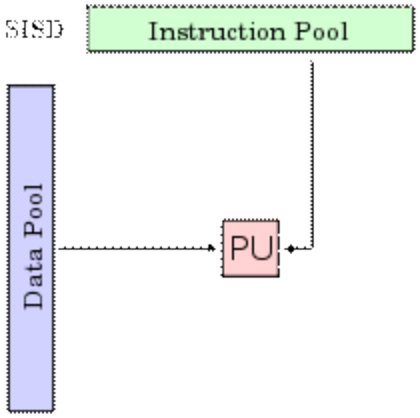
1. Modely SIMD, SPMD. MPSD, MPMD.

Wednesday, May 29, 2013 4:49 PM

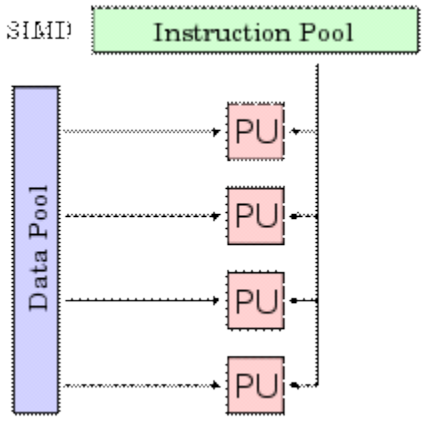
SISD (Single Instruction Single Data)

- Sekvenční výpočet, žádný paralelismus, např. MSDOS



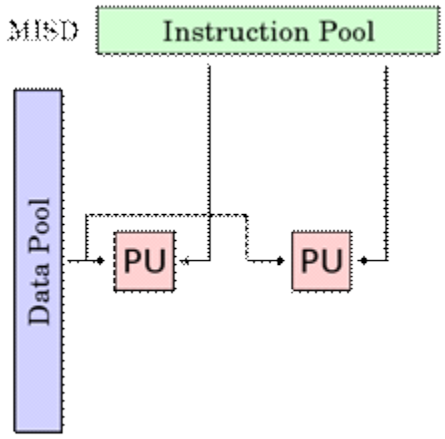
SIMD (Single Instruction Multiple Data)

- Datový paralelismus, např. vektorový paralelní počítač, maticové a vektorové výpočty na GPU (operace pro úpravu kontrastu v obrázku apod.), vektorová instrukce = jedna instrukce zpracuje několik dat najednou



MISD (Multiple Instruction Single Data) = MPSD (Multiple Program Single Data)

- Používáno pro výpočty odolné proti poruchám, několik systémů zpracovává ta samá data a musejí se shodnout na výsledku – např. letadla. Používá se také v pipeline architektuře, kde několik procesů zpracovává data v jednom datovém proudu, např. montážní linka. Urychlení u pipeline je N (rovno počtu fází).

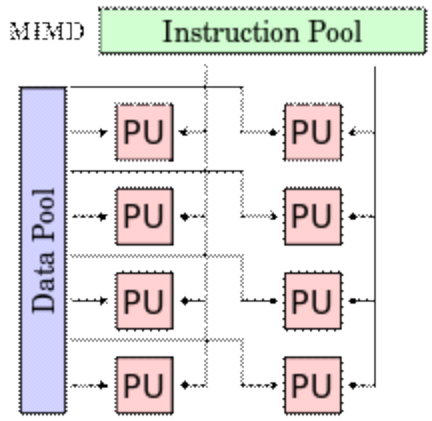


- Vlastnímu programování paralelní úlohy by měla předcházet fáze analýzy, ve které se rozhoduje o základním výpočetním přístupu, který se použije pro dekompozici výpočtu na jednotlivé složky - procesy.
- Navrhuje se, co bude řešit který proces/vlákno, dekompozice dat pro ně.
- Rozhodování nemusí být ovlivněno cílovou architekturou, protože uvedené modely jsou dostatečně obecné.
- **Flynnova taxonomie:**

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

MIMD (Multiple Instruction Multiple Data)

Několik procesorů zároveň vykonává různé instrukce nad několika různými daty. Procesory pracují buď asynchronně, nebo mají sdílenou paměť – o zajištění integrity se stará OS, nebo distribuovanou paměť – má lepší škálovatelnost.



SPMD (Single Program Mutiple Data)

- 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 zpracování velkých objemů dat, k procesů běží podle stejného programu a zpracovává strukturou stejné, ale hodnotově různé části dat.
- Příklady použití: Monte Carlo, iterační numerická řešení

MPMD (Multiple Program Multiple Data)

- 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 – např. distribuované simulace, systém spolupracujících komponent.

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

2. Paralelismus na úrovni instrukcí a predikce skoků.

Pipelining

- Rozdělení zpracování jedné instrukce mezi různé části procesoru a tím i dosažení možnosti zpracovávat více instrukcí najednou.
    - Instruction Fetch (IF) - vyzvednutí instrukce
    - Instruction Decode (ID) - dekódování instrukce, zároveň se načítají registry
    - Execute (EX) - provedení instrukce
    - Memory Access (MA) - čtení z paměti
    - Write Back (WB) - zápis výsledku do paměti

					Current Instruction				
Cycle 1	IF	ID	EX	MA	WB				
Cycle 2		IF	ID	EX	MA	WB			
Cycle 3			IF	ID	EX	MA	WB		
Cycle 4				IF	ID	EX	MA	WB	
Cycle 5					IF	ID	EX	MA	WB

Branch prediction – předikce skoků

- Procesor obsahuje tzv. Branch prediction, kdy spekulativně vykonává kód dopředu podle toho, jaký předpokládá výsledek skoku. Uvedená konverze nemění stav branch-prediction, tj. neovlivňuje urychlování volající funkce, a kdyby se alokovala paměť, např. pro nový string, tak už to stav branch-prediction ovlivní.
- Jedním z problémů pipeliningu je fakt, že instrukce následující po skoku se vyzvedává dřív než je skok dokončen. Primitivní implementace vyzvedává vždy následující instrukci, což vede k tomu že se vždy mýlí, pokud je skok nepodmíněný. Pozdější implementace mají jednotku předpovídání skoku, která vždy správně předpoví (tedy předem dekóduje) nepodmíněný skok a s použitím cache se záznamem předchozího chování programu se pokusí předpovědět i cíl podmíněných skoků nebo skoků s adresou v registru nebo paměti.
- Skoky se predikují k udržení pipeliny. V případě, že se predikce nepovede, bývá nutné vyprázdnit celou pipeline a začít vyzvedávat instrukce ze správné adresy, což znamená relativně velké zdržení.
- Souvisejícím problémem je přerušení. Softwarová přerušení a výjimky jsou z hlediska procesoru skoky a je možné je predikovat, i když složitěji než obyčejné skoky. Hardwarová přerušení mohou oproti tomu dorazit kdykoliv bez předchozího upozornění, na druhou stranu v mnoha případech je možné „podvádět“ tak, že se přerušení nezačne provádět okamžitě, ale zařadí se na konec fronty, jakoby přišlo o několik instrukcí později. Znamená to ale další komplikaci při selhání predikce skoku - musí se totiž zrušit instrukce mezi skokem a přerušením, ale ne samotné přerušení, a cílovou adresu skoku uložit na zásobník pro návrat z přerušení.

Out of order

- Procesor vykonává instrukce v jiném pořadí, než je dáno programem tak, aby byly vykonány v co nejkratším čase.

