1. Ciele paralelného programovania paralelných výpočtov a vzťah k programovaniu systémov reálneho času. Charakteristika a využitie paralelných architektúr. Vyváženosť a dodatočné náklady pri paralelných výpočtoch.

Cieľom programovania **systémov reálneho času** je:
• uspokojenie požiadaviek na čas vykonávania systému
• zabezpečenie spoľahlivosti systému z hľadiska správnosti jeho funkcie

Pri programovaní systémov reálneho času je často používaný pojem *súbežné programovanie*. namiesto pojmu paralelné programovanie.

Cieľom súbežného programovania je špecifikácia komunikujúcich sekvenčných procesov v systéme, pričom nie je podstatné, aby výsledný systém bol implementovaný na paralelnom počítači.

Paralelizmus využívaný v systémoch reálneho času je zvyčajne hrubozrnnejší ako pri paralelných výpočtoch.

Podľa toho, na akej cieľovej architektúre je systém reálneho času vykonávaný, hovoríme o multiprogramovaní, viacnásobnom spracovaní alebo distribuovanomprogramovaní. Ich vzťah:

P	Pseudoparalelizmus	Paralelizmus	
PR	Multiprogramovanie	Viacnásobné spracovanie	Distribuované spracovanie
A	Jednoprocesorová architektúra s prideľovaním času	Viacprocesorová architektúra so spoločnou pamäťou	Viacprocesorová architektúre bez spoločnej pamäti

Cieľom výkonných **paralelných výpočtov** je teda:
• podstatné zníženie času výpočtu
• spracovanie veľkého množstva údajov
• dosiahnutie tých istých výsledkov ako na sekvenčnom počítači, avšak pomocou viacprocesorovej architektúry a v čo *najkratšom čase*.

Na rozdiel od systémov reálneho času nie je potrebné uvažovať o žiadnom dostatočnom čase vykonávania, pretože cieľom je dosiahnuť vykonávanie v čo najkratšom čase.

V súvislosti s výkonnými paralelnými výpočtami ide vždy o paralelné programovanie určené pre paralelné počítačové architektúry (vyplýva nevyhnutnosť využitia paralelnej počítačovej architektúry).

Paralelné programovanie má aj svoje nevýhody, najmä tieto:
• Nie všetky problémy možno efektívne paralelizovať
• Medzi efektívne paralelizovateľným problémom a druhom paralelnej architektúry je silný vzťah
• Ak je problém efektívne paralelizovateľný, je potrebné ho dekomponovať

Flynnova klasifikácia paralelných počítačových architektúr je založená na štyroch základných architektúrach: SISD, SIMD, MISD a MIMD.
SISD - existuje počas vykonávania jediný prúd inštrukcií a jediný prúd údajov, znamená to, že v určitom okamihu jedna inštrukcia spracováva jeden údaj a teda ide o sekvenčný jednoprocesorový počítač, ktorý sa pri výkonných paralelných výpočtoch nepoužíva.

SIMD – charakteristická veľkým množstvom jednoduchých procesorov, z ktorých každý má lokálnu pamäť pre údaj, ktorý spracováva. Každý procesor vykonáva súčasne tú istú inštrukciu na jej patriacom lokálnom údaji, pokračujúc ďalšími inštrukciami.
Výhodou architektúry SIMD je to, že je vhodná pre riešenie masívne paralelných problémov, kde tá istá operácia je vykonávaná na veľkom počte rozličných objektov.
Nevýhodou je to, že ak záťaž procesorov nie je vyvážená (napr. pri nerovnorodých problémoch s hrubším stupňom paralelizmu), výkonnosť je slabá, pretože vykonávanie je synchronizované v každom kroku, čakajúc na najpomalší procesor.

MISD - obsahuje špecializované rýchle procesory, komunikujúce prúdovým spôsobom.
Vzhľadom na obmedzený aplikačný záber, MISD sa používa zriedkavo.

MIMD - Architektúra MIMD pozostáva obvykle z menšieho počtu nezávislých procesorov ako SIMD. Tieto procesory sú schopné vykonávať nezávislé prúdy inštrukcií, a teda môžu vykonávať aj rozdielne programy.
Rozšírenie základnej Flynovej klasifikácie spočíva v ďalšom rozčlenení architektúr MIMD na:

• *architektúry s distribuovanou pamäťou (Distributed Memory – DM)*

- každý procesor má svoju vlastnú lokálnu pamäť, procesor má prístup iba do tejto svojej lokálnej pamäti, komunikácia medzi procesormi prebieha výlučné prostredníctvom správ, keďže tu neexistuje žiadna spoločné používaná pamäť.

Výhody:

- škálovateľnosť omnoho viac ako architektúra MIMD so spoločnou pamäťou.
- čím viac rastie nepravidelnosť problému, tým je podpora škálovateľnosti vyššia ako pri architektúre SIMD

Nevýhody:

- výkonnosť závisí na štruktúre a priepustnosti prepojovacej siete, keďže pri vzraste fyzickej vzdialenosti medzi procesormi rastie čas prístupu ku vzdialeným údajom.

• *architektúry s virtuálnou spoločnou pamäťou (Virtual Shared Memory – VSM)*

- Koncepčné je kombináciou architektúr s distribuovanou a spoločnou pamäťou, realizovaná je však fyzicky ako architektúra s distribuovanou pamäťou.
- Priamy prístup ku vzdialenej pamäti je realizovaný vyhradeným spoločným adresným priestorom a podpornými obvodmi, ktoré zabezpečujú komunikáciu nezávisle na vzdialenom procesore.
- Rýchlosť komunikácie je veľmi vysoká, vzhľadom na technické riešenie na veľmi vysokej úrovni. Napriek tomu s rastúcimi vzdialenosťami medzi procesormi sa komunikácia spomaľuje rovnako ako pri architektúre MIMD s distribuovanou pamäťou

• *architektúry so spoločnou pamäťou (Shared Memory – SM)*

- obsahuje menší počet procesorov, z ktorých každý má prístup do globálnej pamäti prostredníctvom zbernice alebo iného druhu prepojenia

Výhody:

- je jednoducho programovateľná, pretože tu neexistuje žiadna explicitná komunikácia medzi procesormi navzájom a prístup ku globálnej spoločnej pamäti možno riadiť pomocou techník známych pre viacnásobné spracovanie, napr. semaforov

Nevýhody:

- nie je dostatočné škálovateľná (s rastúcim stupňom paralelizmu problému nerastie úmerne efektívnosť vykonávania) pre jemnozrnne paralelné problémy, a to kvôli „hladovaniu", ktoré vzniká v dôsledku častej potreby na prístup k spoločnej pamäti.

Súhrn nákladov pri paralelnom výpočte

• Náklady na komunikáciu a synchronizáciu zahŕňajú čas spotrebovaný pri výpočte na komunikáciu a synchronizáciu.

• Ťažko paralelizovateľné sekvenčné časti kódu.

• Algoritmické náklady: Paralelné algoritmy, použité namiesto najrýchlejších sekvenčných algoritmov môžu viesť v porovnaní s nimi k väčšiemu počtu operácii počas vykonávania.

• Náklady pri programovaní: Paralelizácia často vedie k vzrastu nákladov pri programovaní, spojených s indexovaním, volaniami procedúr, skracovaním cyklov, ktoré obmedzuje potenciálny zisk z vektorizácie, apod.

