

Paralelizace cyklů

- Sekvenční algoritmus musí cyklem projít přes všechny části dat
- Lze provést dekompozici dat, můžeme cykly paralelizovat a tím urychlit výpočet
- => datový paralelismus SPMD

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

- Máme jeden programový kód, ale můžeme ho vykonávat na několika procesorech
 - Tj. vykonat co nejvíce smyček paralelně
- Je třeba určit
 - Lokální proměnné //a
 - jsou inicializována uvnitř smyčky
 - Sdílené proměnné
 - hodnoty se přenášejí mezi jednotlivými iteracemi
 - dělí se na nezávislé a závislé
 - nezávislé jsou tehdy, když //Items
 - jsou využívána pouze pro čtení
 - v případě pole jde pouze o jeden prvek, se kterým se pracuje pouze v jedné iteraci
 - závislé proměnné se dělí na //sum
 - redukční
 - uzamykané
 - uspořádané

- Redukční proměnná //sum
(reduction variable)
 - Nejprve je čtena, pak zapsána
 - Uvedené se odehraje v jedné iteraci
- Uzamykaná proměnná //min
(locked variable)

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

- Může být čtena i zapisována v několika iteracích
- Může být čtena i zapisován několikrát po sobě v jedné iteraci
- Pokud by se iterace neprováděly sekvenčně, ale v náhodném pořadí, výsledek by byl stále správný

```
for i:=1 to High(Items) do randomly
```

- Uspořádaná proměnná (ordered variable)
 - Správného výsledku je dosaženo pouze tehdy, jsou-li iterace vykonávány pouze ve stanoveném pořadí

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

- Na rozdíl od předchozích cyklů, tento nelze urychlit tak, že vlákna současně vykonají operace nad svou částí pole (a pak jedno z nich zpracuje mezivýsledky). Ačkoliv... ne vždy.

Paralelizace součtu pole

```
var GlobalSum:integer = 0;
    Items:array of integer;
    Guard:TCriticalSection;

thread DoPartialSum(Offset, Size:integer);
var i:integer;
    sum:integer = 0;
begin
    for i:=Offset to Offset+Size do
        sum:=sum+Items[i];

        Guard.Acquire;
        GlobalSum:=GlobalSum+sum;
        Guard.Release;
    end;

procedure RunThreads;
var i, j, k, offset, size:integer;
begin
    DivMod(Length(Items), NumberOfThreads, i, j);

    Offset:=0; Size:=i;
    for k:=1 to NumberOfThreads-1 do
        begin
            DoPartialSum(Offset, Size);
            Inc(Offset, Size);
        end;
    DoPartialSum(Offset, Size+j);
end;

begin program
    GetInputData;
    RunThreads;
    WaitForThreads;
    PrintGlobalSum;
end program;
```