Vektorové instrukce procesoru

- Dobrý překladač může vytvořit takový strojový kód, který umí využít paralelizačních schopností procesoru.
  - Některé (např. GCC v4) umí tzv. auto-vektorizaci, kdy je část kódu rovnou převedena na vektorové instrukce.
  - Proto se liší výkonnost kódu v závislosti na překladači a zvolené míře optimalizace.
- Ani ten nejlepší překladač neumí odhadnout vše, proto je třeba mu umět pomoci takovým zápisem kódu, ze kterého toho pozná co nejvíce.
  - Např. pomocí Loop-Unrolling.
  - Proto je třeba znát, jak to vypadá z pohledu procesoru.
  - Low-level techniky, pokud pro to není zvláštní důvod, se vyplatí nechat na překladači.
  - Ale algoritmy na vyšší úrovni jsou záležitosti pro programátora.

```
procedure VectorAdd (dst, src:PVector); assembler;
asm
…
   mov ecx, VectorSize    //128 bits = 16 bytes => 4
@loop:

   fld  [eax]  //dst -> ST0
   fadd [edx]  //src
   fst  [eax]  //ST0 -> dst
   add   eax, 4
   add   edx, 4

   dec   ecx
   cmp   ecx, 0
   jne  @loop
…
end;
```

```
type
  PVector = ^TVector;
  TVector = packed record
    x, y, z, w: single;
      //sizeof(single) = 4
 end;
```

```
SSE
movups  xmm0, [eax] //dst
movups  xmm1, [edx] //src
addps   xmm0,  xmm1
movups [eax],  xmm0 //dst
```

Loop Unrolling

- Nápověda pro překladač s autovektorizací.
- 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);`

Vektorové paralelní počítače

- Vector Processor, Array Procesor
- Mohou provádět operace s vektory čísel na úrovni instrukcí strojového kódu
- Dříve doména superpočítačů jako Cray-1, dnes:
    - MMX (Intel) - Matrix Math eXtension
    - SSE1-5 (Intel, AMD) - Streaming SIMD Extensions
    - 3DNow! (AMD)
    - AVX (Intel) - Advanced Vector Extensions
    - AltiVec (IMB, Apple, Motorola)

Superskalární architektura

- U superskalárního procesoru dochází k paralelizaci na úrovni (proudu) strojových instrukcí, které za sebou následují (to je odlišnost od vícejádrových procesorů, které zpracovávají současně instrukce z několika samostatných procesů či vláken).

(CISC a RISC - jen pro zajímavost)

- CISC instrukce se překládají na mikrokód, aby i procesory s CISCovou instrukční sadou mohly těžit z výhod RISCové architektury.
- Široká instrukční sada procesorů CISC usnadňuje jejich programování, protože není některé operace nutné rozepisovat (například násobení), avšak ve strojovém kódu (nebo v jazyce symbolických adres) se dnes programuje jen minimálně.
- Složitost CISC procesorů vede k problémům při výrobě (velká spotřeba materiálu, větší pravděpodobnost vady, komplikovaný návrh, problémy s vysokými frekvencemi, pipelining, cache atd). Typickými zástupci koncepce CISC jsou procesory rodiny Motorola 68000 a procesory postavené na architektuře Intel x86.
- RISC označuje procesory s redukovanou instrukční sadou, jejichž návrh je zaměřen na jednoduchou, vysoce optimalizovanou sadu strojových instrukcí, která je v protikladu s množstvím specializovaných instrukcí ostatních architektur.
- Mezi zástupce RISC procesorů patří ARM, MIPS, AMD Am29000, ARC, Atmel AVR, PA-RISC, Power (včetně PowerPC), SuperH, SPARC, DEC Alpha.

3. Paralelizace výpočtu součtů prefixů - charakteristika a řešení.

Typy proměnných

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

- **Lokální** proměnné (**a**)
- **Sdílené** proměnné
  - **Nezávislé** - využívané pouze pro čtení (**Items**)
  - **Závislé**
    - **Redukční** - v jedné iteraci čteny i zapsány (**sum**)
      (redukční - může být výsledkem redukčního stromu)
    - **Uzamykatelné -** Mohou (ale nemusí) být čteny i zapisovány v každé iteraci (i několikrát po sobě). Správný výsledek dostaneme i po náhodné posloupnosti iterací (**min**)
      (např. úloha hledání minima)
    - **Uspořádané** - správného výsledku je dosaženo pouze tehdy, pokud jsou iterace vykonávány ve stanoveném pořadí. Pole nelze rozdělit podle počtu vláken a nechat každé zpracovat svou část.
      Array[i]:=Array[i] + Array[i-1]

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

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

Výpočet součtu prefixů

Princip se dá použít na: Třídění, Lexikální analýzy, histogramy, teorie grafů, práci s řetězci.

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

$[a_0, a_1, \dots, a_{n-1}]$

$[(a_0), (a_0+a_1), \dots, (a_0+a_1+\dots+a_{n-1})]$

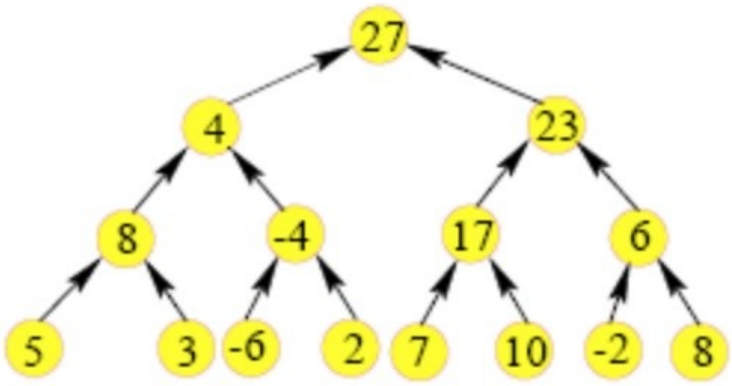
[4, 5, 3, 1, 6]

[4, 9, 12, 13, 19]

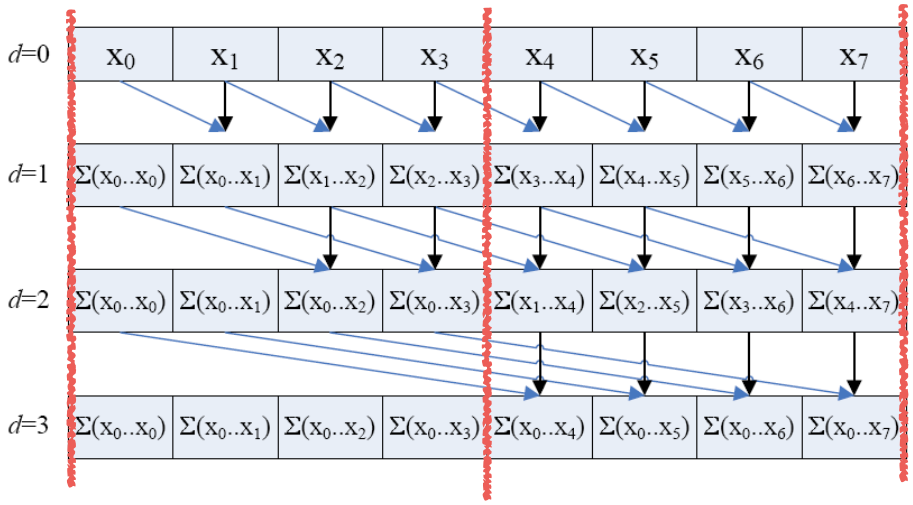
1. Uspořádaná proměnná zabraňuje zapisovat do pole v jiném než určeném pořadí. Zavede se proto kopie pole, s i-1 se se do něj přistupuje a nezapisuje.

```
Move(Temp, Items, Length(Items));

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

2. Bez uspořádanosti lze součet n-tého prvku rozložit do jiné posloupnosti součtů než v sekvenční verzi. Uvedený strom ukazuje, že lze paralelizovat součet posledního prvku.



3. Lze-li paralelizovat poslední prvek, je to možné i u ostatních. Mezisoučty vznikají v krocích d. Významná část mezisoučtů je využita jako mezihodnota dalších mezisoučtů.



Vlákna to mají takhle vertikálně rozdělený - (příklad pro 2 vlákna)

Vždycky zkopírujou svou část Itemů do Tempu a na pozici [i] udělají součet Temp[i-step]+Items[i].

Začíná se vždycky až na pozici [step], step se v každém kroku násobí dvěma.

4. Programový kód pro jedno vlákno, hranice dat pro vlákno je určeno pomocí Size a Offset.

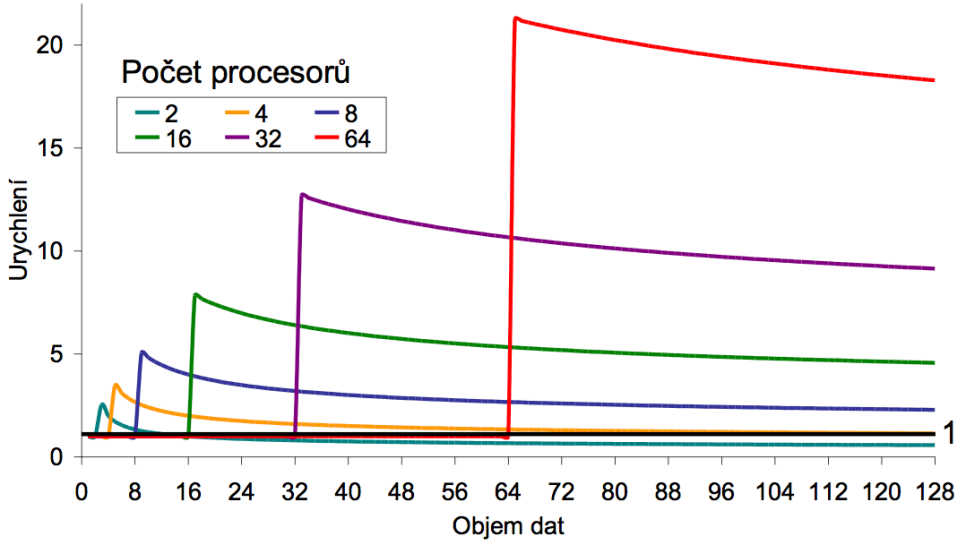
```
step:=1;
while step < High(Items) do
  begin
    Move(Temp^, @Items[Offset], Size);

