Složitost, urychlení, metriky paralelizace, účinnost a korektnost paralelního výpočtu

Složitost

- je funkce *f(n)*, jejíž hodnota ukazuje dobu výpočtu nebo systémové nároky (paměť) algoritmu nebo programu v závislosti na velikosti vstupních dat (nejhorší hodnota ze všech možných kombinací dané velikosti vstupních dat)
- např. sekvenční algoritmus pro součet *n* čísel má složitost *f(n) = n*, protože maximální doba výpočtu je úměrná počtu čísel *n*
- složitost se označuje písmenem *O*, v předchozím případě tedy *O(n)* znamená, že doba výpočtu je lineárně závislá na počtu čísel *n*.

Asymptotická složitost

klasifikace algoritmů, třídy složitosti: O(1) - konstantní, O(log n) – logaritmická, O(n) - lineární, O(mxn), $O(n^2)$ – kvadratická, $O(2^n)$,$O(n^m)$ - exponenciální

Metriky

Způsoby srovnání rychlosti výpočtu paralelního algoritmu se sekvenčním.

Amdahlův zákon

Amdahlův zákon říká, že nelze dosáhnout urychlení většího než $1/f$ (a prakticky že výpočet zpravidla nelze paralelizovat úplně). Uvažuje pouze tradiční způsob postupné paralelizace.

buď $0 \leq f \leq 1$ část výpočtů, které nelze paralelizovat (musejí se provádět sekvenčně). Maximální urychlení $S(n,p)$ dosažitelné při použití p procesorů je:

$$S \leq \frac{p}{1 + f \times (p-1)} \leq \frac{1}{f}$$

⇨ **Maximální urychlení je 1/f**

Příklad pro 8 procesorů a 10% neparalelizovatelného kódu:

$$S(n,p) \leq \frac{8}{1 + 0{,}1 \times (7)} \cong 4{,}71$$

A to je o hodně míň než 8.

význam: např. pro program s 10% sekvenční částí nelze dosáhnout víc než desetinásobného urychlení. => paralelizace hlavně umožňuje řešit rozsáhlejší úlohy spíš než nekonečně zrychlovat.

Gustafsonův zákon

Hodně procesorů a hodně dat, sériově prováděný kód je zanedbatelný.

$$S(P) = P - \alpha\,(P-1)$$

kde P určuje počet procesorů, S je zrychlení a α značí sekvenční složku.

Význam: některé úlohy (s malou datovou základnou) nestojí zato paralelizovat.

Dá se dokázat že zákony Amdahlův a Gustafsonův jsou identické.

Urychlení

Poměr doby výpočtu nejlepším známým sekvenčním algoritmem a doby výpočtu paralelním algoritmem na témže počítači

Anomální urychlení
Použití více procesorů nemusí znamenat jen znásobení výkonu (počtu provedených instrukcí) za jednotku času, urychlení může být vyšší než lineární díky:

- potřebě menšího počtu přepnutí kontextu
- dekompozici prohledávání (dřívější nalezení díky rozdělení úlohy)
- efekt lokálních cache pamětí (rozdělená úloha se vejde do cache, takže není potřeba provádět tolik paměťových operací)
- použití efektivnějších algoritmů pro multiprocesorový systém (paralelizovaný algoritmus je odlišný)

Účinnost

Poměr skutečně dosaženého urychlení k maximálnímu dosažitelnému urychlení ($1/f$).

Příklad: sekvenční výpočet trvá 10 s, paralelní na 4 procesorech 5 s, urychlení je 2,0 a účinnost je 0,5.

Korektnost paralelního výpočtu

Korektní běh: program musí pokaždé doběhnout a skončit se stejným výsledkem (i kdyby ten výsledek nebyl dobře).
Logicky korektní: program dává správný výsledek.

Základní programové modely pro paralelizaci výpočetní činnosti (SIMD, SPMD, MPSD, MPMD)

SIMD – single instruction multiple data

Vektorové počítače, založené na zpracování více datových toků jednou instrukcí.