Paralelizace výpočtu součtů prefixů

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

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

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

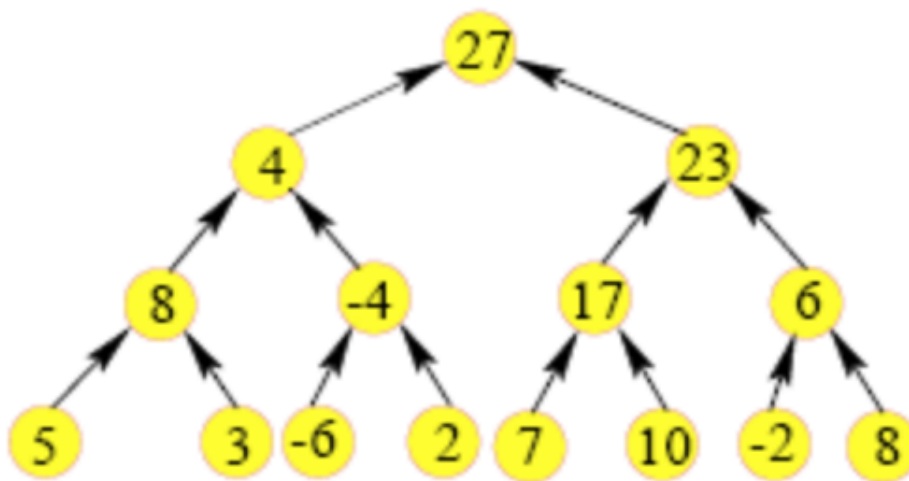
$$[4, 5, 3, 1, 6]$$

$$[4, 9, 12, 13, 19]$$

- Princip paralelizace výpočtu součtů prefixů se dá aplikovat i na některé další sekvenční výpočty, např.
 - Třídění
 - Lexikální analýza
 - Histogramy
 - Teorie grafů
 - Práce s řetězci
- I pro výpočet, který na první pohled vypadá jako beznadějně sekvenční, existuje šance, že ho bude možné paralelizovat
- První krok
 - Uspořádaná proměnná nám zabraňuje zapisovat do pole v jiném, než určeném pořadí
 - => zavedeme ještě jednu kopii pole
 - S $i-1$ už přistupujeme do jiného pole, do kterého se v cyklu nezapisuje => zrušeno uspořádání

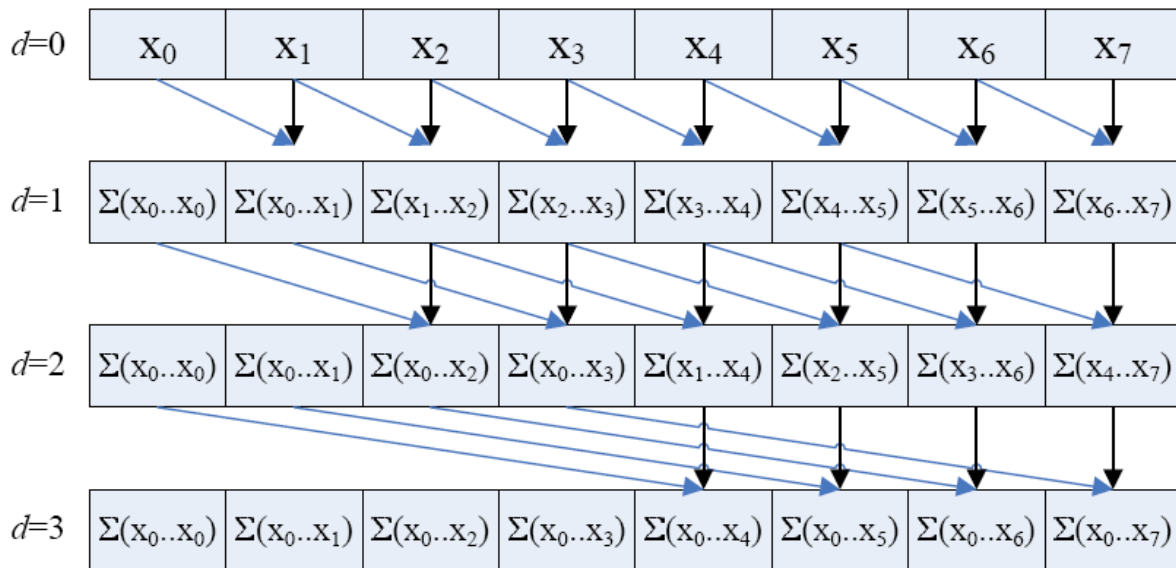
```
Move(Temp, Items, Length(Items));
for i:=1 to High(Items) do
  Items[i]:= Temp[i-1] + Items[i];
  //samozřejmě, že „něco“ chybí k úplnosti
```

- Krok druhý
 - V sekvenční verzi jsme s každým součtem získali jeden prefix
 - Výpočet n -tého prefixu se skládá z $n-1$ součtů
 - Protože už jsme se ale zbavili uspořádanosti, můžeme součet n -tého prvku rozložit do jiné posloupnosti součtů než v sekvenční verzi



<http://download.informatik.uni-freiburg.de/lectures/AdvancedAlgorithmsDatastructures/2006SS/Slides/thm14%20-%20parallel%20prefix.ppt>

- Pokud bychom měli k dispozici dva procesory, uvedený strom ukazuje, že výpočet posledního prvku můžeme paralelizovat
- Krok třetí
 - Jestliže lze paralelizovat výpočet posledního prvku, pak lze paralelizovat i výpočet ostatních prvků
 - Mezisoučty vznikají postupně v krocích – d
 - Významná část mezisoučtů je využita jako mezihodnota dva dalších mezisoučtů