• **Nevyváženosť záťaže**: Čas vykonávania paralelného algoritmu je určený časom vykonávania na procesore, ktorý pracuje s najväčšou záťažou. Ak záťaž nie je vhodne rozdelená medzi procesory, vzniká nevyváženosť záťaže, prejavujúca sa nečinnosťou tých procesorov, ktoré musia čakať naprázdno na iné procesory, kým neukončia svoj čiastkový výpočet.

2. Vlastnosti paralelných algoritmov a paralelných problémov, definícia inherentného, ohraničeného a neohraničeného paralelizmu, tried efektívne a optimálne paralelizovateľných problémov a téza paralelného výpočtu.

Inherentný paralelizmus - je paralelizmus vlastný problému alebo algoritmu. Ak je problém inherentne paralelný, je riešiteľný nielen sekvenčným algoritmom, ale aj paralelným algoritmom. Dva algoritmy tej istej funkcie ktoré riešia ten istý problém môžu mať rôzny stupeň inherentného paralelizmu.
Neohraničený paralelizmus - algoritmu je vyjadrený počtom paralelných krokov, t.j. paralelným časom výpočtu, neberúc pritom do úvahy zdroje počítačového systému.
Ohraničený paralelizmus - je vyjadrený paralelným časom výpočtu, berúc do úvahy (ohraničené) zdroje počítačového systému.

Triedy efektívne a optimálne paralelizovateľných problémov
Nech *k* je celočíselná konštanta a *n* je veľkosť problému.

> Potom **efektívny paralelný algoritmus** je vykonávaný v polylogaritmickom čase $(O(\log^k n))$ pri použití polynomiálneho počtu (n^k) procesorov.

Problémy riešiteľné pomocou efektívnych paralelných algoritmov patria do triedy NC (Nick (Pippenger)'s class).

> **Optimálny paralelný algoritmus** je algoritmus, pri ktorom súčin $p.T$ paralelného času T a počtu procesorov p rastie s veľkosťou problému n lineárne.

Optimálnosť tiež znamená, že $p.T = T_S$, kde T_S je čas výpočtu pri použití najrýchlejšieho známeho sekvenčného algoritmu pre daný problém.

Ak má byť problém riešený paralelným spôsobom, potom nemusí preň existovať nevyhnutne optimálny paralelný algoritmus. Musí však preň existovať prinajmenšom efektívny paralelný algoritmus, t.j. problém musí patriť do triedy NC. Na druhej strane, k ťažko paralelizovateľným problémom patria:
• P-úplné problémy (vykonávané v polynomiálnom sekvenčnom čase)
• NP-úplné problémy.

Téza paralelného výpočtu
Nech *F* je ľubovoľná funkcia veľkosti problému *n*.

> Potom podľa tézy paralelného výpočtu trieda problémov, ktoré možno riešiť s neohraničeným paralelizmom v čase $F(n)^{O(1)}$ je zhodná s triedou problémov, ktoré možno riešiť sekvenčne v pamäti veľkosti $F(n)^{O(1)}$.

Téza paralelného výpočtu ukazuje teda na *dôležitosť pamäti pri paralelnom výpočte*, pretože, neformálne povedané, koľkokrát pri paralelnom výpočte oproti sekvenčnému klesne čas výpočtu, toľkokrát vzrastú nároky na pamäť.

3. Dekompozícia paralelných problémov vo vzťahu na jednotlivé druhy paralelizmu, využitie paralelizmu v aplikáciách vysokého výkonu a zodpovedajúce programové modely a paralelné architektúry, ich výhody a nevýhody.

Dekompozícia paralelných problémov

• Efektívny paralelný problém možno rozoznať vo vzťahu pre idealizované typy, ktoré sú vlastné tomuto problému.

• Na základe dekompozície problému vznikne množina nezávislých alebo vzájomne komunikujúcich procesov (úloh), ktorú možno priradiť vhodnej paralelnej architektúre vyváženým spôsobom, redukujúc pritom čo najviac náklady.

Všeobecné pravidlá dekompozície:

Po priradení úlohy procesoru musia byť zdroje systému dostatočné na to, aby nedošlo k prudkému zníženiu výkonnosti alebo dokonca k zrúteniu výpočtu. Preto je potrebné predchádzať nasledujúcim nežiadúcim javom:

• presunom častí programu medzi hlavnou pamäťou a diskovou pamäťou v dôsledku nevhodnej záťaže procesorov, spôsobeného najmä:

– preťažením v dôsledku nevhodnej dekompozície

– preťažením systému v počítačových klastroch pri využití operačného systému s prideľovaním času

• presunom častí programu medzi hlavnou pamäťou a diskovou pamäťou v dôsledku nedostatočnej pamäti pre spracovávané údaje

• nevyváženému použitiu blokujúcich, resp. neblokujúcich procedúr pre odovzdávanie správ

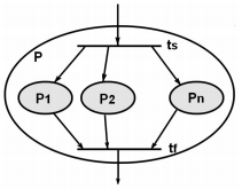
• zrúteniu výpočtu kvôli zablokovaniu

• zrúteniu výpočtu kvôli nedostatku pamäti

Jednoduchý paralelizmus

• *Problém P je možné rozdeliť na množstvo podproblémov n, ktoré možno riešiť nezávisle.*

• Výpočet končí vtedy, keď je ukončené vykonávanie všetkých procesov.



• Jednoduchý paralelizmus možno využiť na ľubovoľnej úrovni dekompozície problému a pre ľubovoľný typ architektúry.

• Jednoduchý paralelizmus je hrubozrnný

• Jednoduchý paralelizmus možno niekedy s úspechom využiť pri prúdovom paralelizme pre zlepšenie vyváženosti pri prúdovom spracovaní.

• *Jednoduchá dekompozícia*

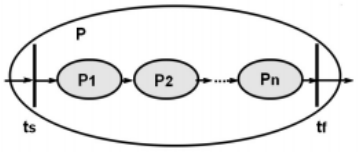
Prúdový paralelizmus

• *Problém P je rozdelený na množinu podproblémov, ktoré sú riešené postupne – v následnosti*

• Procesy počas vykonávania riešia rozdielne problémy P_i , pričom každý procesor spracováva údaje toho istého typu.

• Vo všeobecnosti je obtiažne dekomponovať problém do dostatočne dlhého prúdu podproblémov, čo je predpokladom pre využitie dostatočne veľkého počtu procesorov, pracujúcich paralelne.

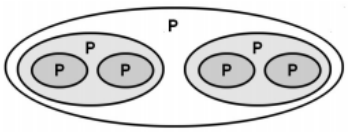
• Prúdové spracovanie je pre menší počet procesorov efektívne iba ak je nasýtené dostatočne veľkou množinou údajov. Ide tu o problém nábehu, ktorého časový interval musí byť čo najmenší v pomere k celkovému času prúdového spracovania.



