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.

Výkonné paralelné výpočty sa uplatňujú predovšetkým pri modelovaní, simulácii a analýze zložitých technických, fyzikálnych a biologických systémov, s cieľom skúmania ich vlastností a predpovedania ich správania. Riešenie problémov na báze výkonných paralelných výpočtov vyžaduje interdisciplinárny prístup na jednej strane a silnú špecializáciu na druhej strane.

Okrem samozrejmej znalosti kódovania v programovacom jazyku a znalosti príslušných algoritmov, je pri programovaní výkonných paralelných výpočtov mimoriadne dôležité zvoliť pre riešenie vhodnú počítačovú architektúru, prispôsobiť tejto architektúre paralelný výpočet, vedieť dopredu odhadnúť čas vykonávania výpočtu a poznať metódy a prostriedky, ktorými možno dosiahnuť efektívne riešenie.

Metodológia paralelného programovania sa uplatňuje v dvoch smeroch:

- Pri využití paralelných algoritmov zameraných na vysoko výkonné paralelné výpočty.
- Pri programovaní paralelných alebo pseudoparalelných systémov, známych pod pojmom systémy reálneho času.

Cieľom programovania systémov reálneho času je

- uspokojenie požiadaviek na čas vykonávania systému
- zabezpečenie spoľahlivosti systému z hľadiska správnosti jeho funkcie

Pri programovaní výkonných paralelných výpočtov je základným cieľom dosiahnutie tých istých výsledkov ako na sekvenčnom počítači, avšak pomocou viacprocesorovej architektúry. Nie je pritom potrebné uvažovať o žiadnom dostatočnom čase vykonávania, pretože cieľom je dosiahnuť vykonávanie v čo najkratšom čase.

Cieľom výkonných paralelných výpočtov je:

- podstatné zníženie času výpočtu (oproti času vykonávania toho istého výpočtu na jednoprocesorovom počítači)
- spracovanie veľkého množstva údajov.

Pseudoparalelizmus na jednoprocesorovom systéme je pre výkonné paralelné výpočty úplne nevyužiteľný. Preto v súvislosti s výkonnými paralelnými výpočtami ide vždy o paralelné programovanie určené pre paralelné počítačové 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 pred samotnou paralelnou implementáciou dekomponovať.

Cieľom správnej dekompozície problému je:

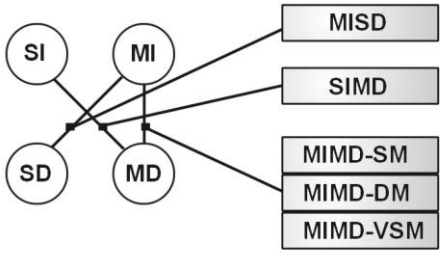
- zabezpečenie dostatku zdrojov paralelného počítačového systému počas výpočtu.
- naplánovanie optimálnej záťaže každého procesora a minimalizácia nákladov, menovite:
  1. času komunikácie
  2. časti kódu, ktoré sú vykonávateľné iba sekvenčné

Jedným z kľúčových problémov, ktoré je potrebné v súvislosti s paralelnými výpočtami riešiť, je vyvažovanie paralelného výpočtu. Program, ktorý realizuje paralelný výpočet, je vyvážený vtedy, ak vykonávanie je distribuované medzi jednotlivé procesory čo najefektívnejšie a každý procesor pracuje väčšinu času na výpočte a

minimum času venuje komunikácii s inými procesormi. Cieľom správnej dekompozície je teda vývoj vyvážených programov staticky, t.j. pri návrhu. Dynamické vyvažovanie znamená vyvažovanie počas vykonávania. Tento spôsob vyvažovania využíva prídavné algoritmy, ktoré spôsobujú zvýšené náklady pri vykonávaní.

Paralelné architektúry

Flynnova klasifikácia paralelných počítačových architektúr je založená na troch kombináciách jediného prúdu inštrukcií (Single Instruction– SI), resp. viacerých prúdov inštrukcií (Multiple Instruction – MI) a jediného prúdu údajov (Single Data – SD), resp. viacerých prúdov údajov (Multiple Data – MD). Takto by sme dostali štyri základné druhy architektúr: SISD, SIMD, MISD a MIMD.



Obrázok 1 Flynnova klasifikácia paralelných počítačových architektúr

Pretože pri architektúre 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.

Na druhej strane, architektúra MIMD umožňuje súčasné vykonanie viacerých inštrukcií (patriacich rôznym prúdom), z ktorých každá spracováva údaj patriaci inému prúdu údajov.

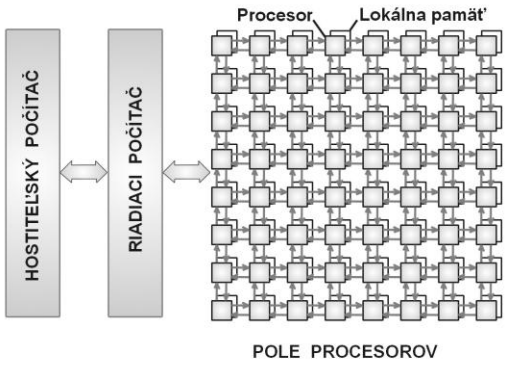
Okrem architektúry SISD teda všetky ostatné umožňujú využiť paralelizmus problému. Rozšírenie základnej Flynovej klasifikácie spočíva v ďalšom rozčlenení architektúr MIMD na:

- architektúry s distribuovanou pamäťou (Distributed Memory – DM)
- architektúry s virtuálnou spoločnou pamäťou (Virtual Shared Memory – VSM)
- architektúry so spoločnou pamäťou (Shared Memory – SM)

Architektúra SIMD

Počítačová architektúra SIMD je 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 (posunmi a inými operáciami charakteristickými pre polia) v uzavretom kroku, resp. inštrukciami vydanými mriežke procesorov riadiacim procesorom.