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

    Barrier;                                     //konstrukce pro
    step:=step shl 1; //*2                       //závislou prom.
  end;
```

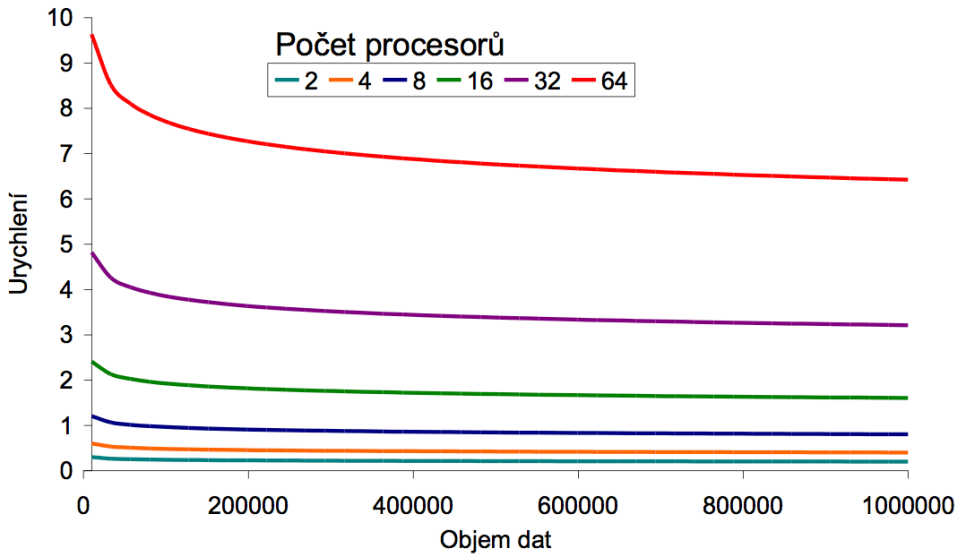
Shrnutí

- Původní uspořádaná proměnná nahrazena jinou uspořádanou proměnnou **`step`** (d).
- Hlavní cyklus řízený uspořádanou proměnnou má méně iterací, v každé se provádí více práce, kterou lze již paralelizovat.

Urychlení

- Sekvenčně - $O(n)$
- Paralelně jedním vláknem - $O(0{,}5 \cdot n \cdot \log_2 n)$
- Při vhodné paralelizaci se zanedbáním režie plánování, kopírování - $O((0{,}5 \cdot n \cdot \log_2 n)/m)$
- Algoritmu je závislý na **počtu** procesorů a **objemu** dat, urychlení jen za určitých podmínek:
- U řádově statisíců dat:
  - cca stejné nebo pomalejší při 8 procesorech, mnohem pomalejší při méně
  - cca 2x urychlení při 16 procesorech
  - cca 4x urychlení při 32 procesorech
  - cca 8x urychlení při 64 procesorech

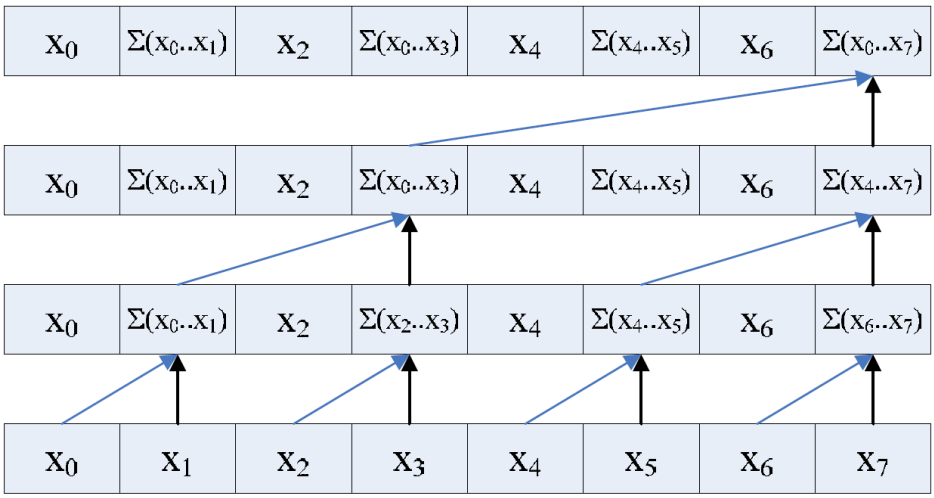




Optimalizace

Lze provést optimalizace algoritmu se kterou se dostaneme na složitost $O(n)$ - stejně jako u sekvenční, tedy při spuštění paralelně $O(n/m)$.

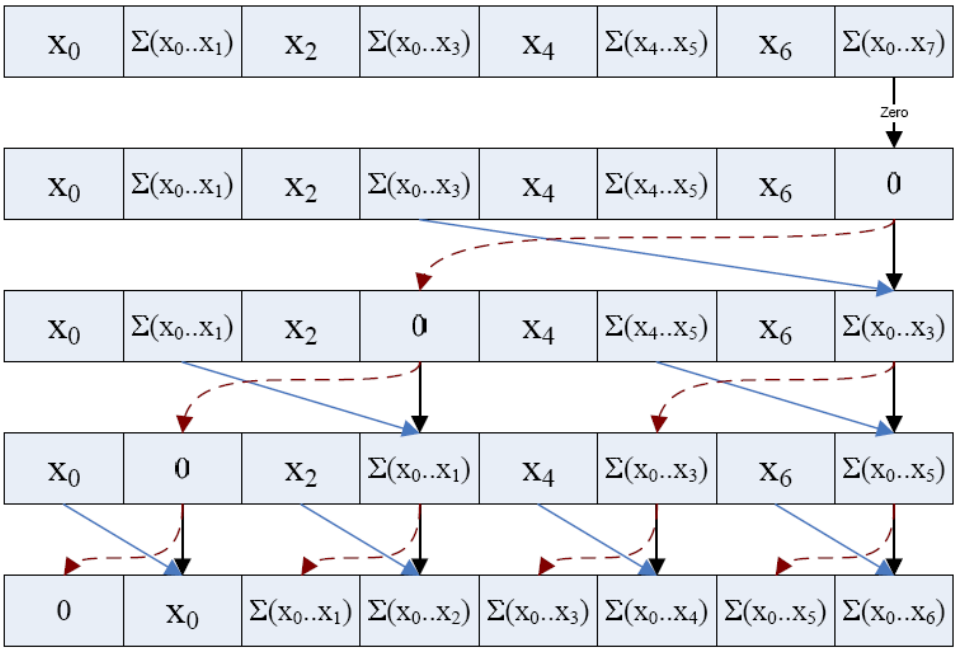
1. Redukční fáze od zdola nahoru