• Prúdový paralelizmus je zväčša hrubozrnný, a preto sa architektúry MISD zriedkavo používajú pre riešenie problémov dekomponovaných prúdovým spôsobom.
• Prúdový paralelizmus je užitočný na vyšších úrovniach dekompozície problému, najmä pri dekompozícii problémov pre hrubozrnnejšie architektúry MIMD
• *Funkcionálna dekompozícia* - Dekompozícia problému obsahujúceho prúdový paralelizmus

Expanzívny paralelizmus

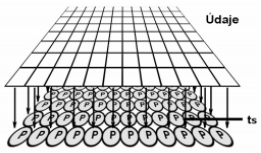
• *Problém P je riešiteľný riešením množiny n tých istých problémov rekurzívne. Ak n=2 potom sa volá rozdeľuj a panuj*



• Tento paralelizmus môže byť buď hrubozrnný alebo jemnozrnný
• Z funkčného hľadiska je dekompozícia problému triviálna a expanzívne paralelné problémy sú riešiteľné optimálnymi paralelnými algoritmami.
• Ak však veľkosť problému nie je známa, ľahko dochádza k preťaženiu paralelnej architektúry v dôsledku vyčerpania zdrojov. V tomto prípade expanzívny paralelizmus treba riadiť počas vykonávania, čo vedie k zvýšeným nákladom pri programovaní.
• Expanzívne paralelné problémy možno riešiť efektívne buď na architektúrach SIMD alebo MIMD, a to pomocou programového *modelu údajového paralelizmu* (v prípade architektúr SIMD) alebo pomocou programového *modelu odovzdávania správ* (pri architektúrach MIMD).
• *Hierarchická dekompozícia*

Masívny paralelizmus

• *Množina funkčne totožných problémov P a veľkej množiny údajov spracovávaných v tom istom čase*
• V typickom prípade ide o jemnozrnný paralelizmus, ktorý môže byť využitý najmä na architektúrach SIMD na základe programového modelu údajového paralelizmu. Možno ho však využiť aj v architektúrach MIMD trochu hrubozrnnejším spôsobom, s prídavnými nákladmi na programovanie.



• Výpočet je založený na nezávislých úlohách spracovávajúcich nezávislé mn. údajov.
• Pritom jednotlivé množiny údajov sa môžu aj navzájom prekrývať.
• *Údajová dekompozícia*

Využitie paralelizmu pre výkonné výpočty

• Aplikácie vysokého výkonu sú vyvíjané prioritným využitím ***masívneho*** a ***expanzívneho*** paralelizmu, ktoré sú inherentné problémom pravidelnej povahy. Tieto druhy paralelizmu môžu byť totiž nielen hrubozrnné, ale veľmi často sú jemnozrnné.
• ***Jednoduchý*** a ***prúdový*** paralelizmus je zvyčajne hrubozrnný. Tieto dva druhy paralelizmu sú inherentné problémom nepravidelnej povahy. Preto *vysoko výkonnú paralelnú aplikáciu možno ťažko realizovať využitím iba jednoduchého a prúdového paralelizmu*.

Programové modely a paralelné architektúry

K najčastejšie využívaným programovým modelom pre programovanie výkonných paralelných výpočtov patria:
• *Programový model údajového paralelizmu*. Je podporovaný predovšetkým architektúrami SIMD – vektorovými a maticovými procesormi, ale tiež výkonnými superpočítačmi, založenými na architektúre MIMD.
• *Programový model odovzdávania správ*. Je podporovaný architektúrami MIMD s distribuovanou pamäťou, ktoré sú realizované buď ako superpočítače alebo ako klastre počítačov v lokálnej sieti.

4. Definícia zrýchlenia a efektívnosti paralelného výpočtu a celkového zrýchlenia a efektívnosti. Odvodenie Amdahlovho pravidla pre neškálovateľné a škálovateľné problémy, porovnanie obidvoch prístupov. Meranie a výpočet podielu paralelného kódu.

Zrýchlenie a efektívnosť

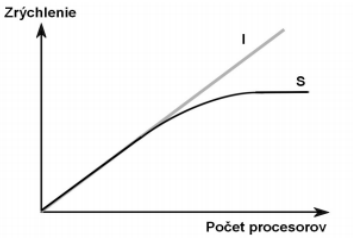
Nech T(n,1) je sekvenčný čas na 1 procese
T(n,p) je paralelný čas na p procesoch

Zrýchlenie $S(n,p)=\frac{T(n,1)}{T(n,p)}$ Efektívnosť $E(n,p)=\frac{S(n,p)}{p}$

Škálovateľnosť

• Programy sú škálovateľné, ak zrýchlenie je priamo úmerné vzrastajúcemu počtu procesorov.

• V ideálnom prípade očakávame S(n, p) = p a E(n, p) = 100 %. V praxi však platí S(n, p) < p, t.j. E(n, p) < 100 %. Preto ak je problém nedostatočne škálovateľný, zvýšenie počtu procesorov už nevedie k zvýšeniu zrýchlenia, viď obr., nanajvýš k zvýšeniu ceny paralelnej architektúry. Preto pri výbere paralelnej architektúry je potrebné zvážiť, pre aké druhy paralelných problémov bude určená, jej naddimenzovanie neznamená automaticky efektívne riešenie paralelných problémov v budúcnosti.



Celkové zrýchlenie a **celková efektívnosť**.

Nech $T_N(n)$ je čas vykonávania najrýchlejšieho známeho sekvenčného algoritmu na jednom procesore.

Potom numerická efektívnosť je definovaná podielom: $\frac{T_N(n)}{T(n,1)}$

Celkové zrýchlenie : $\overline{S}(n,p)=\frac{T_N(n)}{T(n,p)}$ Celková efektívnosť $\overline{E}(n,p)=\frac{\overline{S}(n,p)}{p}$

Numerická efektívnosť je mierou kvality sekvenčného algoritmu berúc do úvahy spracovávané údaje. Avšak vzhľadom na to, že v praxi $T_N(n)$ nemusí byť známy, často sa berie do úvahy dobrý sekvenčný algoritmus namiesto najlepšieho.

Amdahlovo pravidlo

Zrýchlenie dosiahnuteľné na paralelnom počítači môže byť značne ohraničené existenciou malej časti sekvenčného kódu, ktorý nemožno paralelizovať. Túto skutočnosť vyjadruje Amdahlovo pravidlo:

> Nech α je časť operácii počas výpočtu, ktoré musia byť vykonávané sekvenčne, taká, že platí $0 \le \alpha \le 1$. Potom, maximálne zrýchlenie dosiahnuteľné na paralelnom počítači s p procesormi je ohraničené vzťahom:

$$S(n,p)=\frac{1}{\alpha+(1-\alpha)/p}\le\frac{1}{\alpha}$$

Ak napríklad počas vykonávania 10% kódu musí byť vykonaného sekvenčne, potom maximálne zrýchlenie je 10, nezávisle na počte dostupných procesorov.