Obrázok 2 Štruktúra architektúry SIMD

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.

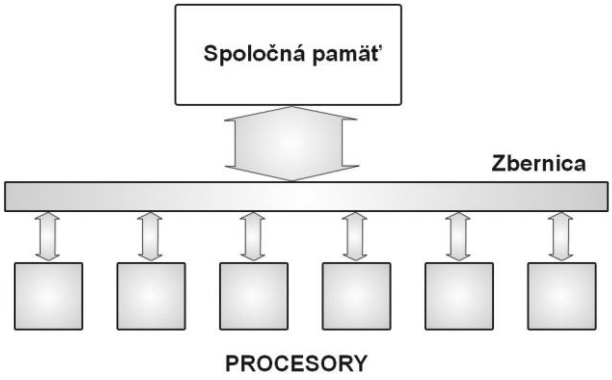
Príkladmi architektúr SIMD sú ICL Distributed Array Processor, Convex, alebo Thinking Machine Corporation's CM-200.

Architektúra 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.

SM – Architektúra MIMD so spoločnou pamäťou

Architektúra MIMD so spoločnou pamäťou SM obsahuje menší počet procesorov, z ktorých každý má prístup do globálnej pamäti prostredníctvom zbernice alebo iného druhu prepojenia.



Obrázok 3 Štruktúra architektúry SM - MIMD so spoločnou pamäťou

Hlavnou výhodou tejto architektúry je to, že 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.

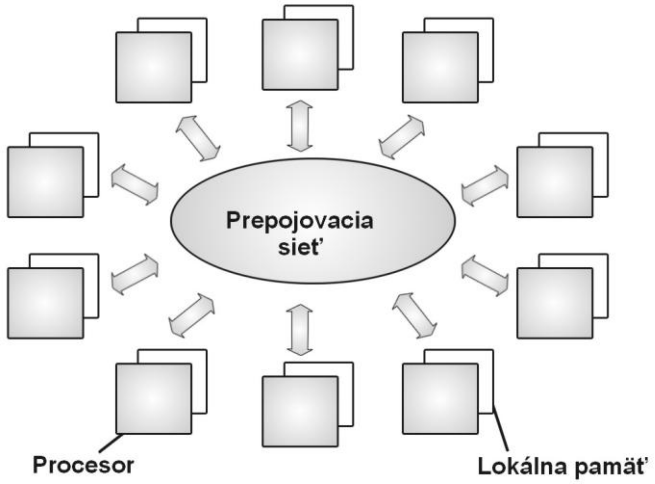
Hlavnou nevýhodou je to, že nie je dostatočné škálovateľná (s rastúcim stupňom paralelizmu problému nerastie úmerne efektívnosť vykonávania) pre jemnozrnne paralelné problémy, a to kvôli „hladovaniu", ktoré vzniká v dôsledku častej potreby na prístup k spoločnej pamäti. Procesy vykonávané na jednotlivých procesoroch hladujú – ich prácu a výpočte brzdí komunikácia so spoločnou pamäťou.

Príkladom architektúry MIMD so spoločnou pamäťou SM je počítač SGI Power Challenge.

DM – Architektúra MIMD s distribuovanou pamäťou

Architektúra MIMD s distribuovanou pamäťou je charakteristická tým, že

- 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.



Obrázok 4 Štruktúra architektúry DM - MIMD s distribuovanou pamätou

K najhlavnejším výhodám architektúry MIMD s distribuovanou pamäťou patrí:

- Podporuje škálovateľnosť omnoho viac ako architektúra MIMD so spoločnou pamäťou.
- Hoci jemnozrnne paralelné pravidelné problémy škáluje horšie ako architektúra SIMD, čím viac rastie nepravidelnosť problému, tým je podpora škálovateľnosti vyššia ako pri architektúre SIMD.
- Pre hrubozrnne paralelné problémy škáluje omnoho lepšie ako SIMD.

Jej nevýhodou je, že výkonnosť závisí na štruktúre a priepustnosti prepojovacej siete, keďže pri vzraste fyzickej vzdialenosti medzi procesormi rastie čas prístupu ku vzdialeným údajom. Preto je potrebné venovať pozornosť minimalizácii komunikačných nárokov.

Príkladmi architektúry MIMD s distribuovanou pamäťou sú počítače SGI Origin alebo Meiko Computing Surfaces.

K základným architektonickým riešeniam prepojovacej siete patrí:

- prepojenie procesorov zbernicou
- mriežkové prepojenie
- grafové prepojenie

VSM – Architektúra MIMD s virtuálnou spoločnou pamäťou

Architektúra MIMD s virtuálnou spoločnou pamäťou má tieto charakteristické rysy:

- 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.

Ako príklad tohto architektonického riešenia možno uviesť počítače Cray a v súčasnosti SGI.