```
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. Záloha součtu posledního prvku

3. Fáze smetení (down sweep), od shora dolu.



```
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];
    tems[i+d]:=Items[i+d]+temp;
  end; Barier; d:=d shr 1;
end;
```

4. Posunutí všech součtů o jeden doleva, 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;
```

Shrnutí

Součet paralelních prefixů alias “scan”:

- Dvě varianty
    - Inclusive - první verze
    - Exclusive - nula z optimalizace na začátku, neobsahuje finální součet
- Podporováno např. v MPI (později) funkcí MPI_Scan
- Nemusí se jenom sčítat
- I některé súlohy s defaultně uspořádanými proměnnými jdou řešit

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

Amdahlův zákon

Amdahlův zákon byl formulován v roce 1967 norsko-americkým počítačovým architektem Genem Myronem Amdahlem. Vyjadřuje maximální předpokládané zrychlení systému po vylepšení některé z jeho částí. Doba běhu sekvence je rozdělena čas strávený výpočtem sériové části programu t_s a části, kterou je možno paralelizovat t_p.

$$T = t_s + t_p$$

Paralelizací jde urychlit samozřejmě jen část tp, tudíž při použití p procesorů dosáhneme teoretického času.

$$T_p = t_s + \frac{t_p}{p}$$

Zrychlení (speedup) tedy bude:

$$S = \frac{T}{T_p}$$

Symbolem *f* se značí poměr času stráveného výpočtem sérové části kódu ku celkovému času:

$$f = \frac{t_s}{t_s + t_p}$$

Amdahlův zákon ukazuje, že s rostoucím počtem procesorů se zvyšuje zrychlení, ale maximálně může dosáhnout hodnoty *1/f*. Předpokládá se zde úkol o konstantní velikosti, který se nemění s navyšováním počtu procesorů.

$$S = \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}$$

Gustafsonův zákon

Tento zákon je pojmenován po jednom z jeho autorů Johnu L. Gustafsonovi a vznikl na základě nedostatku Amdahlova zákona. Gustafson tvrdil, že s vyšším počtem procesorů je možné stejný problém vyřešit ve stejném čase přesněji, resp. mění pohled na paralelizaci výpočtu tak, aby během stejného času bylo možné vyřešit složitější problémy.

Zpracování problému na paralelním počítači můžeme rozdělit na:

$$a + b$$

kde *a* je sekvenční část programu a *b* paralelní. Na p procesorech je potom čas výpočtu:

$$a + p \cdot b$$

kde *p* je počet procesorů (zvyšuje se) a *b* je paralelní část výpočtu na jednom procesoru (fixní).

Dostaneme tedy urychlení: $S = \frac{a + p \cdot b}{a + b}$ Pokud definujeme alfa jako $\alpha = \frac{a}{a+b}$,

můžeme odvodit Gustafsonův zákon: $S = p - \alpha \cdot (p - 1)$

Porovnání Amdahlův vs Gustafsonův

Gustafsonův zákon v žádném případě Amdahlův zákon nepopírá, říká něco jiného. V Amdahlově verzi se zabýváme možným zrychlením problému o konstantní velikosti a s konstantní sekvenční částí - v tomto případě je možné zrychlení skutečně omezeno. Gustafson si za konstantu volí dobu běhu na paralelním systému a říká, že zrychlení není omezeno, ovšem za předpokladu dostatečně velkého problému.

Zformulujeme oba zákony bez počítačové terminologie. Představme si auto jedoucí z místa A do místa B, přičemž tato místa jsou vzdálena 60 km. Za první hodinu jízdy auto urazilo 30 km.

Amdahlův zákon říká, že i když auto pojede zbytek cesty libovolnou rychlostí, nemůže jeho průměrná rychlost překročit 60 km/h (i kdyby jel po první hodině nekonečnou rychlostí, urazil by 60 kilometrů za hodinu).

Gustafsonův zákon oproti tomu říká, že pokud auto pojede dostatečně daleko, může dosáhnout libovolné průměrné rychlosti. Aby například dosáhlo v průměru 90 km/h, stačí, aby jelo další hodinu rychlostí 150 km/h nebo další dvě hodiny rychlostí 120 km/h apod.

Karp-Flattova metrika

Karp-Flattova metrika se dá použít ke zjištění příčiny neefektivity paralelního algoritmu. Umožňuje určit, jestli je způsobena nadměrnou režií nebo velkou sériovou částí. Spočítá se z naměřených hodnot podle vzorce:

$$e = \frac{\frac{1}{\psi} - \frac{1}{p}}{1 - \frac{1}{p}}$$

kde *p* je počet procesorů a *psi* je urychlení na *p* procesorech. Z průběhu hodnoty *e* při zvyšujícím se počtu procesorů *p* lze odvodit, že je-li *e* konstantní, příčinou neefektivity je velká sériová část, při rostoucím *e* je příčinou nadměrná paralelní režie.

5. Programové prostředky pro multithreading - POSIX, WinAPI, C++11.

C a vlákna POSIX

- V standardním Ansi C nejsou primitiva pro vytváření vláken, Vše potřebné v knihovně POSIX
- Cca 100 procedur “pthread_”
- 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).
- Tři základní struktury, reprezentovány pomocí tzv. **handle**, což je zobecněný ukazatel.
  - **Vlákna (pthread_t)**
    - Vlastní program vlákna je uložen v metodě `void* prog_name(void* arg){}`
    - Vlákno se rozběhne **hned po vytvoření.**
    - Stavy: ready, running, waiting, terminated.
    - Vlákno se ruší pthread_exit (ze stejného vlákna) nebo pthread_cancel (z cizího).

```
pthread_t thread_handle;
int x;
int status = pthread_create(&thread_handle, NULL, prog_name, &x);
```

  - **Mutexy (pthreat_mutex_t)**
    - zámek, synchronizační primitivum používané pro vzájemné vyloučení v kritické sekci
    - normální - vlákno ho může zamknout jen jednou
    - rekurzivní - vlákno ho může zamknout víckrát (rekurzivní zpracování globálních dat)
    - ladící - pozná se opakované zamčení

```
pthread_mutex_t mutex_handle = PTHREAD_MUTEX_INITIALIZER;
pthread_mustex_init(mutex_handle); // dynamicky místo 👆

pthread_mutex_lock(mutex_handle);
pthread_mutex_unlock(mutex_handle)
state = pthread_try_lock(mutex_handle)
```

  - **Podmínkové proměnné (pthread_cond_t)**
    - Používají se pro synchronizaci vláken. Vlákno se uspí do té doby, dokud se nesplní nějaká podmínka, která se pak signalizuje pomocí podmínkové proměnné.
    - Podmínkové proměnné implementují frontu, kde vlákna čekají 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.