Odvodenie Amdahlovho pravidla (pre neškálovateľné problémy)

Predpokladajme, že všetky náklady sú nezávislé od počtu procesorov (čo nemusí byť vždy pravda). Potom celkový čas výpočtu *T(n, 1)* možno rozdeliť na paralelný a sekvenčný čas, podľa vzťahu

$$T(n,1) = T^P(n) + T^S(n)$$

Pri použití *p* procesorov dostávame paralelný čas v tvare

$$T(n,p) = T^P(n)/p + T^S(n)$$

Definujeme $\alpha = \frac{T^S(n)}{T(n,1)}$ a $S(n,p) = \frac{T(n,1)}{T(n,p)}$

Potom môžeme odvodiť:

$$\begin{aligned} S(n,p) &= \frac{T(n,1)}{T(n,p)} \\ &= \frac{T(n,1)}{T^S(n)+T^P(n)/p} \\ &= \frac{T(n,1)}{T^S(n)+(T(n,1)-T^S(n))/p} \\ &= \frac{1)}{T^S(n)/T(n,1)+(1-T^S(n)/T(n,1))/p} \\ &= \frac{1}{\alpha+(1-\alpha)/p} \end{aligned}$$

Odvodenie Amdahlovho pravidla (pre škálovateľné problémy)

$T(n,1) = T^s(n) + T^p(n).p$

$T(n,p) = T^s(n) + T^p(n)$

$$\alpha = \frac{Ts(n)}{T(n,p)}$$

$$S = \frac{T(n,1)}{T(n,p)} = \frac{Ts(n)+Tp(n).p}{T(n,p)} = \frac{Ts(n)+p(T(n,p)-Ts(n))}{T(n,p)} = \alpha + p.(1-\alpha)$$

Porovnanie obidvoch prístupov

U mnohých výpočtov sekvenčná časť *α = α(n)* klesá prudko k nule so vzrastom veľkosti problému. Preto v prípade, že problém je škálovateľný, α(n) závisí na počte procesorov a Amdahlovo pravidlo stráca značne svoj význam.

Paralelná časť kódu

Amdahlovo pravidlo je užitočné pri meraní paralelnej časti kódu 1−α pri použití p procesorov, podľa vzťahu:

$$1-\alpha = \frac{p}{p-1}.\frac{S(n,p)-1}{S(n,p)}$$

5. Spôsoby a prostriedky využitia masívneho paralelizmu v programovom modeli údajového paralelizmu. Hlavné metódy využitia expanzívneho paralelizmu v programovom modeli údajového paralelizmu. Odvodenie redukovaného počtu procesorov pri zachovaní optimálneho paralelného algoritmu.

Programový model údajového paralelizmu

- Pôvodom tohto programového modelu je vektorové programovanie, pri ktorom sú využívané vysoko optimalizované vektorové operácie.
- Paralelizmus možno riadiť pomocou konštrukcií paralelného jazyka, napr. pre paralelné vykonávanie cyklov.
- Programový model údajového paralelizmu je vhodný pre riešenie masívne paralelných problémov pravidelnej povahy, ktoré sa vyskytujú napr. pri spracovaní obrazov.
- Tento programový model sa stal známym predovšetkým v spojení s architektúrami SIMD, pretože problém spracovania rozsiahlej množiny údajov, ktorú možno rozdeliť na nezávislé množiny a spracovávať v malých krokoch jednoduchej funkcie, je problémom pravidelnej povahy.
- Masívne paralelné problémy pravidelnej povahy však možno riešiť aj na superpočítačoch typu MIMD (napr. SGI), pretože tieto dosiahli vysokú rýchlosť komunikácie a sú pre ne dostupné aj výkonné paralelné jazyky (napr. HPF - High Performance Fortran).

Masívny paralelizmus v modeli údajového paralelizmu

Využitie masívneho paralelizmu v programovom modeli údajového paralelizmu je založené na rozpoznaní nezávislých množín údajov v základnej množine spracovávaných údajov a na paralelizácii cyklov.

Nech M a N sú množiny množín M_i and N_i definované takto:

$$M = M_1 \cup M_2 \cup \ldots \cup M_n \qquad N = N_1 \cup N_2 \cup \ldots \cup N_n$$

a f_i, pre $i = 1 \ldots n$ sú funkcie (algoritmy), ktorými sa počíta množina M

$$M = \{f_i\ (N_i) \mid i = 1 \ldots n\}$$

t.j. $M_i = f_i\ (N_i)$ pre $i = 1 \ldots n$.

Napríklad porovnajme teraz vzorový sekvenčný a paralelný algoritmus pre *nezávislé mn. údajov*:

(sekvenčný – zložitosť O(n))

```
CONST n = 1000;
VAR M : ARRAY[1..n] OF typ1;
    N : ARRAY[1..n] OF typ2;
    i : INTEGER;
BEGIN
  FOR i := 1 TO n DO
    M[i] := f_i (N[i])
  END FOR
END
```

(paralelný – zložitosť O(1))

```
CONST n = 1000;
VAR M : ARRAY[1..n] OF typ1;
    N : ARRAY[1..n] OF typ2;
    i : INTEGER;
BEGIN
  FORALL i = [1 .. n] IN PARALLEL DO
    M[i] := f_i (N[i])
  END FORALL
END
```

Ďalší spôsob využitia je *paralelné násobenie matíc*.

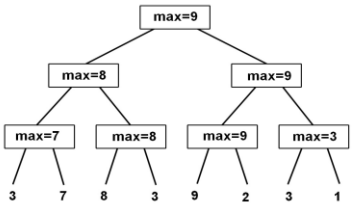
Expanzívny paralelizmus v modeli údajového paralelizmu

Využitie expanzívneho paralelizmu v modeli údajového paralelizmu je založené na vhodnom spôsobe zobrazenia spracovávaných údajov do pamäti a v nahradení rekurzie iteráciou.

Metóda rozdeľuj a panuj

Pre ilustráciu predpokladajme problém nájdenia maximálnej hodnoty z množiny {3, 7, 8, 3, 9, 2, 3, 1} metódou Rozdeľuj a panuj, podľa obr.

Napriek teoretickej zložitosti O(log n) tohto algoritmu, takáto priamočiara aplikácia metódy rozdeľuj a panuj vedie

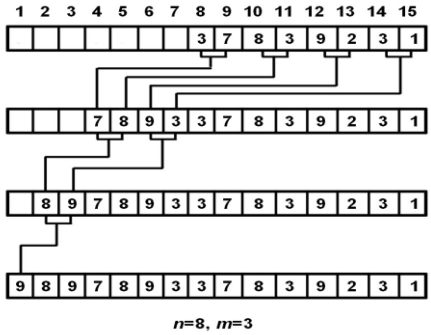


- k nízkej záťaži (slabému využitiu) paralelného počítača v prvých krokoch výpočtu.
- Vzniká nebezpečenstvo preťaženia (a prípadného zablokovania) v nasledujúcich krokoch (samozrejme, pre rozsiahlejšiu množinu vstupných údajov)

Preto je potrebné v prípade expanzívneho paralelizmu uplatniť špecifické metódy, napr. metódu vyváženého stromu

Metóda vyváženého stromu

- Metódu vyváženého stromu možno uplatniť namiesto metódy rozdeľuj a panuj vtedy, ak veľkosť problému n, $n = 2^m$ je známa.
- Vstupné údaje je potrebné umiestniť do poľa veľkosti 2n − 1, na pozície n,(n + 1),. . . ,(2n − 1).



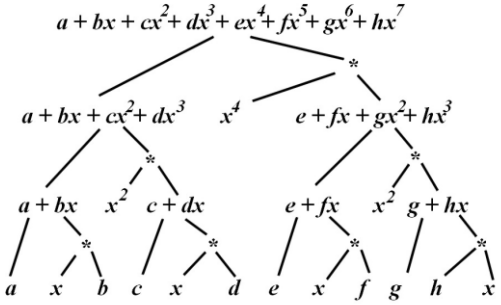
Metóda binárneho stromu

- Je ďalšou z mnohých metód, ktoré sú používané pri využití expanzívneho paralelizmu v programovom modeli údajového paralelizmu. Táto metóda na rozdiel od predošlej nevyžaduje, aby bol strom výpočtu vyvážený.
- Pre ilustráciu predpokladajme, že je potrebné vypočítať polynóm p(x) stupňa n v bode x = x_0.