<http://beowulf.lcs.mit.edu/18.337/lectslides/scan.pdf>

- Čtvrtý krok
 - Programový kód pro jedno vlákno
 - Hranice dat každého vlákna je daná rozmezím
Offset a Size

```

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

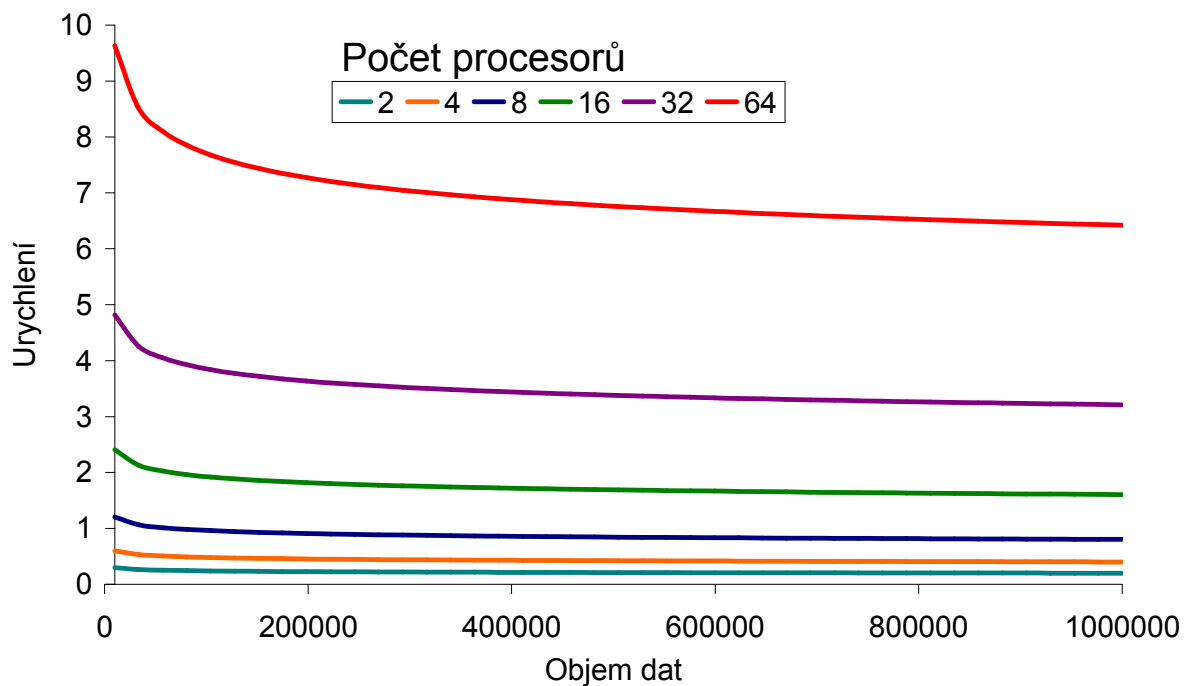
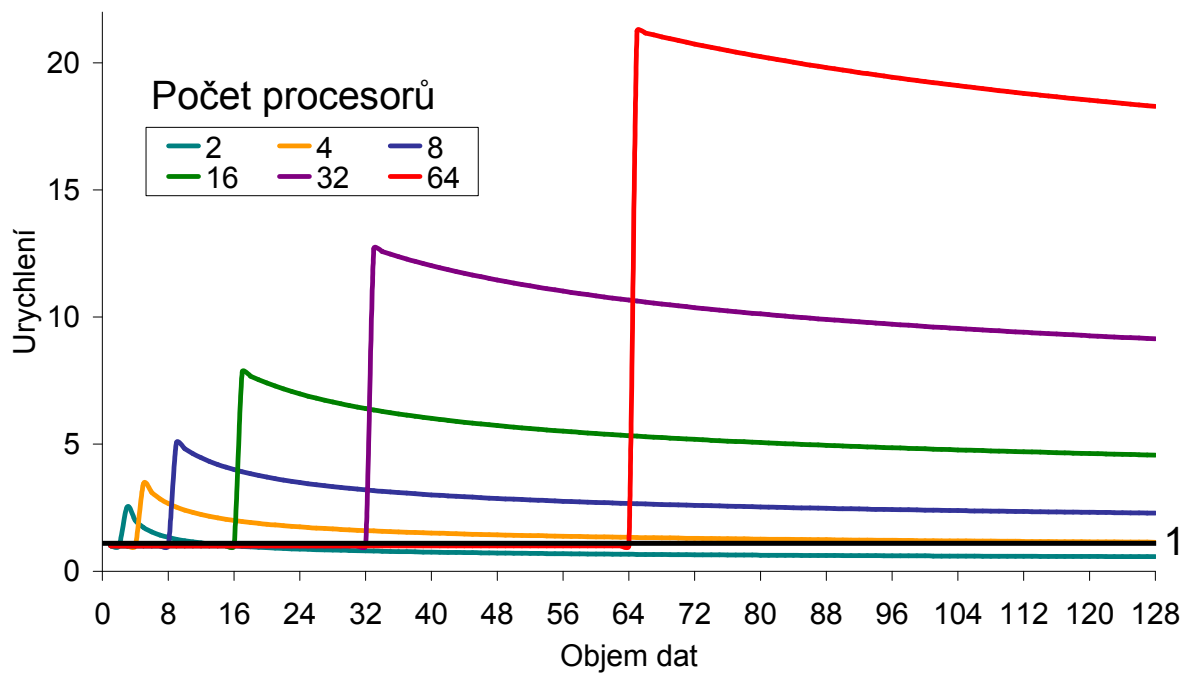
    Barrier;
    for i:=max(step, Offset) to Offset+Size do
      Items[i]:= Temp[i-step] + Items[i];

    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
- 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
 $\sim O(0,5 \bullet n \bullet \log_2 n)$
 - 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
 - Každé dvě vlákna, které se střídají o stejný procesor znamenají pouze nárůst instrukcí k vykonání, bez urychlení
 - Splníme-li podmínky vhodné paralelizace, pak s m procesory dostáváme $\sim O((0,5 \bullet n \bullet \log_2 n)/m)$