Architektúra MISD

Paralelná architektúra MISD obsahuje špecializované rýchle procesory, komunikujúce prúdovým spôsobom. Vzhľadom na obmedzený aplikačný záber, MISD sa používa zriedkavo. Navyše, problém je možné dekomponovať iba využitím funkcionálnej dekompozície, ktorá je vhodná iba pre hrubozrnne paralelné problémy.

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 viest v porovnaní s nimi k väčšiemu poctu 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.

Nie všetky problémy sú efektívne paralelizovateľné. Aj vtedy, ak je nejaký problém efektívne paralelizovateľný, je nevyhnutné vybrať pre jeho riešenie vhodný algoritmus, ktorý opäť musí byt paralelný, v opačnom prípade nie je možné dosiahnuť zrýchlenie 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.
Hoci sekvenčný a paralelný algoritmus je funkčné rovnaký, paralelným algoritmom je potrebné venovať zvýšenú pozornosť, pretože môžu viest k vyčerpaniu zdrojov, a to najmä:

- k preplneniu pamäti
- k aritmetickému pretečeniu

Neohraničený a ohraničený paralelizmus

Neohraničený paralelizmus algoritmu je vyjadrený počtom paralelných krokov, t.j. paralelným časom výpočtu, neberúc pritom do úvahy zdroje počítačového systému.
Napríklad, zložitosť sčítania n hodnôt počítaného paralelne je O(log n) krokov, kde n je veľkosť problému.
Ohraničený paralelizmus algoritmu je vyjadrený paralelným časom výpočtu, berúc do úvahy (ohraničené) zdroje počítačového systému.

Efektívne a optimálne paralelné algoritmy

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 $p.T = TS$, kde TS je čas výpočtu pri použití najrýchlejšieho známeho sekvenčného algoritmu pre daný problém.
Ak má byt 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), a samozrejme
- 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 sekvenčné 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.

Efektívne paralelizovateľný problém možno prakticky rozpoznať na základe rozpoznania idealizovaných druhov paralelizmu, ktoré sú vlastné tomuto problému, a to aj vo vzájomnej kombinácii. 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 nežiaducim javom:

- presunom častí programu medzi hlavnou pamäťou a diskovou pamäťou v dôsledku nevhodnej záťaže procesorov, spôsobeného najmä:
  - preťažením v dôsledku nevhodnej dekompozície (napr. pri cyklickom priradení procesorov pri prúdovom spracovaní)
  - 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

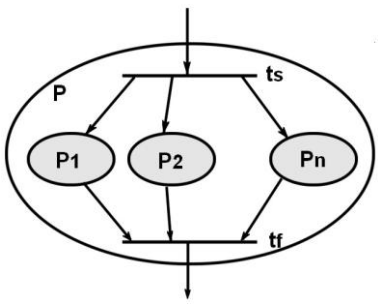
Druhy paralelizmu

Paralelizmus je inherentný efektívne paralelizovateľným problémom. Idealizované druhy paralelizmu sú tieto:

- Jednoduchý paralelizmus
- Prúdový paralelizmus
- Expanzívny paralelizmus
- Masívny paralelizmus

Jednoduchý paralelizmus

Problém *P* možno rozdeliť na množinu *n* podproblémov *P1, P2, . . . , Pn*, ktoré možno riešiť nezávisle, počnúc v čase t_s a končiac v čase t_f, podľa obrázku:



Obrázok 5 Jednoduchý paralelizmus problému P

Charakteristika jednoduchého paralelizmu

- Vykonávané úlohy (procesy), ktoré riešia podproblémy *Pi* sú počítané nezávisle a zvyčajné spracovávajú nezávislé množiny údajov.
- Výpočet končí vtedy, keď je ukončené vykonávanie všetkých procesov.
- Jednoduchý paralelizmus možno využiť na ľubovoľnej úrovni dekompozície problému a pre ľubovoľný typ architektúry.
- Je potrebné venovať pozornosť prideľovaniu času a vyváženému priraďovaniu procesov procesorom, ak počet procesorov *p* je menší ako veľkosť problému *n*.
- Jednoduchý paralelizmus je hrubozrnný (v praxi nie je možné dosiahnuť vysokú hodnotu *n*), keďže je založený na dekompozícii funkcie problému *P* na množinu rozdielnych funkcií definovaných pre podproblémy.
- Jednoduchý paralelizmus možno niekedy s úspechom využiť pri prúdovom paralelizme pre zlepšenie vyváženosti pri prúdovom spracovaní.

Jednoduchá dekompozícia

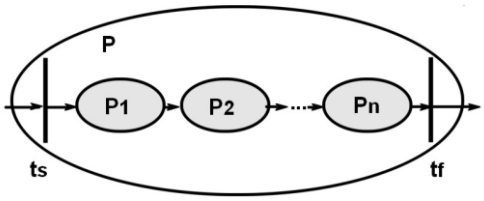
Ak T(Pi) je časový interval na vykonanie úlohy spojenej s riešením problému *Pi*, potom paralelný čas výpočtu $t_f – t_s$ pri $p = n$ procesoroch je daný maximom časových intervalov *T(Pi)*, t.j.