```
pthread_cond_init
pthread_cond_wait // blokující operace, vlákno se převede
                     do stavu waiting, a odemkne se zámek
pthread_cond_timedwait
pthread_cond_signal
pthread_cond_broadcast - jako notifyAll()
pthread_cond_destroy
```

WinAPI

Process

- Má virtuální paměťový prostor, systémové zdroje, bezpečnostní kontext (kdokoliv nemůže provádět cokoliv), jedinečný identifikátor, spustitelný kód.
- Prioritní třída - vlákna pak mohou mít prioritu pouze v rozsahu prioritní třídy procesu.
- Má alespoň jedno vlákno (primární - to hlavní), které může vytvářet další - threads a fibers.

Thread

- Entita v rámci procesu, kterou plánuje OS. Všechny vlákna sdílí paměťový prostor a zdroje svého procesu. Vlastní handler výjimek. Priorita: OS zajišťuje inverze priorit, dynamic boost, foreground/input včetně realtime.
- Jedinečný identifikátor
- TLS – thread local storage, data, která jsou specifická/lokální pro daný thread. 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 výjimek.
- Hodně různých stavů vlákna, obecně je lze rozdělit do čtyřstavového modelu (ready, running, waiting, terminated).

Fiber

- Běží v kontextu vlákna, plánuje ho thread procesu, jeden thread může naplánovat několik fibers.
- FLS – fiber local storage = analogie k TLS, FLS je asociováno s threadem.
- Má malý kontext (v porovnání s kontextem threadu) – zásobník, podmnožinu registrů a inicializační data.
- Má přístup do TLS threadu, v jehož kontextu běží.
- Nemá prioritu.

Synchronizační objekty

Oproti všemožným datovým strukturám POSIXu má WinAPI pouze jedinou - **handle**. Díky tomu je pro synchronizaci jakoukoli konstrukcí volána jedna funkce **WaitForSingleObject** (případně **WaitForMultiplebjects**, to lze použít třeba i místo sady pthread_join).

- **Event**
  - Stavy signaled/nonsignaled
  - Může být buď pulsní, tj, překlopí se do nonsignaled jakmile propustí jednoho čekajícího, nebo zůstane signaled (Manual reset Event, Auto Reset Event).
  - CreateEvent, SetEvent, ResetEvent.
- **Mutex** - vyžaduje přepnutí do režimu jádra. CreateMutex, OpenMutex, ReleaseMutex.
- **Semafor** - klasický semafor, CreateSemaphore, OpenSemaphore, ReleaseSemaphore
- **Kritická sekce** - má velkou režii, EnterCriticalSection (blokující), TryEnterCriticalSection (neblokující), LeaveCriticalSection.

Další

- **IO operace** jsou buď blokující nebo asynchronní
  - Konstrukce pro kontrolu dokončení (včetně cancelIO)
- **APC, asynchronous procedure call**
  - Rutina, která se provede v kontextu daného threadu
  - Každý thread má svou frontu APC, jednotlivé APC jsou plánovány místo normálního běhu vlákna
  - Když je APC zařazeno do fronty, je zavoláno SW přerušení. Při příštím naplánování threadu je provedena APC funkce.
- **User mode scheduling**
  - Lightweight mechanismus (jako fibers)
  - Na rozdíl od fiberů má UMS objekt svůj kontext
  - UMS thread se vytvoří konverzí z normálního threadu

C++0x aka C++11

- Základem je třída std::thread

```
#include <thread>

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

- Prvním argumentem třídy je funkce (příp. její adresa), dalšími pak její argumenty. Argumenty se kopírují do interního úložiště threadu.

- **Synchronizační primitiva:**
  - std::**mutex**
    - vlákno vlastní mutex od chvíle, kdy úspěšně zavolá **lock()** (čeká, pokud je zamčený), nebo **try_lock()** (vrací ihned true/false). Ostatní vlákna jsou blokována, nebo dostávají false.
    - odemčení mutexu pomocí **unlock()**, může provést pouze vlákno aktuálně vlastnící mutex, jinak je výsledek nedefinovaný
  - std::**recursive_mutex** - může být jedním vláknem zamčený víckrát, stejněkrát musí být odemčen
  - std::**timed_mutex**
  - std::**recursive_timed_mutex** - navíc metody **try_lock_for()** a **try_lock_until()**

- **Synchronizační šablony**
  - šablona std::**lock_guard** - zajistí, že daný blok kódu bude hlídán mutexem a že že kritická sekce bude opuštěna vždy, když vykonávaný kód opustí daný blok kódu. Řeší i zpracování výjimek, na konec try do catch se nemusí dávat unlock.
  - šablona std::**unique_lock** - umožňuje vytvořit zámek, který se zamkne až na vyžádání. std::lock(lock1, lock2) - uzamknutí unique_locků s bez **dealokovat**

- **Podmínkové proměnné** std::**condition_variable**. Její metody:
  - **notify_one** - odblokuje jedno z čekajících vláken
  - **notify_all** - odblokuje všechna čekající vlákna
  - **wait** - zablokování aktuálního vlákna dokud není probuzeno. Jako parametr bere zámek, který **musí** být zamčený v době volání wait.
  - **wait_for** - zablokování aktuálního vlákna dokud není probuzeno nebo než uplyne doba
  - **wait_until** - zablokování aktuálního vlákna dokud není probuzeno nebo do určité chvíle

- **Atomické operace a inicializace**
  - std::**call_once** - zavolá funkci jen jednou, i kdyby byla volána z více vláken
  - std::**atomic** - šablony definující atomické typy
    typedefy, např. std::atomic_char je std::atomic<char> atd.

- **Asynchronní volání**
  - std::**async** - vykoná kód funkce nezávisle na vláknu, ze kterého byla std::async zavolána, nezaručuje ale, že se vykoná v jiném vláknu
  - std::**future** - vrací hodnotu std::**async** výpočtu
  - std::**promise** - např. pro I/O operace. Je svázán s future, jeden thread pomocí set_value zapíše do promise výsledek, druhý thread si ho z future vyzvedne. Pormise je v podstatě úložiště hodnoty, dokud si ji někdo nevyzvedne přes svázanou future.

(Java - není již v otázce)

(Ada - není již v otázce)

6. Intel Threading Building Blocks.

- Open Source, podpora více platforem
- TBB knihovna jako taková využívá vláken OS, ale programátor s nimi již nepracuje. Místo toho jsou v TBB tzv. **Tasky** – knihovna pak vykonává jednotlivé úlohy paralelně.
- Programátor specifikuje tasky a jejich návaznosti
- **Redukce cache-coolingu**
  - tradičně M vláken, N procesorl, kde M > N
  - když běží vlákno, data se dostávají do cache procesoru
  - přepnutím vlákna se musí přepsat cache - vznikají cache miss když vlákno data nenajde
  - TBB poskytuje vlákna tak, aby M < N, pracuje na úloze, dokud není dokončena => redukuje cache-cooling
  - Tradiční vlákna nic takového neumí
- **Task stealing**
  - způsob plánování úloh
  - každá procesor má zásobník úloh
  - dokončí-li svůj zásobník, může se pokusit ukrást cool data z vrcholu zásobníku jiného procesoru
- TBB vykonává úlohy paralelně jak nejlíp to se současným know-how jde
  - Lze si napsat vlastní plánovač tasků (ale to je opravdu pokročilá záležitost, člověk to nenapíše lépe než lidi, kteří se tím zabývají roky).
- Protože TBB nespoléhá jen na direktivy překladače, lze paralelizovat i takové úlohy, které by se s OpenMP paralelizovaly těžko.
- **Základní konstrukce**
  - parallel_for, parallel_reduce, parallel_scan
  - parallel_while, parallel_do, pipeline, parallel_sort
  - mutex, spin_mutex, queuing_mutex…
  - fetch_and_add, fetch_and_increment…
- **Případová studie použití**
  - Výpočetně náročná aplikace na rozsáhlými daty s GUI
  - Jedno vlákno obsluhuje GUI
  - Jedno vlákno obsluhuje I/O
  - Výpočet
    - přístup s vlastními vlákny by znamenal:
    - vytváření a rušení vláken (další režie pro OS)
    - jejich správná synchronizace
    - optimální je stejný počet vláken jako procesorů, ale to vždy není možné programově zařídit
  - S TBB
    - programátor napíše jedno vlákno, ve kterém spustí výpočet úloh TBB
    - optimálně by každá úloha měla po dokončení vracet další úlohu, která se má vykonat jako navazující

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

- Ada je objektově orientovaný jazyk se silnou verifikací typů (nelze implicitně přetypovat datové typy – např. void pointer).
- Nepoužívá interpretr (tedy je kompilovaný).
- Nepodporuje fibres.
- 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 Rendez-Vous (zasílání zpráv).