 - Se zanedbanou režii plánovače, nově přidaným kopírováním obsahu paměti – kopie pole, atd.



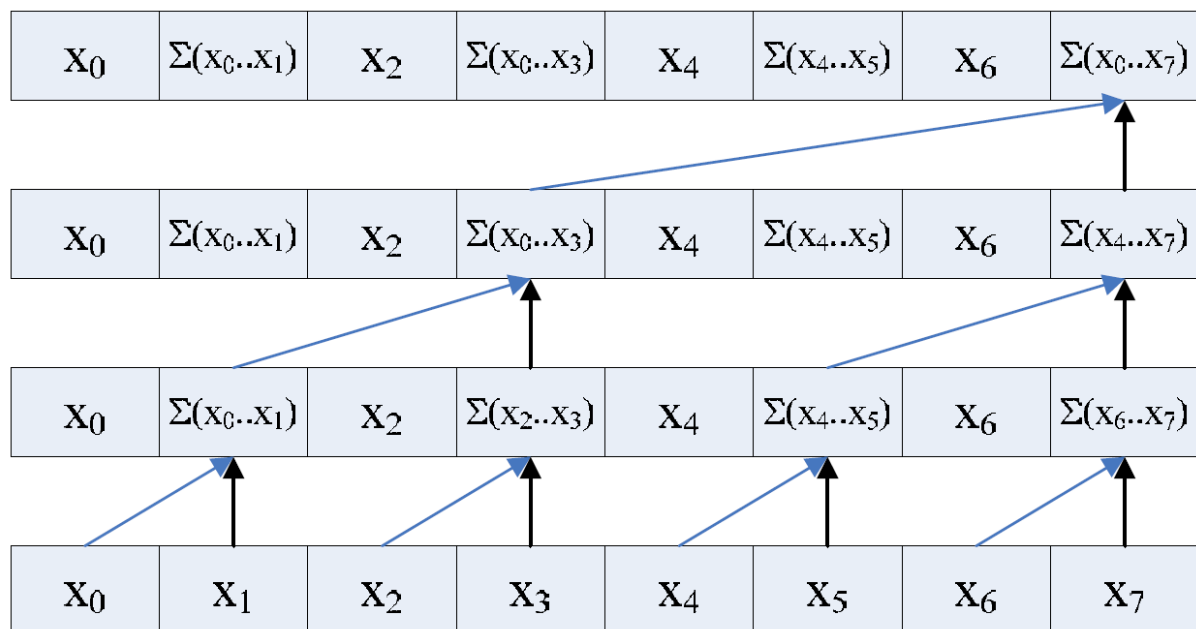
- Algoritmus je evidentně závislý na počtu procesorů a objemu zpracovávaných dat
- Hodí se spíše k HW akceleraci; např. nvidia G80 má limit pole 512, paralelně na ní běží 32 vláken
- Nicméně, podařilo se urychlit původní sekvenční algoritmus alespoň za nějakých podmínek – cena řešení?

- Optimalizace

- Ve skutečnosti lze provést ještě další úpravu algoritmu, kterou se s paralelní verzí dostaneme na složitost $O(n)$
 - Tj. stejně jako u sekvenční verze, jenomže už ji můžeme spustit paralelně
 - $O(n/m)$
- Navíc to celé provedeme in-place, stejně jako sekv.

- Krok první

- Redukční fáze – od zdola nahoru



<http://beowulf.lcs.mit.edu/18.337/lectslides/scan.pdf>