$T(n, P) = t_f – t_s = \max \{ T(Pi) \mid i = 1 \ldots n \}$, if $p = n$

Preto pri jednoduchej dekompozícii požadujeme približne rovnaké časové intervaly $T(P_i)$ riešenia jednotlivých podproblémov, t.j. $T(P_1) \approx T(P_2) \approx \ldots \approx T(P_n)$

Prúdový paralelizmus

Problém *P* možno rozdeliť na *n* podproblémov $P_1, P_2, \ldots, P_n$, ktoré sú riešené jeden po druhom v postupnosti, pričom spracovávajú rozsiahlu množinu údajov, počnúc spracovanie v čase t_s a končiac v čase t_f , podľa obrázku:



Obrázok 6 Prúdový paralelizmus problému P

Charakteristika prúdového paralelizmu

- Úlohy (procesy) počas vykonávania riešia rozdielne problémy *Pi*, pričom každý procesor spracováva údaje toho istého typu. Dekompozícia problému obsahujúceho prúdový paralelizmus sa nazýva funkcionálna dekompozícia.
- Vo všeobecnosti je obťažné dekomponovať problém do dostatočné dlhého prúdu podproblémov, čo je predpokladom pre využitie dostatočné veľkého poctu procesorov, pracujúcich paralelne.
- Navyše prúdové spracovanie je pre menší počet procesorov efektívne iba v tom prípade, ak je nasýtené dostatočné veľkou množinou údajov. Ide tu o problém nábehu, ktorého časový interval musí byt 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, ako sú napr. počítačové klastre, keďže ich využitie je univerzálnejšie ako architektúr MISD.

Funkcionálna dekompozícia

Za predpokladu, že *T(Pi)* je časový interval na riešenie podproblému *Pi* v prúde a spracovávaná množina údajov na vstupe obsahuje *m* prvkov, efektívne prúdové spracovanie pri použití *p* procesorov vyžaduje splnenie dvoch nasledujúcich podmienok.

1. V ideálnom prípade je potrebné problém *P* dekomponovať na $n = p$ podproblémov, pre ktoré platí

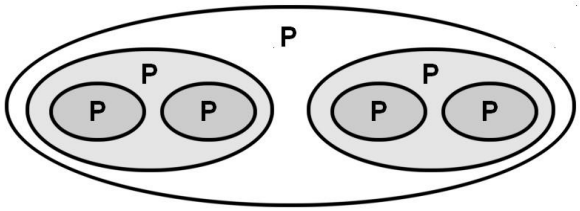
   $T(P_1) \approx T(P_2) \approx \ldots \approx T(P_n)$

   V opačnom prípade procesory budú zbytočne čakať na ukončenie práce iných procesorov.

2. Pre zabezpečenie vysokej výkonnosti výpočtu je potrebné, aby počet prvkov m spracovávanej vstupnej množiny bol dostatočne vysoký, aby dostatočne nasýtil prúd paralelne pracujúcich procesorov.

Expanzívny paralelizmus

Problém *P* je riešiteľný riešením množiny *n* tých istých problémov rekurzívne, vid obr.7. V špeciálnom prípade, ak $n = 2$, potom sa takáto metóda nazýva metóda *rozdeľuj a panuj*.



Obrázok 7 Expanzívny paralelizmus problému P

Charakteristika expanzívneho paralelizmu

- Tento paralelizmus môže byt buď hrubozrnný alebo jemnozrnný, v závislosti od poctu hierarchických úrovní riešenia problému počas výpočtu.
- Z funkčného hľadiska je dekompozícia problému triviálna (keďže tá istá funkcia je aplikovaná na rôznych úrovniach) 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

Dekompozícia expanzívne paralelného problému sa nazýva hierarchická dekompozícia.

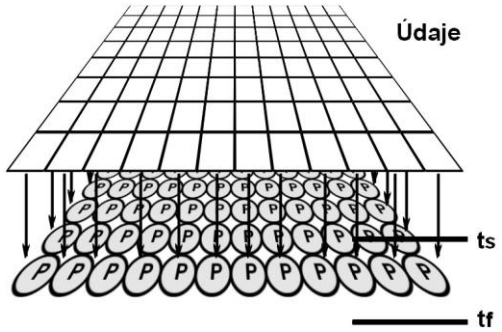
V prípade, že *T(P)* je časový interval na vykonanie úlohy pre riešenie problému *P*, ktorý je riešený rekurzívne dvoma tými istými úlohami, a to do *m* úrovní, teda ak celková veľkosť problému je $n = 2^m - 1$, celkový paralelný čas je *O(log n)*. Ako uvidíme, pri architektúrach SIMD je možné odvodiť redukovaný počet procesorov, t.j. menší počet ako očakávaných *O(n)*, pri ktorom algoritmus zachováva stále svoju optimálnosť.

V prípade architektúr MIMD nie je výhodné vykonávať úlohu s expanzívnym paralelizmom bez akéhokoľvek riadenia, pretože výkonnosť je daná nielen počtom procesorov, ako je to pri architektúrach SIMD, ale tiež komunikačnými nákladmi. Preto expanzívny problém binárneho vyváženého stromu má zmysel dekomponovať iba do úrovne, ktorej hodnota *m´* je rovná poctu procesorov *p*.

Ak maximálna úroveň m nie je známa, je nemožné využiť expanzívny paralelizmus v plnej miere, toto však nie je typický prípad. Hierarchická dekompozícia sa často používa na riešenie problémov pravidelnej povahy, napr. pri spracovaní údajov matice spracovaním podmatíc.