Nech n = 2^k − 1 pre celočíselnú konštantu k. Polynóm vyjadríme v tvare

$$p(x) = r(x) + x^{(n+1)/2} q(x)$$

kde q(x) a r(x) sú polynómy stupňa 2^{k-1} − 1, ktoré možno počítať paralelne podľa obr.



Odvodenie redukovaného počtu procesorov (patrí to ku Expanzívnemu paralelizmu v modeli údajového paralelizmu)

- Ako uvidíme z nasledujúceho odvodenia, je možné zredukovať počet procesorov na hodnotu $p = n/\log n$:

1. Predpokladajme $p < n/2$ procesorov a rozdeľme n prvkov na p skupín. Nech $(p - 1)$ skupín obsahuje $[n/p]$ zvyšná skupina $n-(p-1)[n/p]$ $(\leq n/p)$ prvkov.
2. Priraďme procesor každej skupine, celkove máme p procesorov. Každý z nich hľadá maximum sekvenčne v rámci skupiny a paralelne s inými skupinami, v čase $[n/p]-1+\log p$
3. Substitúciou $p = n/\log n$ dostaneme $[n/(n/\log n)]-1+\log(n/\log n)$, t.j. zložitosť $O(\log n)$. To znamená, že takáto substitúcia zachováva optimálnosť paralelného algoritmu.
4. Preto $p=n/\log n$ je redukovaný počet procesorov (za predpokladu, že veľkosť problému je známa).

6. Charakteristika programového modelu odovzdávania správ, definícia a dôsledok latentnosti. Charakteristika modelu SPMD a MPMD. Základné druhy operácii pre odovzdávanie správ medzi dvoma procesmi v MPI a ich sémantika z hľadiska volania, návratu a ukončenia procedúry MPI.

Podstata modelu odovzdávania správ je takáto:
• Programátor pri riešení paralelného problému definuje *sekvenčné procesy, ktoré môžu navzájom komunikovať formou odovzdávania správ*. Za synchronizáciu je zodpovedný programátor.
• Namiesto paralelného jazyka *sa používa sekvenčný jazyk* a operácie pre odovzdávanie správ sa aktivujú volaním interfejsu pre odovzdávanie správ, ktorý zabezpečuje prenos správ cez fyzickú komunikačnú sieť, ktorou sú paralelné procesory navzájom prepojené.
• Najznámejším programovým modelom pre odovzdávanie správ je model založený na jedinom programe a viacerých množinách údajov (*SPMD* – Single Program Multiple Data) ktorý je aj prakticky dobre zvládnutý, na rozdiel od modelu založenom na rôznych procesoch dynamicky vytváraných a vykonávaných na rôznych procesoroch, známy ako model *MPMD* – Multiple Program Multiple Data.
Pomocou modelu odovzdávania správ možno riešiť rôznorodejšie paralelné problémy z týchto dôvodov:
• Programátor nie je obmedzovaný dekompozíciou problému nepravidelnej povahy, ktorá vedie k problému pravidelnej povahy
• Odovzdávanie správ umožňuje využiť väčší rozsah zrnitosti paralelizmu:
- Hrubozrnnejšie paralelné problémy možno riešiť pomocou počítačových klastrov s prepojením zbernicou,
- Jemnozrnnejšie problémy zasa pomocou klastrov počítačov prepojených rýchlou prepojovacou sieťou alebo pomocou superpočítačov, najčastejšie v podobe jednoskriňových počítačov, založených na vzájomnej komunikácii veľkého množstva procesorov, ktorá je realizovaná technológiou koordinačných procesorov.

• Existuje všeobecne akceptovaný *štandard MPI (Message Passing Inteface)*, ktorý je implementovaný na veľkom pocte rôznych superpočítačov a počítačových klastrov.

Charakteristika modelu SPMD
• Tá istá časť programu alebo rôzne časti programu môžu byt vykonávané v tom istom čase rôznymi procesmi.
• Každý proces má priradenú svoju lokálnu pamäť.
• Komunikácia je realizovaná volaniami špeciálnych procedúr pre odovzdávanie správ.

Charakteristika modelu MPMD
• Chovanie je záležitosťou súčasného výskumu.
• Model založený na rôznych procesoch dynamicky vytváraných a vykonávaných na rôznych procesoroch
• Na každú aplikáciu v modeli MPMD možno použiť taktiež model SPMD.

Operácie a procedúry pre odovzdávanie správ
• V štandarde MPI každá operácia pre odovzdávanie správ sa aktivuje pomocou volania knižničnej procedúry pre odovzdanie správy z procesu. Z hľadiska lokálnosti účinku volania procedúr MPI, ktoré ich aktivujú operácie MPI, rozoznávame operácie MPI:
- •Lokálne - Ich ukončenie (návrat z procedúry MPI) závisí iba na procese, ktorý ich volá. Tieto operácie nevyžadujú komunikáciu s inými procesmi.
- •Nelokálne - ich ukončenie môže závisieť na vykonaní nejakej procedúry MPI volanej iným procesom. Takáto operácia môže vyžadovať komunikáciu s iným procesom definovaným používateľom.

• Z hľadiska programátora sú dôležité tri časové okamihy:
1) *čas volania procedúry MPI* – ak zanedbáme latentnosť, potom = čas začiatku operácie

2) *čas návratu*: K návratu z volanej procedúry môže dôjsť aj skôr, než bola operácia ukončená
3) *čas vykonania aktivovanej operácie* (napr. odovzdania správy = čas ukončenia operácie odoslania správy): ukončenie operácie znamená iba to, že odosielanie správy sa dostalo do stavu, v ktorom proces, ktorý správu odoslal, môže opätovne a bez akéhokoľvek rizika použiť svoje zdroje, čo je podstatná informácia pre programátora.

Komunikáciu medzi dvoma procesmi:
Táto komunikácia sa uskutočňuje medzi odosielajúcim a prijímajúcim procesom a môže byt:
• blokujúca:
- Ak návrat z procedúry MPI indikuje, že po príchode z volanej funkcie môžeme všetky zdroje používať
• neblokujúca:
- Ak k návratu z procedúry MPI môže dôjsť predtým, než operácia MPI bola ukončená, a preto používateľ nemôže automaticky použiť zdroje špecifikované vo volaní.

Komunikácia medzi dvoma procesmi
• Predpokladajme, že na strane odosielajúceho procesu sú údaje pripravené v bafri na odoslanie prijímaciemu procesu a na strane prijímacieho procesu existuje buffer pre prijatie týchto údajov.
• Ďalej, nech každý z n procesov má svoje poradové číslo (rank) r z rozsahu 0 . . . n − 1.
• Hovoríme, že odoslanie správy procesom r_1 procesu s poradovým číslom r_2 zodpovedá prijatiu správy procesom r_2 odoslanej procesom r_1.
• Vzhľadom na to, že poradové číslo zdrojového procesu r_1 a cieľového procesu r_2 sú parametrami procedúry MPI pre odoslanie správy volanej procesom r_1, v dôsledku čoho sa tieto poradové čísla stanú súčasťou obálky správy, správu prijme proces, ktorý volá procedúru MPI pre prijatie správy s tými istými parametrami označujúcimi zdrojový proces r_1 a cieľový (prijímajúci) proces r_2. Preto môžeme hovoriť aj o zodpovedajúcich volaniach procedúr MPI pre odoslanie a prijatie správy.
• Zjednodušene povedané, v tomto odseku ide o to, že k MPI_Send existuje analogické MPI_Recv.