Příklad: maticové a vektorové výpočty na GPU. Cluster herních konzolí nakoupených se slevou.

SPMD – single program multiple data

dekompozice dat – paralelizace cyklů, pokud není vázaná proměnná

- několik procesorů autonomně vykonává jeden program nad různými daty
- používá se v případě, kdy relativně jednoduchá činnost je prováděna nad objemnými daty
- vhodný pro víceprocesorové nasazení, při pseudoparalelním výpočtu (na jednom jádře) by došlo spíš ke zpomalení (přepínání vláken)
- příklad: monte carlo, iterační numerická řešení

MPSD – multiple program single data

- několik různých procesů zpracovává data v jednom datovém proudu
- zřetězené zpracování dat (analogií z běžného života je montážní linka)

- jedná se o zpracování rozsáhlého proudu datových prvků, přičemž nad jednotlivými prvky jsou vykonávány nějaké (libovolně složité) operace, které je možné svěřit různým specializovaným procesům a vykonávat je paralelně pro několik prvků datového proudu.

Použití:

- v tzv. Pipeline architektuře - např. grafická karta je zařízení optimalizované pro proudové zpracování dat
- pro výpočty odolné proti poruchám - několik různých systémů zpracovává ty samá data a musejí se shodnout na výsledku – např. řízení letu raketoplánu

MPMD – multiple program multiple data

- Farmer – worker
- Vhodné pro složitou činnost a málo objemná nebo nehomogenní data
- Vhodný k implementaci i na jednoprocesorovém stroji
    - Nemusí jít jen o urychlení, ale třeba o lepší strukturu
    - Třeba rozdělení dat na vektorová pro GPU stroje a ostatní pro normální
    - Vícevrstvé aplikace – GUI ve zvláštním vlákně (např. progress bar načítání z pásky)

Paralelizace cyklů

- Pokud lze provést dekompozici dat, můžeme cykly paralelizovat (SPMD)

Příklady

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

Rozdělení proměnných

- **Lokální proměnné** – **a**
- **Sdílené proměnné**:
    - **Nezávislé** – jsou využívány pouze pro čtení – **Items**
    - **Závislé**:

- **Redukční** –jsou v jedné iteraci čteny i zapsány – **sum** (redukční – může být výsledkem redukčního stromu)
- **Uzamykatelné** – Mohou (ale nemusí) být čteny i zapisovány v každé iteraci (a to 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]**

Paralelizace cyklu se závislou uspořádanou proměnnou

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

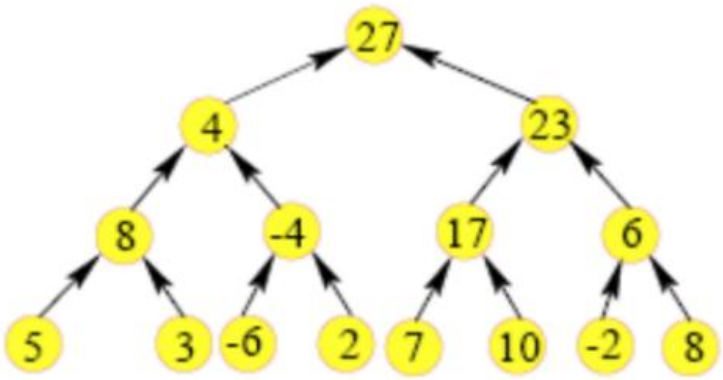
Klasický příklad – **paralelní prefixový součet**