Masívny paralelizmus

Masívne paralelný problém pozostáva z rozsiahlej množiny problémov *P,* ktoré sú funkčné identické a spracovávajú paralelne rozsiahlu množinu údajov v tom istom čase, podľa obr.8. Preto sa tento druh paralelizmu nazýva tiež údajovým paralelizmom.



Obrázok 8 Masívny paralelizmus problému P

Charakteristika masívneho paralelizmu

- V typickom prípade ide o jemnozrnný paralelizmus, ktorý môže byt 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 množiny údajov.
- Pritom jednotlivé množiny údajov sa môžu aj navzájom prekrývať.

Údajová dekompozícia

Údajová dekompozícia je z hľadiska funkcie triviálna, keďže tá istá úloha sa vykonáva v podstate na nezávislých množinách údajov. Zaťaženie procesorov je závislé iba na dekompozícii údajov.
Ak na základe dekompozície údajov možno dosiahnuť funkciu, ktorá je do takej miery jednoduchá, aby bolo možné použiť architektúru SIMD, potom je možné dosiahnuť veľmi výkonný výpočet. Architektúra SIMD má totiž vysoký výkon pri riešení úloh pravidelnej povahy a využíva operácie pre posun polí, ktoré sú užitočné pre posun okien, cez ktoré sú prístupné rozsiahle údaje, organizované v mriežke.

Ak je masívny paralelizmus kombinovaný s iným druhom paralelizmu, aj vtedy je možné dekomponovať údaje hrubozrnnejším spôsobom, berúc pozorne do úvahy hranice prekrytia, a potom možno použiť s výhodou architektúru MIMD, či už v podobe superpočítača, alebo počítačového klastra.
Pri masívnom paralelizme je však nevýhodné použitie viacprocesorovej architektúry s procesormi rozdielneho výkonu, pretože potom je to potrebné zohľadňovať pri dekompozícii údajov.

Využitie paralelizmu pre výkonné výpočty

- Vo všeobecnosti aplikácie vysokého výkonu sú vyvíjané prioritným využitím masívneho a expanzívneho paralelizmu, ktoré sú inherentné problémom pravidelnej povahy. Tieto druhy paralelizmu môžu byt totiž nielen hrubozrnné, ale veľmi často sú jemnozrnné.
- Jednoduchý a prúdový paralelizmus je zvyčajné 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

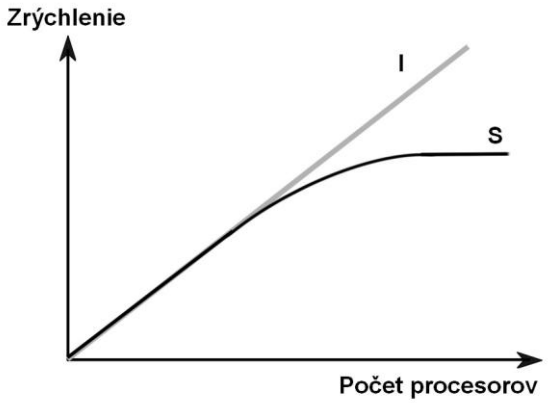
Nech *T(n, 1)* je sekvenčný čas vykonávania na jednom procesore a *T(n, p)* je čas vykonávania na *p* procesoroch, pri riešení toho istého problému veľkosti *n*. Zrýchlenie *S(n, p)* a efektívnosť *E(n, p)* sú definované nasledujúcimi vzťahmi:

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 poctu 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 poctu procesorov už nevedie k zvýšeniu zrýchlenia, vid obr.9, 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.



Obrázok 9 Ideálna (I) a skutočná (S) závislosť zrýchlenia od počtu procesorov

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 $\overline{S}(n,p)$ a celková efektívnosť $\overline{E}(n,p)$ sú definované takto:

$$\overline{S}(n,p)=\frac{T_N(n)}{T(n,p)} \qquad \overline{E}(n,p)=\frac{\overline{S}(n,p)}{p}$$

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

Amdahlovo pravidlo

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

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

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

Odvodenie Amdahlovho pravidla

Pre zjednodušenie predpokladajme, že všetky náklady paralelného výpočtu spôsobené paralelizáciou, vrátane nákladov spôsobených sekvenčnou časťou kódu možno spojiť dovedna. Číže predpokladajme, že všetky náklady sú nezávislé od poctu procesorov (čo nemusí byt vždy pravda). Potom celkový čas výpočtu T(n, 1) možno rozdeliť na paralelný a sekvenčný čas, podľa vzťahu: $T(n,p) = T^P(n) + T^S(n)$

Pri použití *p* procesorov dostávame paralelný čas v tvare: $T(n,p) = T^P(n)/p + T^S(n)$

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

Potom môžeme odvodiť:

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

$$= \frac{T(n,1)}{T^S(n) + T^P(n)/p}$$

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

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

$$= \frac{1}{\alpha + (1-\alpha)/p}$$

Nedostatok Amdahlovho pravidla

Pri odvodení Amdahlovho pravidla sme vychádzali z predpokladu, že platí $T^S(n) = T^S$ (t.j. hodnota je konštanta), berúc do úvahy konštantnú veľkosť problému. Podľa Amdahlovho pravidla masívne paralelné systémy (obsahujúce obrovský počet procesorov) nie sú užitočné, keďže ich výkon nemožno efektívne využiť.