Komunikácia medzi dvoma procesmi
• Pri komunikácii medzi dvoma procesmi existujú štyri režimy odosielania správ:
1. Štandardný
2. Bafrovaný
3. Synchronizovaný (rendezvous)
4. Režim pripravenosti
• Režim prijímania správ je však iba jeden.
• Vo všetkých prípadoch môže ísť o komunikáciu blokujúcu alebo neblokujúcu

7. Blokujúce a neblokujúce operácie MPI, ich porovnanie z hľadiska využitia zdrojov a zablokovania výpočtu. Vzťah k asynchrónnemu a synchrónnemu odovzdávaniu správ - rendezvous. Poradie správ a vzťah k deterministickému výpočtu.

Typy komunikácie: blokujúca, neblokujúca (definícia v 6.)

Blokujúca:

• Procedúry MPI pre odoslanie správ v *štandardnom*, *bafrovanom* a *synchrónnom* režime nevyžadujú, aby bolo predtým doručené zodpovedajúce prijatie správy, t.j. aby sa predtým uskutočnilo volanie procedúry MPI pre prijatie správy.
• Ak však je procedúra MPI pre odovzdávanie správ volaná v *režime pripravenosti*, a nedošlo predtým k volaniu zodpovedajúcej procedúry pre prijatie správy, spôsobí to chybu a výsledok nie je nedefinovaný.

• Okrem bafrovaného odoslania správ, ktoré je **lokálne** (k návratu dochádza bezprostredne po volaní príslušnej procedúry MPI), odosielanie správ v ostatných režimoch **nie je lokálne**.
• Operácia odovzdania správy v
- *synchrónnom režime* je ukončená vtedy, ak zodpovedajúca operácia prijímania správy prijala údaje z bafra odosielajúceho procesu, ktorý ho následne môže opätovne použiť.
- *bafrovanom režime* končí bezprostredne po volaní procedúry MPI
- *štandardnom režime* končí dvojako: tak ako v synch. alebo v bafrovanom režime
- *režime pripravenosti* končí dvojako: tak ako v synch. alebo v bafrovanom režime

• Sémantika synchrónneho odovzdávania správ, známeho pod pojmom rendezvous (čítaj randevú) je daná synchrónnym režimom blokujúceho odovzdávania správy na strane zdrojového procesu a blokujúcim prijímaním správy na strane cieľového procesu.

Štandardný SEND `MPI_Send()`
• *k návratu dochádza vtedy, keď je bafer možné opätovne použiť*:
a) buď po vykonaní receive u príjemcu (ak je správa dlhšia) – analógia synchrónneho
b) buď po skopírovaní dát do interného bafra (aj je správa kratšia) – an. bafrovaného
• čiže môže ale nemusí byť bafrovaný (závisí od implementácie a dĺžky správ)
Bafrovaný SEND `MPI_Bsend()`
• *k návratu dochádza ihneď – hneď je možné opätovne využiť bafer*
• pripojenie a odpojenie buffra:
1. alokácia miesta v jazyku C
2. alokácia `int MPI_Buffer_attach(void * buffer, int size)`
3. buffer môžeme použiť rôzne
4. `int MPI_Buffer_detach(void * buffer, int *size))`

Synchrónny SEND `Mpi_Ssend()`
• *neskončí sa , pokiaľ príjemca nezavolá receive a kým buffer nemôže byť použitý*
• ukončenie implikuje že receive prebehol
• externá rutina na oneskorenie
Režim pripravenosti SEND `MPI_Rsend()`
• blokujúci režim pripravenosti, prijímateľ už musel volať receive

RECEIVE `MPI_Recv()`
• vždy blokujúci
• ak sa špecifikuje nie konkrétny proces, ale `MPI_ANY_SOURCE` a tiež namiesto konkrétnej značky sa použije `MPI_ANY_TAG`, potom detailne informácie vieme získať zo statusu: `status.MPI_SOURCE`, `status.MPI_TAG`, `status.MPI_ERROR`

Neblokujúca

• Cieľom neblokujúcej komunikácie je zvýšenie výkonnosti prostredníctvom prekrytia výpočtu a komunikácie.
• K návratu pri neblokujúcich procedúrach pre odosielanie správ vo všetkých štyroch režimoch dochádza bezprostredne po ich volaní, teda predtým, ako je správa úplne prenesená z bafra odosielajúceho procesu. To isté platí aj pre neblokujúce prijatie správ.
• Neblokujúce odosielanie správ možno kombinovať s blokujúcim prijímaním správ a naopak.
• Pri použití neblokujúcich operácii je veľmi dôležité, aby medzi volaním procedúry MPI pre odoslanie a prijatie správy a následným testom na ukončenie operácie *bolo možné vykonať užitočný výpočet*

Štandardný SEND `MPI_Isend()`
• môžeme testovať v akom je stave `MPI_Test`
• `MPI_Wait` čaká, kým nedôjde k odozve
• req – vnútorná štruktúra nás nezaujíma
Bafrovaný SEND `MPI_Ibsend()`
• bafrovaný, neblokujúci
Synchrónny SEND `Mpi_Issend()`
• synchrónny, neblokujúci - Wait/Test bude úspešné iba ak prijímateľ vykonal receive
Režim pripravenosti SEND `MPI_Irsend()`
• to isté ako pri `MPI_Rsend`, avšak neblokujúce
RECEIVE `MPI_Irecv()`

Posielanie správ
- Správy sa nepredbiehajú
- Procesy prijímajú správy v takom poradí v akom boli odoslané
- Podmienka je, že procesy sú jednovláknové a proces nepoužije procedúru na prijatie akejkoľvek správy
- Ak sú procesy viacvláknové, nie je možne určiť relatívne usporiadanie operácií v čase v rôznych vláknach

8. Princíp skupinovej komunikácie v MPI, definícia hlavného procesu, skupinové operácie v MPI, ich syntax, sémantika a aplikácia.

Princíp skupinovej komunikácie v MPI

• Všetky procesy v skupine potrebujú zavolať funkciu pre odovzdávanie správ. Niektorý proces v skupine môže byť root ak je to potrebné. Nie všetky skup. kom. musia mať určený root proces.
• Na jednej strane, počet procesov komunikujúcich navzájom skupinovo nie je ohraničený, na druhej strane, skupinová komunikácia synchronizuje výpočet, keďže je ukončená vtedy, keď ukončí komunikáciu každý proces. Preto je skup. komunikácia vhodná pri problémoch pravidelnej povahy.
• Ďalšie obmedzenie skupinovej komunikácie spočíva v tom, že ju možno použiť iba v rámci jedného z dvoch typov komunikátorov, ktoré sa nazývajú intrakomunikátory napr. MPI_COMM_WORLD.