Tasky

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

```
task jméno is
deklarace jmen komunikačních typů
end jméno;;
task body jméno is
lokální deklarace a příkazy
end jméno;
```

- Pro ukončení procesu 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 rendezvous, kterým eliminuje potřebu semaforů (umí je nahradit), umožňuje synchronní a nepřímo (zavedením pomocného procesu) i asynchronní komunikaci procesů zasíláním zpráv. Dovoluje tak elegantní konstrukci monitorů.

- 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.
- Z pohledu klienta je rendezvouz procedurální volání (lokální nebo vzdálené).
- Klient se blokuje do té doby, než server vykoná accept statement.
- Poté co se ze serveru vrátí výsledek, oba pokračují nezávisle na sobě dál.

```
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
- průběh dostaveníčka - accept:
    - klient zavolá server
    - server si převezme parametry
    - server provede výpočet, klient spí o 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.

```
//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 podmínka =>     //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.

Porovnání s monitorem

- 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.

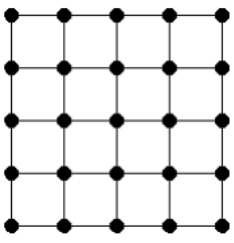
8. Výpočetní prostředí s distribuovanou pamětí.

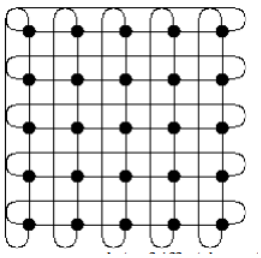
- Systém pro paralelní výpočet s distribuovanou pamětí se skládá z výpočetních uzlů a komunikačních kanálů.
  - Univerzální počítačová síť (softwarový multipočítač, SETI)
  - Univerzální paralelní počítač (stavěný přímo pro paralelní počítač, Cluster)
  - Jednoúčelový paralelní počítač (jedna konkrétní aplikace s maximální optimalizací)
- 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 ovšem závisí na dalších parametrech:
    - Objemu interakce
    - Celkovém objemu zpracovávaných dat
    - Konkrétní HW architektura
    - Jak dalece je použitý programový kód optimální pro danou architekturu

HW pohled

je dobrý mít

- Topologie obecně
  - **Pravidelná**: krychle, mřížka, hvězda
  - **Nepravidelná**: Internet
- Fyzická topologie
  - **Pevná**: procesory jsou propojeny komunikačním kanálem
    - Adresa může reflektovat polohu v síti
    - Každý s každým
    - 2d mřížka $d_{max} = 2(n - 1)$
    - Toroid $d_{max} = n-1$
    - 3d mřížka, n-rozměrné krychle…
  - **Flexibilní**: přepínání okruhů, přepínání paketů





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)
- **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, aby z jedné sítě staly dvě

Snahou je dosáhnout

- Co největšího počtu uzlů v síti
    - Škálovatelnost
- Co nejmenší komunikační vzdálenosti – dmax
    - Tj. omezit komunikační zpoždění
- Co nejmenší počet sousedů
    - Aneb, i komunikační kanál něco stojí
- Co největší přenosové rychlosti

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 výpočetem
        - 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/lockCstepped algoritmy – na společném uzlu by dva spolupracující procesy na sebe musely čekat dobu výpočtu jednoho kroku

Identifikace procesů

- Jedinečná ID procesů
- 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)

Komunikační schéma

- Fyzická topologie
- Síťová topologie
- Virtuální topologie (komunikační vazby procesů)
- Ideálně 1:1 (aby docházelo k nejmenším zpožděním)

9. Rozdíly mezi PVM a MPI.

Thursday, May 30, 2013 8:34 AM

- 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
  - flag – možnosti spuštění
    - 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
  - 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 patřídí 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 oprace.

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.