Na druhej strane, je to aj otázkou vzťahu paralelných systémov a paralelných problémov. Výkonné paralelné systémy sa používajú na riešenie vysokoparalelizovateľných problémov, u ktorých je tendencia k škálovateľnosti pri vzrastajúcom pocte procesorov.

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

Paralelná časť kódu

Amdahlovo pravidlo je užitočné pri meraní paralelnej časti kódu $1-\alpha$ pri použití p procesorov, podľa vzťahu:

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

Treba však povedať, že Amdahlovo pravidlo nie je mierou kvality kódu. To znamená, že najlepší sekvenčný algoritmus môže byt predsa len rýchlejší ako algoritmus, ktorý je paralelizovateľný.

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

Programový model údajového paralelizmu má tieto najpodstatnejšie črty:

- 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).

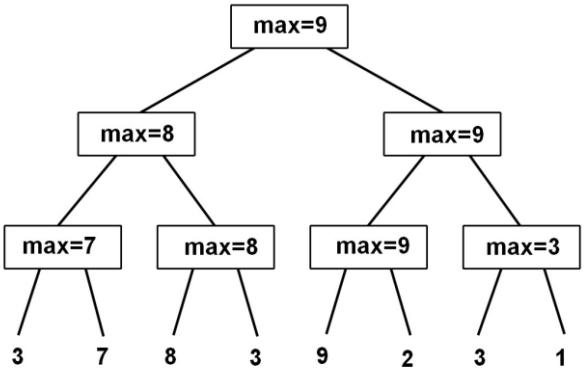
Masívny paralelizmus v modeli údajového paralelizmu

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:
$M = M1 \cup M2 \cup \ldots \cup M_n$ $N = N_1 \cup N_2 \cup \ldots \cup N_n$

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* $i = 1 \ldots n$.
Pritom je podstatné, že výpočet M_i nezávisí od N_j , pre $i \neq j$.
Porovnajme teraz vzorový sekvenčný a paralelný algoritmus pre nezávislé množiny údajov.

Expanzívny paralelizmus v modeli údajového paralelizmu

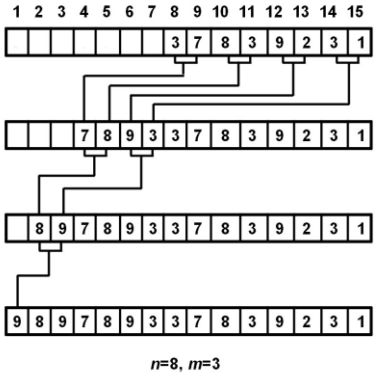
Využitie expanzívneho paralelizmu v modeli údajového paralelizmu je založené na vhodnom spôsobe zobrazenia spracovávaných údajov do pamäti a v nahradení rekurzie iteráciou.
Metóda Rozdeľuj a panuj
Pre ilustráciu predpokladajme problém nájdenia maximálnej hodnoty z množiny *{3, 7, 8, 3, 9, 2, 3, 1}* metódou Rozdeľuj a panuj, podľa obr. 10:



Obrázok 10 Hľadanie maximálnej hodnoty

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 pola veľkosti $2n - 1$, na pozície $n,(n + 1),\ldots,(2n - 1)$.



Obrázok 11 Metóda vyváženého stromu

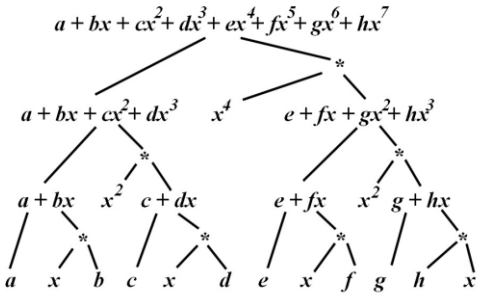
Zníženie poctu procesorov
Vezmime do úvahy algoritmus pre nájdenie maxima z n čísel pomocou metódy vyváženého stromu.
Tento algoritmus sa vykoná v čase *O(log n)* pri použití *n/2* procesorov, ktoré sú však potrebné iba v prvom kroku výpočtu. Preto vzniká otázka, či predsa len neexistuje menší počet procesorov *p*, t.j. $p < n/2$, ktorý je dostatočný pre zachovanie optimálnosti paralelného algoritmu.
Ako uvidíme z nasledujúceho odvodenia, je možné zredukovať počet procesorov na hodnotu:

$$p = \frac{n}{\log n}$$

Odvodenie redukovaného poctu procesorov

1. Predpokladajme $p < n/2$ procesorov a rozdeľme n prvkov na p skupín. Nech $(p - 1)$ skupín obsahuje $\lceil n/p \rceil$ a zvyšná skupina $n - (p - 1)\lceil n/p \rceil (\leq n/p)$ prvkov.
2. Priraďme procesor každej skupine, celkove máme p procesorov. Každý z nich hľadá maximum sekvenčné v rámci skupiny a paralelne s inými skupinami, v čase $\lceil n/p \rceil - 1 + \log p$
3. Substitúciou $p = n/\log n$ dostaneme $\lceil n/(n/\log n) \rceil - 1 + \log(n/\log n)$, 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).

Metóda binárneho stromu
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.12:



Obrázok 12 Metóda binárneho stromu

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.