Skupinové operácie v MPI, ich syntax, sémantika a aplikácia

Broadcast

• Root zapíše data do buffra a všetci ostatný čítajú čo zapísal root.
• `int MPI_Bcast(void *buf, int count,MPI_Datatype datatype, int root, MPI_Comm comm)`

Reduce

• Rootovský proces vykoná danú operáciu (op) nad dátami od ostatných procesov.
• Obsahy bafrov odosielaných správ musia byť toho istého typu a sú operandami výrazu, ktorého hodnota bude prijatá hlavným procesom ako výsledok výpočtu výrazu.
• `int MPI_Reduce (void *sendbuf, void *recvbuf,int count, MPI_Datatype datatype,MPI_Op op, int root, MPI_Comm comm)`

Scatter

• Hlavný proces pošle množinu dát, ktoré si ostatné procesy rozdelia na pravidelné časti, resp. rozsypanie správy v tvare D_0, D_1, . . . , $D_{(n-1)}$, ktorú odošle hlavný proces, a to takým spôsobom, že proces s poradovým číslom k prijme položku D_k , pre k = 0 . . . n − 1.
• `int MPI_Scatter (void *sendbuf, int sendcount,MPI_Datatype sendtype, void *recvbuf,int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`

Gather

• Hlavný proces proces poskladá vo svojom buffery všetky podčasti ktoré mu pošlu ostatné procesy.
• Inverzná operácia k operácii `MPI_Scatter`. Proces k odosiela správu D_k, pre k = 0 . . . n − 1, a správu v tvare D_0,D_1, . . . ,$D_{(n-1)}$ prijme hlavný proces.
• `int MPI_Gather (void *sendbuf, int sendcount,MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`

AllGather

• Na rozdiel od `MPI_Gather` výslednú správu prijme každý proces
• `MPI_Allgather(void* send_data, int send_count, MPI_Datatype send_datatype, void* recv_data, int recv_count, MPI_Datatype recv_datatype, MPI_Comm communicator)`

Barrier

• Synchronizacná bariéra, všetky procesy po volaní procedúry budú zosynchronizované, t.j. budú po ukončení operácie synchronizácie pokračovať vo vykonávaní v tom istom čase.
• `int MPI_Barrier( MPI_Comm comm )`

Alltoall

• Odoslanie správy všetkým procesom od všetkých procesov.
• `int MPI_Alltoall( void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`

9. Základné typy a odvodené typy MPI a spôsob definície nového typu. Zobrazenie typu, posunutie, rozsah, spodná a horná hranica nového typu. Definícia, porovnanie a využitie odvodených typov. Stláčanie a porovnanie komunikácie na základe odvodeného typu oproti komunikácii prostredníctvom stláčania.

Základné typy:

Typ MPI	Typ v jazyku C
MPI_CHAR	signed char
MPI_SHORT	signed short
MPI_INT	signed int
MPI_LONG	signed long
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	byte
MPI_PACKED	

Ako je možné odosielať a prijímať dáta zložitejšie ako základné typy? :

a) použitie viacerých MPI volaní pre odoslanie a prijatie každého dátového elementu
- neefektívne, pomalé, zdĺhavé, nemotorné

b) použiť typ MPI_PACKED - zbalenie správy pred jej odoslaním a rozbalenie správy po jej prijatí
- výhodou je, že nie je potrebné vytvoriť nový typ
- nevýhody: ne-prenositeľnosť, prenos správ zaťažený zbaľovaním a rozbaľovaním je pomalší ako priamy prenos

c) použiť MPI_BYTE – obídenie dátových typov, podobne ako pri MPI_PACKED nevýhodou je hlavne absencia prenositeľnosti v heterogénnych systémoch

d) **definovať odvodené typy** – prenositeľné, efektívnejšie a elegantnejšie riešenie pre prenos nespojitých alebo zmiešaných (rôznych) údajov**.**

Zobrazenie typu

Každý typ MPI napodobňuje typ jazyka jednotným spôsobom založeným na nasledujúcej definícii Typemap – zobrazenia typu:

$$Typemap = \{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\}$$

kde $disp_i$ je relatívny posun položky typu $type_i$. Napr. pre MPI_INT je zobrazenie: *Typemap = {(int, 0)}*

Horná a Spodná hranica typu

Nech $Typemap = \{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\}$ je zobrazenie typu

Potom spodná hranica typu **lb** a horná hranica typu **ub** je definovaná takto:

$$\mathbf{lb}(Typemap) = min\ \{disp_j\},$$ ak žiadny typ nie je typom `MPI_LB`

$$\mathbf{ub}(Typemap) = max\ \{(disp_j + sizeof(type_j))\} + \varepsilon$$, ak žiadny typ nie je `MPI_UB`

kde `MPI_LB` a `MPI_UB` sú špeciálne typy nulového rozsahu (veľkosti), ktoré možno použiť pri definícii nového typu.

Rozsah nového typu

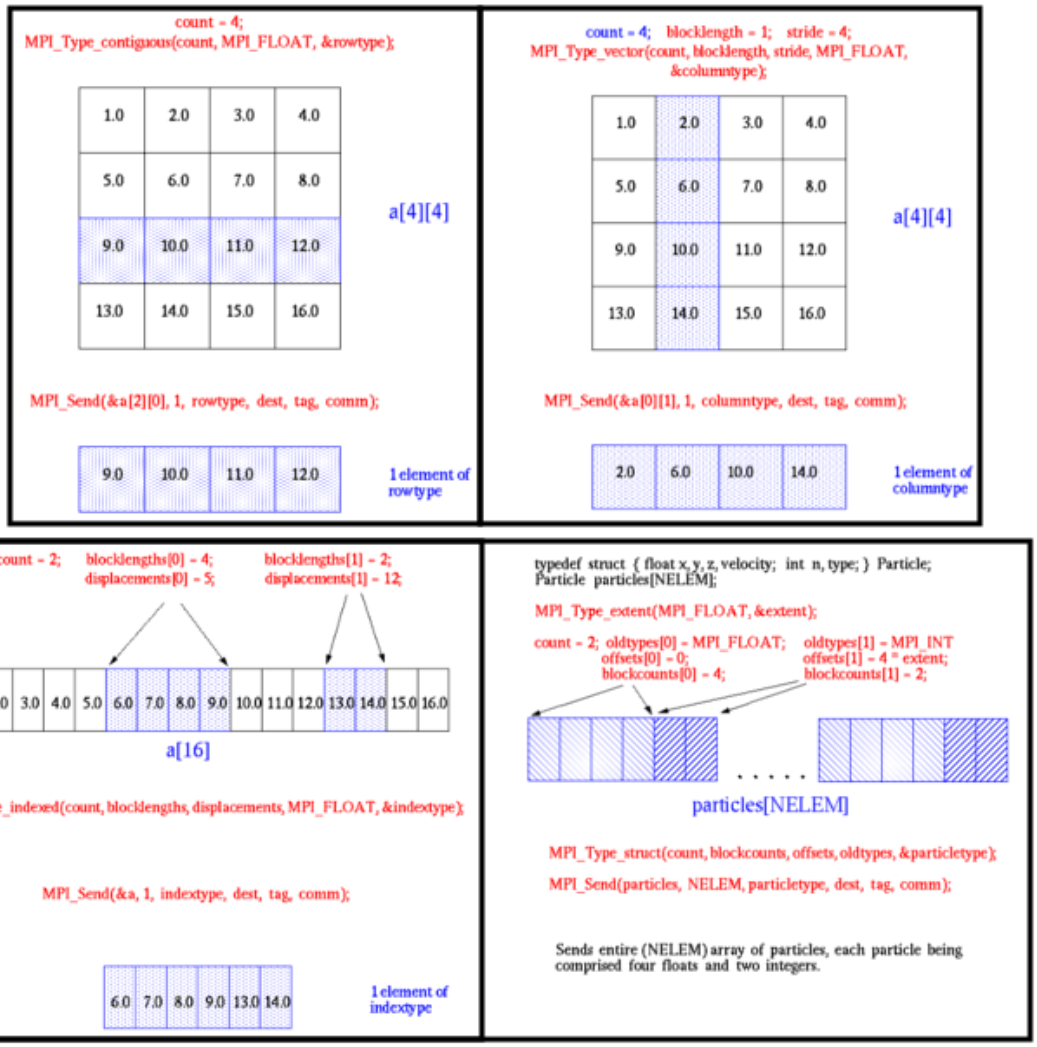
$extent(Typemap) = \mathbf{ub}(Typemap) - \mathbf{lb}(Typemap)$