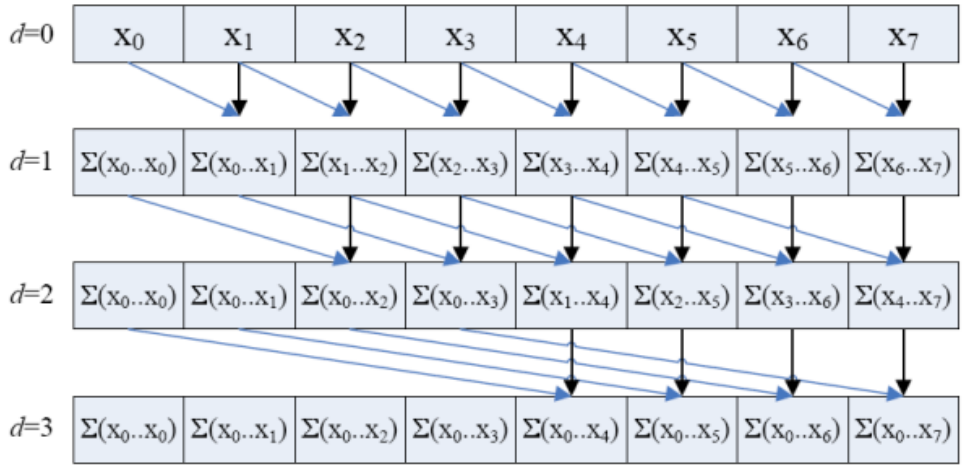
1) Zavedeme si pomocnou kopii pole vstupních prvků (Temp) a z ní vybíráme prvek **i – 1**. Tím jsme zrušili uspořádání.

   Hlavní myšlenka:

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

   A aby se to dalo paralelizovat, je potřeba rozdělit výpočet do stromu.






```
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;
```

Pro lepší pochopení příkladu jsem to přepsal do foru, aby bylo vidět že ten step není sdílený. Tento výpočet je pro počet vláken rovný počtu prvků vstupního pole (8), fakt je to maximální zjednodušení.

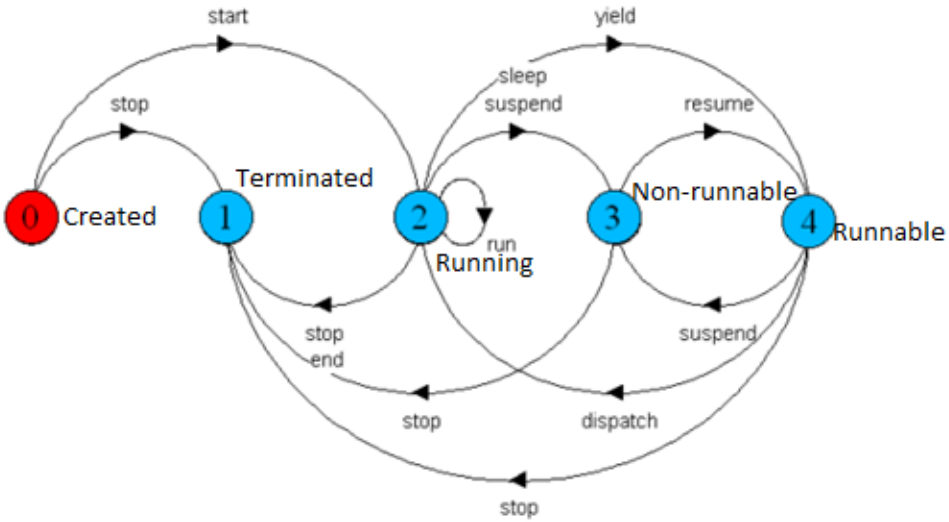
```
for i = 0; i < 4; i++ {
      move //shared
      barrier;
      items [i^2] = previous [(i-1)^2] + items [i^2];
      // zde může být bariéra a závislá uspořádaná proměnná (jako řídící)
}
```

Programové prostředky pro multithreading: Java, rozhraní POSIX pro vlákna v jazyce C, podpora vláken ve WinAPI.