Programový model odovzdávanie správ

Podstata modelu odovzdávania správ je takáto:

- Programátor pri riešení paralelného problému definuje sekvenčné procesy, ktoré môžu navzájom komunikovať formou odovzdávania správ.
- Za synchronizáciu a výmenu údajov medzi procesmi je zodpovedný teda programátor.
- Pri modeli odovzdávania správ sa namiesto paralelného jazyka používa sekvenčný jazyk, (napr. C alebo FORTRAN), 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, ktorého chovanie je záležitosťou súčasného výskumu. Navyše, je preukázané, že pre každú aplikáciu v modeli MPMD možno použiť taktiež model SPMD.

Pomocou modelu odovzdávania správ možno riešiť rôznorodejšie paralelné problémy, ako je to v modeli údajového paralelizmu, a to z týchto dôvodov:

- Programátor nie je obmedzovaný dekompozíciou problému nepravidelnej povahy, ktorá vedie k problému pravidelnej povahy. (Pri použití architektúry SIMD je tento problém riešený veľkým množstvom špecializovaných algoritmov.)
- 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 (a cenovo náročnejšou) prepojovacou sieťou alebo pomocou superpočítačov, najčastejšie v podobe jednoskrinových počítačov, založených na vzájomnej komunikácii veľkého množstva procesorov, ktorá je realizovaná vyspelou technológiou koordinačných procesorov.
- Pre model odovzdávania správ v súčasnosti existuje všeobecne akceptovaný štandard MPI (Message Passing Inteface), ktorý je implementovaný na veľkom počte rôznych superpočítačov a počítačových klastrov.

Programový model SPMD

Charakteristické pre programový model SPMD sú jeho nasledujúce vlastnosti:

- 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.

Pre priblíženie programového modelu SPMD uveďme spôsob práce v systéme LAM/MPI.

LAM/MPI (Local Area Multicomputer) je paralelné prostredie a zároveň vývojový systém pre sieť lokálnych nezávislých počítačov, zapojených do počítačového klastra, resp. pre sieť procesorov v superpočítači.

Systém LAM/MPI má tieto najdôležitejšie vlastnosti:

- LAM je úplnou implementáciou štandardu MPI, v podobe knižnice procedúr MPI,
- obsahuje navyše monitorovacie a testovacie prostriedky, použiteľné jednak počas výpočtu, jednak po jeho ukončení,
- je vhodný aj pre heterogénne siete počítačov,
- má prostriedky pre pridávanie a odstraňovanie uzlov v sieti,
- umožňuje testovanie chybných uzlov a obnovu výpočtu v prípade zistenia chybného uzla,
- umožňuje priamu komunikáciu medzi aplikačnými uzlami,
- umožňuje ovládanie zdrojov systému,
- umožňuje (v prípade štandardu MPI-2) aj vytváranie procesov počas výpočtu, a napokon
- umožňuje komunikáciu na základe viacerých protokolov, t.j. pre architektúry so spoločnou pamäťou (SM) aj s distribuovanou pamäťou (DM).

Zníženie komunikačných nákladov

Náklady na komunikáciu možno znížiť nasledujúcimi prostriedkami:

- návrhom vhodnej topológie procesov pri dekompozícii problému, t.j. takej ktorá je prinajmenšom podobná, ak už nie zhodná s topológiou procesorov – ich usporiadaním v prepojovacej sieti. To vedie k skráteniu ciest odovzdávania správy medzi procesormi.
- maximalizáciou dĺžok správ a minimalizáciou počtu volaní procedúr pre odovzdávanie správ. To znamená, že správy by mali byt dlhé a neodosielané príliš často.
- Prekrytím komunikácie a výpočtu v čase pomocou neblokujúcich operácií pre odovzdávanie správ.

Výkonnosť komunikácie

V štandarde MPI každá správa pozostáva z obálky a údajovej časti. Obálka správy má konštantnú veľkosť a obsahuje štyri údaje:

1. poradové číslo zdrojového procesu (source rank) – určuje, ktorému procesu bola správa odoslaná
2. značka správy (message tag) – umožňuje bližšie rozpoznať rôzne správy odovzdávané medzi tými istými dvoma procesmi.
3. poradové číslo cieľového procesu (destination rank)– určuje, ktorý proces má správu prijať
4. komunikátor – skupina (svet) procesov, v rámci ktorej môžu procesy komunikovať.

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 – t.j. časti programu, ktorá je vykonávaná na jednom procesore sekvenčným spôsobom. Pritom pojem operácie pre odovzdávanie správ v štandarde MPI sa používa nielen pre operácie, ktorých účinkom je prenos údajov od jedného procesu k inému, ale aj množstvo ďalších operácií, aktivovaných opäť príslušnými procedúrami, napr. na zisťovanie stavu prenosu správy, vytváranie skupín procesov a komunikátorov, vytváranie požadovanej topológie procesov, atd. V tomto širšom zmysle budeme operácie pre odovzdávanie správ označovať operáciami MPI a procedúry, ktoré ich aktivujú procedúrami MPI.