```
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;
```

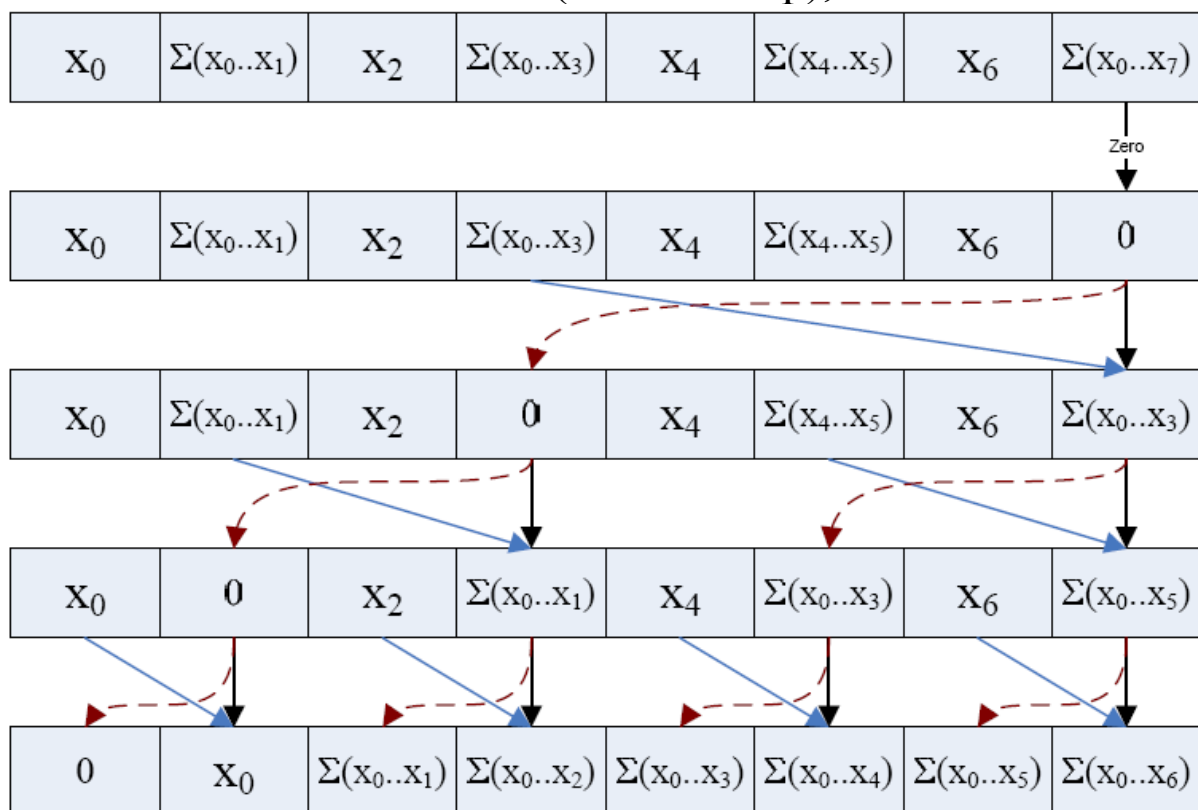
- Krok druhý
 - Záloha součtu posledního prvku
 - Vynulování posledního prvku pole
 - I když to ve skutečnosti dělat nemusíme

```

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

```

- Krok třetí
 - Fáze smetení (down sweep), od shora dolů



<http://beowulf.lcs.mit.edu/18.337/lectslides/scan.pdf>

```

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;
  Barrier;
  d:=d shr 1;
end;

```

- Krok čtvrtý
 - 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;