Java

- Paralelně (ve vláknu) spustitelná třída dědí od Thread nebo implementuje rozhraní Runnable

    - Dědit je vhodné pouze pokud potřebuji překrýt nějakou vlastnost, jinak je lepší implementovat rozhraní (a dědit od jiné třídy, když je potřeba)
    - Public void run()
- Implementace paralelismu (fiber vs. LWP záleží na virtuálním stroji)
- stavy vlákna:
    - **Nové vlákno** – vlákno bylo vytvořeno, ale dosud nebylo spuštěno metodou start()
    - **Běhuschopné** – metoda start() už proběhla; těchto vláken může být více, ale na jednoprocesorovém stroji je vždy jen jedno běžící, ostatní musí čekat na předání řízení
    - **Neběhuschopné** – vlákno, které bylo uspáno metodou *sleep()*, nebo čeká na *wait()*, nebo čeká na I/O
    - **Mrtvé vlákno** – vlákno, jehož metoda run() již skončila



- **Monitor**
    - kritické sekce jsou ošetřeny klíčovým slovem *synchronized* (obvykle metoda, synchronized může být i blok, ale to se nedoporučuje)
    - monitor je implicitně součástí každého objektu
    - 1 zámek, 1 fronta
    - Wait() – pozastaví vlákno na monitoru objektu dokud není zavoláno notify,
    - notify() – vzbudí vlákno na příslušném monitoru,
    - notifyAll() – Vzbudí se všechna vlákna (implementace bariéry)
- Proměnná **volatile** – nemá se cachovat, protože do ní zapisuje několik vláken (aby se nepracovalo s neaktuální hodnotou)
- Metody yield, join, run, setPriority, sleep; některé metody jsou deprecated protože mohou způsobit nekonzistentní stav (stop, suspend)
- **Výhody**
    - Vysokoúrovňový monitor
        - obvykle postačuje
        - dají se na něm postavit jiné metody (bariéra…)
        - není potřeba moc řešit vstup do kritické sekce
- **Nevýhody**
    - Nutnost tvořit synchronizační prostředky s pomocí Monitoru
    - Záleží na virtuálním stroji, jestli používá fibers nebo kernel thready

        - Když VM nepodporuje jádrová vlákna, nelze využít vícejádrový procesor
    - Stále je možné docílit deadlocku (to je ale skoro všude)

Rozhraní POSIX

Portable Operating System Interface – standard pro přenositelnost mezi operačními systémy (hlavně UN*Xového typu, cygwin)

- Sada nástrojů pro práci s vlákny v jazyce C
- Cca 100 procedur „pthread_“
    - Vytvoření, spuštění, join…
    - Mutexy
    - Zámky, bariéry
    - Podmínkové proměnné…
- 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)
- vlákno běží hned po vytvoření
- stavy vlákna jsou ready, running, waiting a terminated
- join vs. detach (join čeká na konec vlákna, detach ho nechá žít vlastním životem a jen po něm sebere prostředky, po detach už se nedá zavolat join)
- **mutex**
    - tři typy zámků:
        - normální – konkrétní vlákno ho může zamknout jen 1x
        - rekurzivní – konkrétní vlákno ho může zamknout vícekrát (pro rekurzivní zpracování globálních dat)
        - ladicí – *ERRORCHECK*; Pozná se opakované zamčení
    - lock, unlock a try_lock
    -

WinAPI

- preemptivní multitasking
- preemptivní multithreading
- podpora kooperativních fibres jádrem
- Oproti všemožným datovým strukturám POSIXu má WinWPI 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)
- Má hodně různých stavů vlákna, obecně je lze rozdělit do čtyřstavového modelu (ready, running, waiting, terminated)
- Thready i Fibery
    - Pro Thread zajišťuje plánování, inverzi priorit apod
    - Fiber je plánován threadem, nemá prioritu
    - Thread-local storage, Fiber-local storage
    - Thready sdílí adresový prostor a zdroje procesu, Fibery běží v kontextu threadu
    - Mnoho funkcí řízení stavu threadu a fiberu
- Synchronizační objekty:

    - Event (pulzní, přepínací)
    - Mutex
    - Semafor
    - Časovač
- 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

Konstrukce jazyka Ada pro paralelní programování

- Ada je jazyk navržený pro vysokou bezpečnost (např. staticky typovaný)
- Paralelní bloky se označují jako Task
- Pro synchronizaci Tasků se používá Rendezvous
    - task je uspán do té doby, než se dostaví druhý task, který s ním chce komunikovat (body komunikace definují entry calls)
- Task je vytvořen v momentě kdy je definována jeho instance
- Pro vytvoření více instancí tasku je nutné jej definovat jako type (viz níž)

Deklarace Tasku

```
Task [type] jméno is
        // deklarace jmen komunikačních typů
        // viz příklad dále
end jméno;
task body jméno is
        // lokální deklarace a příkazy
end jméno;
```

Rendez-vous

- Task Server definuje v jakém pořadí Entry Calls přijímá.
- Je-li potřeba reagovat na různé entry calls, použije se konstrukce se select (viz níž)
    - Jinak může dojít k deadlocku při jiném pořadí volání než jak je čeká server – „server“ čeká na jeden typ entry callu a „klient“ se zablokuje voláním jiného entry callu
- Task (konstrukce accept) je implicitní synchronizační konstrukce

```
task Server is
  entry Start(Num : in Integer);
  entry Report(Num : out Integer);
end Server;

task body Server 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 Server;
```

syntaxe selectu

```
select
        <entry call>;
or
        <entry call>;
else
        <entry call>;
end select;
```

Protected Objects, Protected Types

- Protected moduly jsou lehčí než tasky (vytvoření tasku jen pro ochranu proměnné je zbytečně náročné na zdroje)
- tasky mohou sdílet objekty
- objekt je instance typu – klíčové slovo *type*
    - Lze zavést defaultně protected Type, nebo jen jeden konkrétní protected Object
- 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 (např. inline *counter = counter + x*); 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 (není deklarováno protected)

```
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;
```

Výpočetní prostředí s distribuovanou pamětí – charakteristika a principy realizace základních modelů paralelního výpočtu.

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

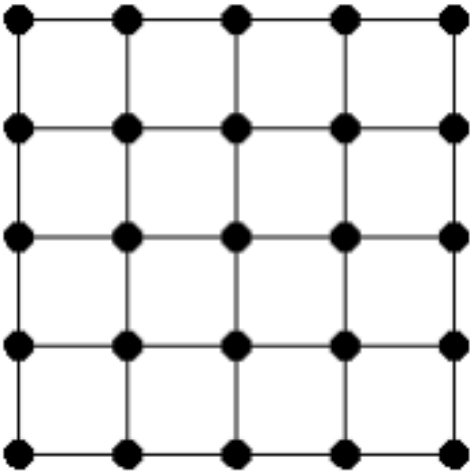
protože systémy s distribuovanou pamětí nemají žádný úzký profil ve formě sběrnice, přes kterou by procesory přistupovaly ke sdílené paměti, hodí se pro úlohy vyžadující tzv. masivní paralelismus (stovky až tisíce procesorů)

Obecně systém s distribuovanou pamětí umožňuje větší urychlení než systém se sdílenou pamětí díky paralelizaci komunikace

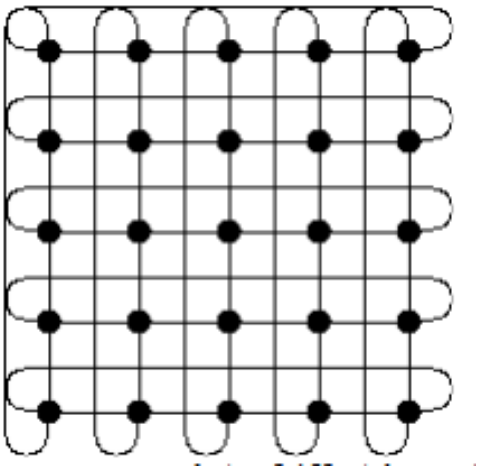
- Zatímco se data přenášejí kanálem, uzel může počítat
- Urychlení ovšem závisí na dalších parametrech
  - Objemu interakce
  - Celkovém objemu zpracovávaných dat
  - Konkrétní hw architektura
  - Jak dalece je použitý programový kód optimální pro danou architekturu