Definícia nového typu MPI:

• Nové typy MPI možno definovať ich konštruovaním zo základných typov MPI alebo predtým definovaných nových typov.

• Nové typy MPI možno kategorizovať podľa počtu polí, ktorých dĺžky treba určiť, podľa počtu relatívnych posunov a podľa počtu rôznych typov a to takto:

- **contignous (*spojitý typ*)** – jedna hodnota dĺžky poľa, žiaden posun, jeden údajový typ (z ktorého je koštruovaný)
- **strided vector (*vektor s obkrokmi*)** – jedna hodnota dĺžky poľa, jeden posun, jeden údajový typ
- **indexed (*indexový typ*)** - viac dĺžok poľa, viac posunov, jeden údajový typ
- **structure (*štruktúra*)** - viac dĺžok poľa, viac posunov, viac údajových typov



Základný postup pri definícii nového typu je takýto:

1. Definujeme meno pre nový údajový typ, napr. `MPI_Datatype datatype;`
2. Treba vypočítať hodnoty argumentov pre procedúru MPI, ktorá vytvorí nový typ. Takáto procedúra sa nazýva tiež konštruktorom údajového typu MPI.
3. Potom možno použiť konštruktor pre definíciu – konštrukciu nového typu MPI.
4. Poslednou akciou je zaznamenanie nového typu volaním `MPI_Type_commit(&datatype)`.

Nový typ datatype možno používať dovtedy, kým sa tento typ neuvoľní pomocou procedúry
`int MPI_Type_free(MPI_Datatype *datatype)`

10. Dôvody a spôsoby konštrukcie nových komunikátorov v MPI, rozdiel medzi skupinami procesov a komunikátormi. Rozdiel medzi intrakomunikátormi a interkomunikátormi. Druhy topológií procesov a ich uplatnenie v komunikátoroch.

Komunikátory a topológia procesov

• Komunikátor je skupina vzájomne komunikujúcich sekvenčných procesov s kontextom. Takýto komunikátor sa nazýva ***intrakomunikátor***. Kontext, ktorý si možno pre jednoduchosť predstaviť ako farbu skupiny procesov, umožňuje odlíšiť dve identické skupiny procesov patriace rôznym intrakomunikátorom.

• Intrakomunikátor umožňuje okrem komunikácie medzi dvoma procesmi aj skupinovú komunikáciu a okrem kontextu môže mat definovanú aj topológiu procesov, ktoré obsahuje, t.j. usporiadanie procesov v priestore.

• Iným druhom komunikátorov sú ***interkomunikátory***, slúžiace na komunikáciu medzi intrakomunikátormi. Interkomunikátory však nemôžu mat definovanú topológiu procesov a tiež neumožňujú skupinovú komunikáciu.

• Koncepcia komunikátorov a topológii, tak ako je definovaná v štandarde MPI, súvisí s potrebou disciplinovanej organizácie paralelného výpočtu v oddelených skupinách, pričom je zaručená nezávislosť týchto skupín, a efektívneho vykonávania paralelného programu, vyplývajúceho z možnosti prispôsobenia topológie procesov topológii procesorov.

Základom pre túto organizáciu sú:

• *dva intrakomunikátory*, ktoré existujú, t.j. sú dopredu definované pre množinu procesov, určenej pri štarte programu (príkazom mpirun), a to:

`MPI COMM WORLD` komunikátor obsahujúci všetky procesy

`MPI COMM SELF` komunikátor obsahujúci jediný proces, a to ten, ktorý ho práve používa.

• *Označenie n procesov v skupine* vnútornými poradovými číslami v rozsahu 0, . . . , n − 1 umožňuje aplikáciu rôznych množinových operácií, na základe ktorých možno vytvárať nové komunikátory, rozdeľovať komunikátor, apod.

Intrakomunikátory a skupiny procesov

• Intrakomunikátory je možné vytvárať:

1. vytvorením duplikátu existujúceho komunikátora pomocou procedúry `MPI_Comm_dup`.
2. rozdelením komunikátora na viacero komunikátorov pomocou procedúry `MPI_Comm_split`

• Okrem uvedených dvoch možností vytvárania komunikátorov priamo z existujúceho komunikátora, je možné extrahovať z existujúceho komunikátora skupinu do premennej špeciálneho typu `MPI_Group` – dochádza tak k *oslobodeniu skupiny procesov komunikátora od kontextu*

• Na základe *množinových operácií* (napr. `MPI_Group_union`, `MPI_Group_intersection`), možno *vytvoriť novú skupinu procesov* a na základe takto vytvorenej novej skupiny procesov konštruovať nový komunikátor.

Interkomunikátory

• Procesy, ktoré patria do rôznych intrakomunikátorov, nemôžu medzi sebou komunikovať automaticky, ale iba vtedy, ak patria do toho istého interkomunikátora.

• To znamená, že interkomunikátor, za predpokladu, že bol vytvorený, umožňuje komunikáciu medzi dvoma procesmi patriacich rôznym intrakomunikátorom. K základným operáciám v súvislosti s interkomunikátormi patria:

`MPI_Intercomm_create` vytvorenie interkomunikátora

`MPI_Intercomm_merge` vytvorenie intrakomunikátora z interkomunikátora

`MPI_Comm_remote_size` zistenie počtu procesov vo vzdialenej skupine (intrakomunikátore) interkomunikátora

`MPI_Comm_test_inter` zistenie, či komunikátor je intrakomunikátorom alebo interkomunikátorom

Topológie procesov

• Pre každý intrakomunikátor je možné definovať jednu z dvoch nasledujúcich topológií procesov:

- *mriežkovú topológiu*, pri ktorej budú procesy intrakomunikátora priradené vrcholom pravidelnej n rozmernej mriežky, alebo
- *grafovú topológiu*, pri ktorej budú procesy intrakomunikátora priradené vrcholom grafu. Táto topológia je všeobecnejšia ako mriežková.

• **Cieľ topológie procesov** – organizácia vzájomnej komunikácie v priestore procesov tak, aby čo najlepšie vystihovala jednak implementované algoritmy, a jednak použitú architektúru