  From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

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();
}
```

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

Thursday, May 30, 2013 8:34 AM

Přidělování práce v prostředí s distribuovanou pamětí

Pozn. Přednáška z PPR b_advance.

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

multiple processes, multiple data

- 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

Hodí se tehdy, když:
- Datový objem zpráv je malý
- Výpočet je v porovnání s objemem zpráv náročný
- Např. nemá smysl počítat pole integerů k součtu na jiný uzel, v dnešní době je **operace mov zhruba stejně časově náročná jako add, muselo by se tedy vykonat spoustu operací navíc** => zpomalení

V heterogenním prostředí se realizuje automatický **load balancing** viz dále, urychlení pak závisí na velikosti 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 k urychlení už vedou, ale zase se zbytečně příliš čeká na procesy, které z různých důvodů pracují pomaleji než ostatní

Ideální velikost posílaných dat tak závisí na:
- Výpočetním výkonu uzlů
- Měla by uzel zaměstnat na 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 využívána všemi procesy
      - □ 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 (pipeline)
- 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

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 se označují termínem 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

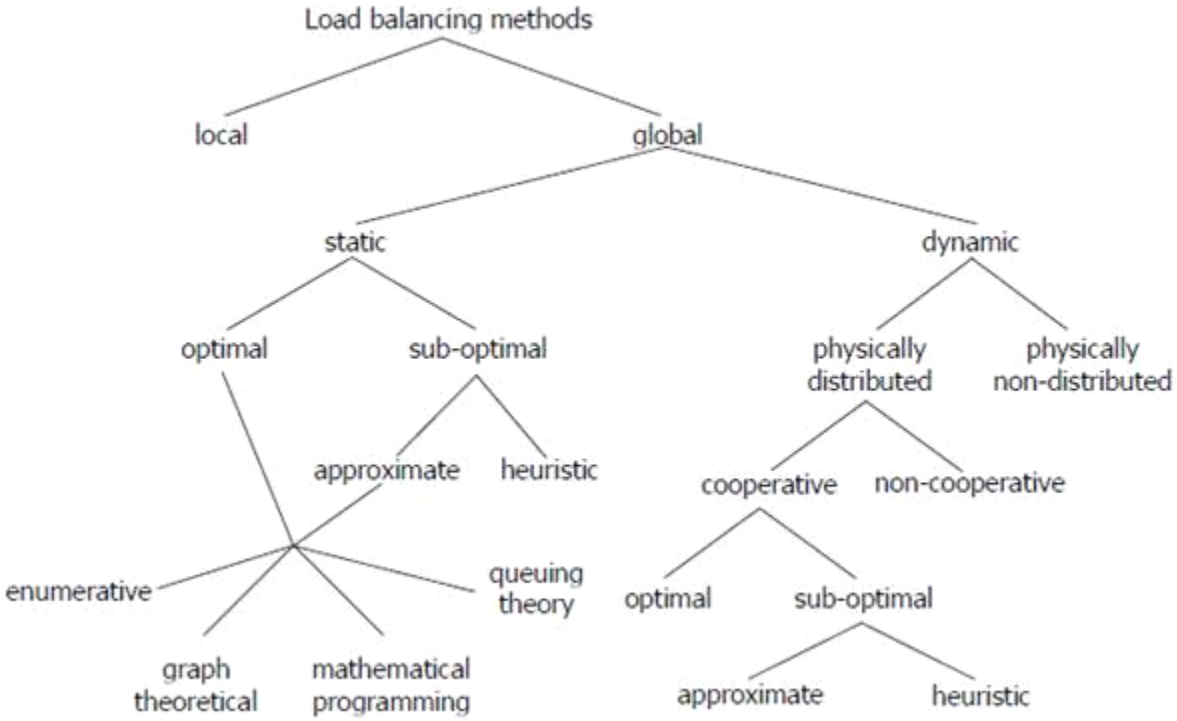
- komponenty červa
  - inicializační kód ke spuštění master
  - inicializační kód ke spuštění slave
  - výpočetní program
  - ze života červa
    - nalezení nevytíženého uzlu
  - žádný uzel nezná globální stav sítě, a proto se každý segment musí postarat o nalezení volné stanice sám za sebe
  - lze hledat například pomocí broadcast 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
- uzly musejí být identické
- co s otevřenými soubory?
  - 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 – pokud ji metoda umožňuje dosáhnout
  - 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

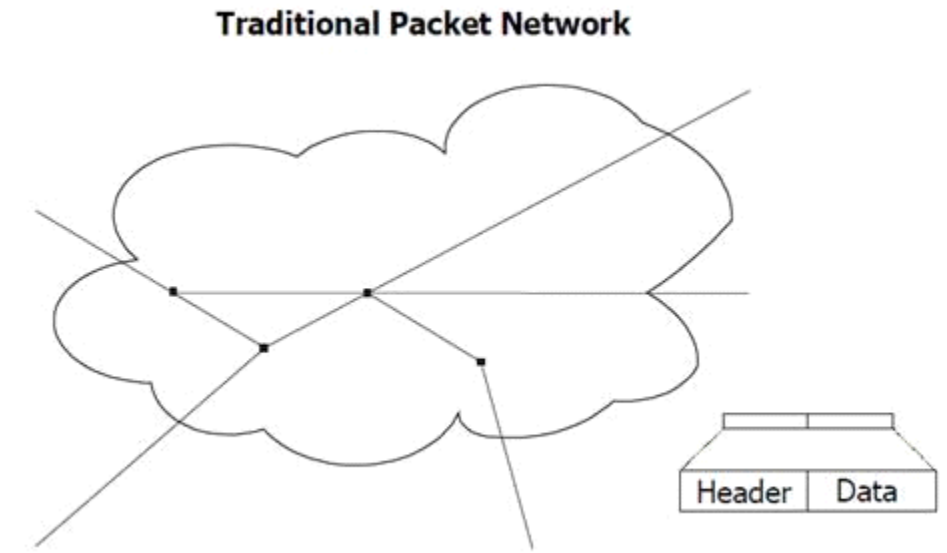
- adaptivní
  - síť prochází změnami během výpočtu – mění se stav uzlů
  - adaptivní metody berou do úvahy i několik předchozích stavů při rozhodování o přidělení zátěže
- kooperativní
  - každý proces se může rozhodovat buď sám za sebe, nebo může na rozhodnutí spolupracovat s ostatními
  - přímo – procesy 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

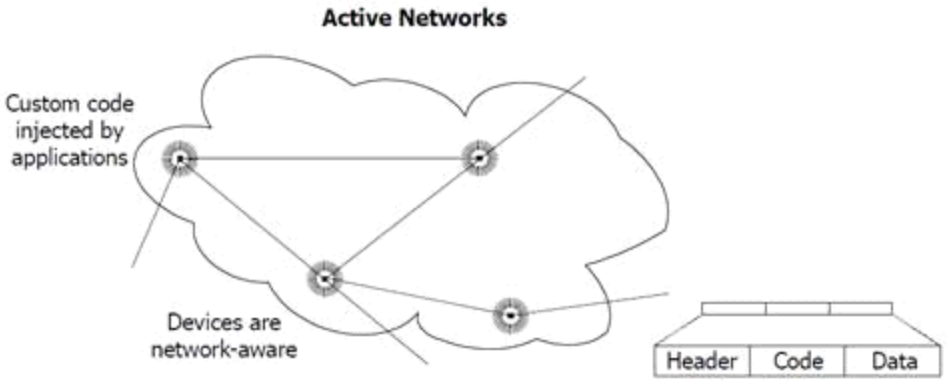
Load Redistribution

- load-balancing tradičně měří zátěž v počtu procesů, což není zrovna to nejlepší
- následující text bude o Load-Redistribution Method in Distributed Environment
- metoda vyžaduje pokročilou síťovou architekturu jako jsou Aktivní sítě

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





- 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
- Při přerozdělování zátěže (load redistribution) se procesy rozhodují samy za sebe, periodicky sledují své okolí (jako kolonie organismů)
  - Dosažení vyváženého stavu ne hned, ale postupně
- Capsule zjistí síťové okolí uzlu z hlediska topologie, výkonnosti, zatížení, komunikačního zpoždění apod.
  - Využije se při hledání výhodnějšího uzlu pro odmigrování
- **Rizika:**
  - *Masová migrace* - více procesů si vybere stejný uzel, ten se stane přetíženým

- *Oscilace* - proces se může pohybovat po síti, aniž by něco počítal
    - řešení - zavedení kreditů, od určitého počtu může migrovat
- *Zbytečná migrace*

Přiřazení procesů na jednotlivé uzly

- **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 výpočetem
            - 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/lockCstepped algoritmy – na společném uzlu by dva spolupracující procesy na sebe musely čekat dobu výpočtu jednoho kroku
- **Identifikace procesů**
    - Jedinečná ID procesů
    - 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)

11. Systémy reálného času.

- Dokončení výpočtu ve stanoveném termínu je kritické
- Termín musí být dodržen bez ohledu na zátěž systému
- př. Brzdy v autě, vojenský systém, podpora života, jaderné zařízení, řízení výroby, mobilní telefony

- **Hard Real Time**
  - Dokončení výpočtu po termínu se považuje za chybu a výsledek bezcenný - *sctrict deadline*
  - Nedodržení termínu může vést k celkovému selhání systému (**airbag, řízení motoru**)
  - Vyžadovány tam, kde hrozí příliš velké škody v případě selhání systému
- **Soft Real Time**
  - překročení termínu se toleruje, systém reaguje se zhoršenou kvalitou poskytovaných služeb (**vypadne pár snímků** videopřenosu, přílet letadla se dozvíme s **několikavteřinovým zpožděním)**
- Výkonnost
  - Nejde o to vypočítat co nejvíc, ale vypočítat to včas
  - 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
- **Výběr algoritmu**
  - Je potřeba znát nejhorší možný čas výpočtu algoritmu (kdy vrátí výsledek)
  - Př. QuickSort - nejlepší $O(N \cdot \log N)$ - nejhorší $O(N^2)$
    HeapSort v praxi pomalejší, ale zaručeno $O(N \cdot \log N)$, proto chodnější
  - Dále důležitá spotřeba paměti, možnost paralelizace

RMA (Rate Monotonic Analysis)

- Plánovací algoritmus systému reálného času
- Přiřazuje priority jednotlivým úlohám tak, aby stihly dokončit výpočet v termínu - možným výsledkem je, že zjistí, že to úloha nestihne
- **Periodická úloha**
  - Je opakovaně/periodicky runnable v pevně daných intervalech
  - Perioda - časové rozmezí, kdy je úloha runnable
  - Deadline - začátek další periody
- **Priorita**
  - Čím kratší perioda, tím častěji běží => větší priorita úlohy
  - Ratio grid rozdělí intrval, aby byl poměr mezi sousedy stejný

- **Plánování**
  - Někdy jsou kritické úkoly zajišťovány úlohami s dlouhou periodou - řešením je rozdělit ji do několik menších s kratší periodou
- **Naplánovatelný systém**
  - C_i/T_i je využití procesoru i-tou úlohou z *n* úloh
  - Lui a Layland dokázali, že

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

U - využití procesoru

n - počet úloh

C_i - výpočetní čas úlohy

T_i - perioda úlohy

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

- Udržíme-li zatížení procesoru n úlohami pod 70%, pak je možné naplánovat všechny úlohy tak, aby dodržely své deadlines
- Neznamená to, že nelze nalét plánování se stejnými vlastnostmi při větším zatížení
- Zbývajích 30% můžou bát non-real time thready

  - Zatížení **70%** můžeme použít jako **rychlý test**, zda je systém **naplánovatelný**

- **Response Time Test**
  - Lze použít, pokud nevychází test se 70%
  - Pro každou úlohu rekurzivně vypočítáme $\mathbf{R_k}$ (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

$$R_{k+1} = C_i + \sum C_j \times ceil\left(\frac{R_k}{T_j}\right)$$

- Algoritmus 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, úlohu lze naplánovat

- **Aperiodické a sporadické úlohy**
  - Zpracování událostí - např. HW přerušení
  - Inicilizace
  - Zotavení se z chyb systému
  - Příkaz / požadavek operátora

  - **Soft deadline**
    - př. změn parametrů radaru
    - dá se tolerovat zpoždění v reakci na příkaz operátora