```

- 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
 - Aneb, i některé úlohy s „defaultně“ uspořádanými proměnnými lze paralelizovat – jen vymyslet jak:-)

Plánování vláken

- KIV/OS
- Dále budeme uvažovat plánování několika vláken na víceprocesorovém systému
- Z pohledu algoritmu
 - Klíčové otázky
 - Kolik vláken máme vytvořit?
 - Jak jim rozdělíme práci?
 - Jak poznat, že už je vše spočítáno?
 - Statické
 - Vytvoří se pevně daný počet vláken bez ohledu, kolik jich může fyzicky běžet najednou
 - Každé vlákno má pevně přidělenou část práce
 - Až ji vykoná, počká na ostatní vlákna, která ještě počítají
 - Dynamické
 - Vytvoří se tolik vláken, kolik jich fyzicky může běžet najednou
 - Vytvoří se seznam prací, ze kterého si budou vlákna postupně vybírat jednotlivé úkoly po dokončení předchozích úkolů
 - Zavedeme čítač aktivních vláken; poslední vlákno zjistí, že čítač je 1 (ono samo) a že seznam prací je prázdný
 - Dynamické plánování má sice nějakou režii navíc, lze ji však zanedbat, ale je univerzální
 - Vede k rovnoměrnému zatížení procesorů
 - Vlákna se nevytvářejí do zásoby – velká režie

- Z pohledu jádra OS a vyvažování zátěže
 - Statické
 - Dopředu jsou známy informace o vláknech, která budeme plánovat
 - Informace jsou s nějakou přesností považovány za dostatečně přesné
 - Na základě informací se vypočítá optimální plán, na kterém procesoru bude které vlákno spouštěno
 - Vlákna jsou pak za běhu alokována podle tohoto plánu na jednotlivé procesory
 - Jenže, který OS vám to umožní?
 - Jenže, optimální plán je úměrný přesnosti informací
 - Dynamické
 - Další režie s plánováním vláken za běhu
 - Lze zanedbat
 - Zajištěno jádrem OS
 - Počítá se se zatížením, které generují ostatní procesy v systému

Granularita

- Poměr objemu výpočtu k objemu komunikace
- Čím jemnější (Fine-Grained), tím menší objem dat je vypočítán mezi komunikací dvou vláken => častá komunikace o malých objemech
- Čím hrubší (Coarse-Grained) – opak jemnější
- Jemnější umožňuje vyšší paralelizaci, ale je třeba „zjemňování zastavit včas“, jinak se výpočet začne zpomalovat díky komunikační režii a režii synchronizace

Urychlení součtu hodnot pole

- Co je rychlejší?
 - Postupný součet všech hodnot pole
 - Nebo fáze odspoda – nahoru součtů prefixů?

Výpočet druhé difference

- V praxi může být funkce zadána konečnou posloupností hodnot, místo předpisem
- Diferenční počet je analogie k diferenciálnímu počtu
- Máme reálnou funkci f
- $x, h, \in \mathbb{R}$
- diferenční krok: $h > 0$
- Difference funkce f v bodě x
 - $\Delta f = f(x+h) - f(x)$
- Druhá difference posloupnosti a_n
 - $\Delta^2 a_n = \Delta(\Delta a_n)$
- $\Delta^0 a_n = a_n$
- $\Delta^k a_n = \Delta(\Delta^{k-1} a_n)$

- Algoritmus výpočtu druhé difference

```
for i:=1 to High(Items) do  
  begin  
    d1[i]:=Items[i]-Items[i-1];  
    d2[i]:=d1[i]-d1[i-1];  
  end;
```

- Co je lepší?

- Použít velmi jemnou granularitu pro dva procesory
 - Jeden bude počítat d1, druhý d2
 - Jakou režii bude mít synchronizace?
- Nebo nasadit řešení producent-konzument
 - Items jsou hodnoty vkládané do bufferu
 - d1 je buffer
 - d2 jsou hodnoty zkonsumované z bufferu
 - Kolik procesorů dokážeme tímhle způsobem využít? 2?
- Anebo nasadit řešení podle výpočtu sum prefixů?
 - Dva průchody, jeden vypočte d1, druhý d2
 - Nebo využít, že je možné paralelně vypočítat 3 sousední hodnoty d1 a z nich 2 sousední hodnoty d1
 - A navrhnout vlastní řešení, které to vypočítá na jeden průchod?