HW pohled

- Topologie obecně
  - Pravidelná: kdychle, mřížka, hvězda
  - Nepravidelná: Internet
- Fyzická topologie
  - Pevná: procesory jsou propojeny komunikačním kanálem
    - **Adresa může reflektovat polohu v síti**
    - Každý s každým
    - 2d mřížka (dmax = $2(n-1)$)



- Toroid (dmax = n-1)



- 3d mřížka, n-rozměrné krychle…

    - Flexibilní: přepínání okruhů, přepínání paketů

Parametry

- N: Celkový počet uzlů v síti
- dij: vzdálenost mezi dvěma uzly (sousedé mají 1)
- dmax: 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, než se z jedné sítě stanou 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

- Dosáhnout 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)

Charakteristika a porovnání výpočetních nástrojů PVM a MPI, příklady použití

PVM i MPI se používají v prostředí s distribuovanou pamětí.

- Oba mohou běžet v heterogenním prostředí
  - PVM vzniklo pro heterogenní sítě
  - MPI navrženo pro clustery
- MPI vzniklo z PVM a vychází z něj
  - MPI nabízí vyšší a jednodušší abstrakce a možnosti pro přenos zpráv

    - PVM má lepší podporu nízkoúrovňových rutin, MPI se tomu vyhýbá kvůli přenositelnosti
- PVM se nestará o topologii, MPI podporuje logické topologie
- MPI má v návrhu snahu o omezení kopírování paměťových bloků

PVM

- pro paralelní počítače různého typu (tj. z pohledu programátora se jedná o programovací prostředek)
- ve formě knihoven v programovacích jazycích C, Fortran a Java, předpona pvm_
- Pro heterogenní sítě
- Nízkoúrovňový prostředek
    - Výměna zpráv asynchronní přes vyrovnávací paměti
    - Jedna aktivní vyrovnávací paměť pro příjem a jedna pro odesílání
- PVMD – démon běžící na pozadí v každém počítači připojeném do sítě
- Počítače v PVM síti jsou identifikovány svým síťovým jménem
- Na jednom stroji několik procesů jedné aplikaci nebo i víc démonů (každý pro jednu aplikaci)
- PVM konzola
    - Přidávání a odebírání počítačů
    - Spouštění a ukončení výpočtů
    - Konfigurace, výpis běžících procesů, výpisy stavu…
- Typicky farmer-worker

```
#include <stdio.h>
#include <pvm3.h>
int main() {
     int mytid;
     mytid = pvm_mytid();
     printf("My TID is %d\n", mytid);
     pvm_exit();
     return 0;
}
```

MPI

- Zase C, fortran, Java, navíc i .NET
- Vyšší oproti PVM
- Ideální pro výpočty nad regulárními daty (matice, vektory, homogenní z hlediska datových typů)
    - Poskytuje prostředky pro „synchronní“ nebo step-locked algoritmy (statické rozdělení práce)
    - Preferuje SPMD model – vyrobí se jeden spustitelný program a ten se roznese po všech uzlech
        - Farmer-workers lze realizovat tak, že proces s nejnižším číslem se ujme řízení
    - Obsahuje globální funkce pro rozdělení dat mezi procesy (MPI_Scatter…)
- Má svoje vlastní primitivní datové typy a z nich je možno složit struktury (lepší přenositelnost a spolehlivost v heterogenním prostředí)

- Má sadu funkcí pro globální operace (MPI_Barrier, MPI_Alltoall…)

```
#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();
}
```