    - co nejmenší zpoždění je žádoucí, ale nikdy na úkor hard dedline úloh
  - **Hard deadline**
    - 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, plánovány podle pravidel RMA
    - Sporadické úlohy jsou nejprve seřazeny EDF (Earliest Deadline First)

- **Synchronizace - kritická sekce**
  - 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 tu s nízkou a tím **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 prioritu dokončí výpočet o něco později
      => Většinou se to stihne
      => Ale pokud ne, vede to k závažným problémům, může dojít k selhání systému

  - **Zákaz všech přerušení**
    - Tj. i hodin
    - Jedna úloha je absolutním pánem systému
    - Kritická sekce musí být krátká, jinak systém selže i tak
    - Používá se v malých systémech, nehodí se pro general-purpose

- **Dědičnost priorit**
  - 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 úlohami požadovány
  - Při vstupu do kritické sekce dostane úloha nejvyšší prioritu ze všech, které tam kdy budou chtít vstoupit
  - Nemůže dojít k **uvíznutí** - neexistuje tedy žádná jiná úloha, která by ji mohla v kritické sekci přerušit
  - Může dojít k **vyhladovění** - úloha s nízkou prioritou bude v kritické sekci moc dlouho, nějaká vyšší nestihne deadline

- **Synchronizace - zprávy**
  - Možnost synchronizace, kdy nedojde k uvíznutí díky zámku kritické sekce
  - Zpráva může obsahovat synchronizační informace
  - Většina OS zprávu kopíruje 2x - z paměti odesílajícího do fronty, z fronty do příjemce
  - To může být pro RT pomalé, lze řešit pouze převedením části pamětí odesílajícího pod příjemce, úlohy na to musí být přizpůsobené

- **Správa paměti**
  - Kritická záležitost i z pohledu času
  - Malloc/free fragmentují, trvá než se najde vhodný blok paměti
    - nemusí se stihnout deadline
    - RTS může běžet klidně několik let bez restartu
  - Defragmentace - nedá předpovědět, jak dlouho bude trvat - zvyšují riziko nestih, 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

- **R**eal **T**ime **O**perating **S**ystem
- 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
    - Minimální režie při přepínání úloh
- Plánování úloh
    - **Událostně řízené**
        - úloha se přepně jen když nastane událost s vyšší prioritou
        - kooperativní multithreading - úloha se po nějaké době dobrovolně vzdá procesoru
    - **Sdílení času**
        - tj .virtualizace procesoru
        - úlohy se přepínají nejen událostmi, ale i hodinami
        - Barrel Procesor
            - HW zaručuje, že v každém cyklu vykoná jednu instrukci z N běžících vláken jednou za N cyklů.
            - Nulová režie přepínání vláken
            - I kdyby nejaké vlákno uvízlo, nebo zpracovávalo nějaké přerušení, další vlákna běží dál ve stanovených časech
            - N je konečné, vysoké N jsou nákladná na design i na výrobu
- Earliest Deadline First (**EDF**)
    - Vlákna v prioritní frontě
    - Při události (jako vytvoření nebo dokončení vlákna) se prohledává fronta a vybírají se vlákna s **nejbližším termínem**, z nich pak 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 deadline
- Monotonic Scheduling
    - viz RMA

RTS bez RTOS

- Dostatečně malý projekt se může obejít bez velkého RTOS
- Př. pro ledničku, kde se jen hlídá teplota a žárovka není nutný RTOS, ale co složitější lednice?
- Rozhodování - cena, velikost projektu, možná časem vlastní kód vytvoří vlastní RTOS

Java Real-Time System

- Java neumí
  - Používat striktně prioritní plánování vláken - zámky neumí inverzi priorit, priority ceiling…
  - RT vhodnou správu paměti
- Proto vzinklo Real Time Specification for Java (**RTSJ**)
  - Specifikuje minimální požadavky na správu vláken, různé modely plánování je možné doinstalovat do JVM
  - Část paměti lze vyloučit z active garbage collectoru
  - Vybraná vlákna lze označit za nepřerušitelná garbage collectorem

X1. Bonus - řazení

- Hloupý **O(N^2)**
  - **Selection sort**
    - Nestabilní
    - Rozdělení na seřazená / neseřazená
    - V neseřazený hledá minimum a vymění ho s prvním prvkem
    - Rozšíří oblast seřazených o jedno
  - **Insert sort**
    - Nestabilní
    - Rozdělení na seřazená / neseřazená
    - Vybere z neseřazených minimum, zařadí ho na správnou pozici do seřazených
  - **Bubble sort**
    - Stabilní
    - Porovnává dva sousední furt dokola

- Chytrý **O(N log N)**
  - **Heapsort**
    - Využití haldy - postupné vkládání do haldy a postupný výběr největšího
  - **Mergesort**
    - Rozdělí neseřazený na přibližně stejný podmnožiny
    - Seřadí podmnožin
    - Spojí podmnožiny
  - **Quicksort**
    - V nejhorším O(N^2)
    - Rozdělení na větší než pivot / menší než pivot (většinou medián)
    - Řadí se obě části zvlášť, pak se skládají dohromady

X2. Bonus - monitor v Javě

- Modul, kde jsou sdružena data i procedury, které s nimi pracují.
- Procesy volají metody, ale nepřistupují přímo k datům.

- Instance třídy, která může být bezpečně použita několika vlákny.
- Všechny metody monitoru jsou vykonávány se vzájemným vyloučením (nanejvýš jedno vlákno může vykonávat metodu monitoru).
- Umožňují také nechat vlákno čekat na nějakou podmínku, během čekání se vlákno vzdá svého exkluzivního přístupu a znovu si ho vezme po splnění podmínky (dá se signalizovat jednomu nebo více vláknům).
    - Tím umožní vstup do monitoru jinému vláknu.
- Výhody
    - Abstrakce, programátor se při použití nemusí starat o to, jak je to uvnitř implementováno.
    - Monitor funguje pro libovolné množství vláken.
    - Nemusím se uvolňovat nic jako mutex.

- Nad podmínkou čekají vlákna, který chtějí do výhradní sekce.
- Nad podmínkou se volá wait() nebo signal(), podobné jako P() a V() s tím rozdílem, že u signal() se nic nestane, pokud je fronta prázdná.
- Vlákna jsou většinou obsluhována FIFO.

- Když je vlákno vzbuzeno nad podmínkou, musí zažádat o zámek.
- V Javě je použita podmínková proměnná SC (Signal & Continue) - vlákno, které signalizuje, stále drží zámek a signalizované vlákno je vzbuzeno, ale musí zažádat o zámek.

(Následuje pár bodů z implementace, viz http://baptiste-wicht.com/posts/2010/09/java-concurrency-part-5-monitors-locks-and-conditions.html)

- Je nutno použít třídy Lock a Condition.
    - lock() a unlock() na začátku a konci kritické sekce - ekvivalence synchronized (tam bysme ale neměli podmínkový proměnný), slouží pro vzájemné vyloučení, ale nikoliv pro synchronizaci, např. pro řešení problému producent - konzument
- Podmínka se vytvoří lock.newCondition().
- Na problém producent / konzument by se vytvořily dvě podmínky - ošetření plné a prázdné fronty.

X3. Závěr

Jedná se o zpracované otázky z předmětu KIV/PPR ke státnicím 2014.

Od loňska se témata docela měnily, čili něco je převzatý ze starších vypracovaných otázek, něco nově zpracovaný z přednášek a pár věcí doplněných z netu.

Good luck všem u státnic, především pokud u PPR narazíte na TXKoutného! :)

georgeee

Kdokoliv jste tyhle otázky použil, obětujte prosím pár kliknutí a napište to sem:

https://docs.google.com/forms/d/1ImCklJTEGjAPsC0eDMKbHYu7VPtHyeeOOuj0SgX7u7U/viewform