kritické sekce
        - Při vstupu do krické sekce dostane úloha nejvyšší prioritu ze všech, které tam kdy budou chtějí vstoupit
        - Alokace – úloha uzamkne zámek kritické sekce, jak si o něj požádá
        - **Nedochází k uvíznutí** - v okamžiku, kdy je v kritické sekci, neexistuje žádná jiná úloha, která by ji mohla přerušit
        - Stále však **může dojít k vyhladovění úlohy** s vyšší prioritou úlohou s nižší prioritou - příliš dlouho je v kritické sekci úloha s nízkou prioritou a zdrží úlohu s vyšší prioritou natolik, že ta nestihne deadline
- **Synchronizace – zprávy**
    - Možnost synchronizace, kdy nedojde k uvíznutí díky zámku kritické sekce
    - Mohou být generovány rychleji než zpracovány
    - Zpráva může obsahovat synchronizační informace
    - Většina OS zprávu kopíruje 2x - Z paměti odesílajícího do fronty zpráv spravované OS v jeho části paměti a z fronty zpráv OS do paměti příjemce
    - Může být pomalé pro RT - Lze zrychlit pouze převedením části paměti odesílajícího pod příjemce a předat pointer (úlohy s takovou praktikou ale musí počítat)
- **Správa paměti**
    - Kritická záležitost i z hlediska času
    - problém: malloc/free fragmentují, trvá než se najde vhodný blok paměti
        - nemusí se stihnout deadline
        - RTS může běžet klidně několik let bez restartu
    - Defragmentace - nedá se předpovědět, jak dlouho bude trvat a projevuje se náhodně - zvyšují riziko nestihnutí deadline
    - Dilema - odmítnout malloc aby se zamezilo defragmentaci i když je dostatek paměti? Nebo povolit občasné ubírání výkonu?
    - Řešení
        - Dva seznamy - volné a obsazené
        - Bloky paměti mají stejnou velikost, dále se nedělí, nefragmentují
        - Memory pool

RTOS

- Real Time Operaing System
- Sám o sobě nezaručuje, že výsledky budou vypočítány včas - to je úkol programátora, aby vytvořil správný program
- Umožňuje dodržet termíny
    - Obecně - Soft Real Time
    - Deterministicky - Hard Real Time
- Nejde o zpracování co nejvíce dat, ale
    - Co nejrychlejší reakce na různé události s dopředu známou dobou trvání

    - Minimální režie při přepínání úloh
- Plánování úloh
    - **Událostně řízené**
        - úloha se přepne pouze tehdy, když nastane událost s vyšší prioritou, než má běžící úloha
        - Kooperativní multithreading – úloha se po nějaké době dobrovolně vzdá procesoru
    - **Sdílení času**
        - tj. virtualizace procesoru
        - úlohy se přepínají nejenom událostmi, ale i podle hodin
        - Barrel Procesor
            - Hw garantuje, že v každém cyklu vykoná jednu instrukci z N běžících vláken jednou za N cyklů
            - Nulová režie přepínání vláken
            - I kdyby některé vlákno uvízlo, nebo zpracovávalo příliš přerušení, další vlákna běží dál ve stanovených časech
            - N je konečné a vysoká N jsou nákladná na design i na výrobu
- **Earliest Deadline First**
    - Vlákna jsou v prioritní frontě
    - Jakmile dojde k události jako vytvoření, či dokončení vlákna, fronta se prohledá a vyberou se vlákna s **nejbližším termínem** a z nich se pak vybere podle **priority**
    - Pokud nároky vláken nepřesahují 100% výkonu procesoru, EDF je umí naplánovat tak, aby se
    - všechny vykonaly včas , v opačném případě nelze určit, kolik vláken nestihne dodržet termín
- **Monotonic scheduling** - viz RMA

RTS bez RTOS

- Dostatečně malý projekt se může obejít bez velkého RTOS, potřebujeme-li jeho funkčnost v množství menším než malém
- př pro ledničku, kde jenom monitorujeme teplotu a rozsvěcíme/zhášíme žárovku, není RTOS nezbytně nutný
- Rozhodování - cena, velikost projektu, možná časem vlastní kód vytvoří vlastní RTOS

Java Real-Time System

- Java neumí
    - Používat striktně prioritní plánování vláken - tím pádem zámky Javy nemohou umět inverzi priorit, priority ceiling, atd.
    - RT vhodnou správu paměti - garbage collector představuje náhodné zatížení systému a pozastavování vláken, takže se nedá garantovat dodržení deadlines
- Proto vzniklo Real Time Specification for Java (RTSJ) - Java Real Time System a Timesys
    - Specifikuje minimální požadavky na správu vláken, různé modely plánování je možné doinstalovat do JVM
    - Část paměti lze vyloučit z aktivit garbage collectoru
    - Vybraná vlákna lze označit za nepřerušitelná garbage collectorem

14. Na co se může dál ptát

14.1. Co se děje když se zavolá return v main

- Když se linkuje program, přilinkuje se k němu runtime knihovna, která obsahuje inicializační kód programu, který mj. nastaví např. argc a argv pro funkci main
  - Obvykle se jmenuje crt0 („cr“ aka „C runtime“; „0“ aka „the very beginning“)
  - Vstupním bodem programu tedy není main, ale crt0, která volá main jako funkci
  - Main má tedy na vrcholu zásobníku adresu do crt0, kam se vrátí posledním příkazem return
  - crt0 pak po návratu z main zavolá službu OS „ukonči proces“
    - což opět nebrání programátorovi, aby tuto službu zavolal kdykoliv jindy
    - nicméně crt0 pak ještě může provádět deinicializaci