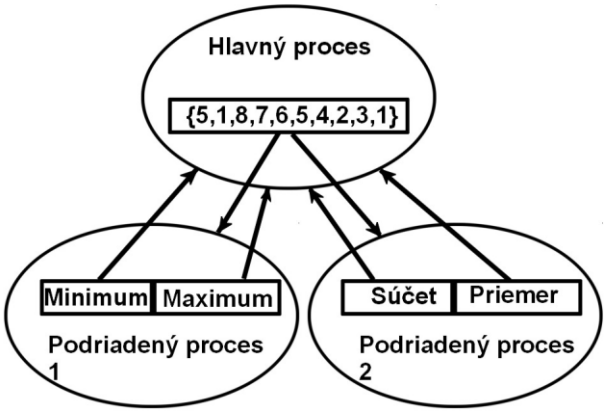
Z hľadiska lokálnosti účinku volania procedúr MPI, ktoré ich aktivujú operácie MPI, rozoznávame

Lokálne operácie MPI: Ich ukončenie (návrat z procedúry MPI) závisí iba na procese, ktorý ich volá. Tieto operácie nevyžadujú komunikáciu s inými procesmi.

Nelokálne operácie MPI: Ich ukončenie môže závisieť na vykonaní nejakej procedúry MPI volanej iným procesom. Takáto operácia môže vyžadovať komunikáciu s iným procesom definovaným používateľom.

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 bafer pre prijatie týchto údajov. Ďalej, nech každý z n procesov má svoje poradové číslo (rank) r z rozsahu $0 \ldots n-1$. 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 tým 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.



Obrázok 13 Príklad komunikácie medzi dvoma procesmi

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

Blokujúca:
Štandardný send

- int MPI_Send(void *buff,int count, MPI_DATATYPE,int disp, int tag,MPI_COMM comm)
- môže ale nemusí byť buffrovaný (závisí od implementácie a dĺžky správ)
- nelokálny(lebo musí byť buffrovaný) závisí na inom procese

Buffrovaný send:

- int MPI_Bsend(…)
- môže byť ukončený skôr, než príjemca zavolá receive
- lokálny
- 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

- int Mpi_Ssend(…)
- neskončí sa , pokiaľ príjemca nezavolá recevve a kým buffer nemôže byť použitý
- ukončenie implikuje že receive prebehol
- externá rutina na oneskorenie

Blokujúci receive

- MPI_Status-prímač môže špecifikovať ľubovoľnú hodnotu
- MPI_ANY_SOURCE –zdroj
- MPI_ANY_TAG- značky
- potom údaje vieme získať zo statusu – status.MPI_SOURCE,status.MPI_TAG,status.MPI_ERROR

Neblokujúca:

Štandardný send

- int MPI_Isend(...,MPI_Status *status,$ MPI_Reguest * req)
- môžeme testovať v akom je stave MPI_Test
- MPI_Wait čaká, kým nedôjde k odozve
- req – vnútorná štruktúra nás nezaujíma

Posielanie správ

- správy sa nepredbiehajú
- procesy primajú 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.

Skupinová komunikácia

Pri skupinovej komunikácii všetky procesy v skupine procesov tvoriacich komunikátor volajú tú istú procedúru MPI. Každý z procesov môže byt hlavným procesom, na druhej strane, nie všetky procedúry MPI pre skupinovú komunikáciu vyžadujú určenie hlavného procesu. Platí tiež, že nie všetky procedúry MPI pre skupinovú komunikáciu vyvolávajú vzájomné odovzdávanie správ. 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. Z toho vyplýva, že skupinová komunikácia je vhodná na riešenie paralelných problémov 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. Príkladom intrakomunikátora je MPI_COMM_WORLD. Komunikáciu medzi dvoma procesmi však možno využiť v rámci intrakomunikátorov, ale aj v rámci interkomunikátorov.

Procedúry MPI pre skupinovú komunikáciu

K najdôležitejším procedúram MPI pre skupinovú komunikáciu patria:

MPI Bcast Vysielanie (broadcast) – ak všetky procesy volajú procedúru MPI_Bcast s tým istým parametrom, označujúcim hlavný proces, potom tento hlavný proces (root) odošle správu všetkým procesom v komunikátore.

MPI Barrier Synchronizačná bariéra – všetky procesy po volaní procedúry MPI_Barrier budú zosynchronizované, t.j. budú po ukončení operácie synchronizácie pokračovať vo vykonávaní v tom istom čase.

MPI Reduce Redukcia, čiže výpočet výrazu – parametrom volaní procedúry MPI_Reduce je poradové číslo hlavného procesu a binárna asociatívna operácia. Obsahy bafrov odosielaných správ musia byt toho istého typu a sú operandami výrazu, ktorého hodnota bude prijatá hlavným procesom ako výsledok výpočtu výrazu. Napr. ak operandami sú celé čísla: c_1 odosielané z bafra procesom p_1, c_2 odosielané z bafra procesom p_2, . . . , c_n odosielané z bafra procesom pn, operácia je MPI_SUM, a ako hlavný proces je definovaný proces 0, potom tento proces prijme do (iného bafra) hodnotu ($c_1 + c_2 + \cdots + c_n$). Vzhľadom na to, že okrem základných operácií (minimum, maximum, súčet, súčin, . . .) je možné definovať aj vlastné zložité operácie pomocou procedúry MPI_Op_create, odosielané správy (operandy výrazu) nemusia byt iba jednoduchými číslami.

MPI Scatter Rozsypanie správy v tvare $D_0,D_1, \ldots ,D_{(n-1)}$, ktorú odošle hlavný proces, a to takým spôsobom, že proces s poradovým číslom k prijme položku D_k, pre $k = 0 \ldots n - 1$.

MPI Gather Pozhŕňanie údajov – inverzná operácia k operácii MPI_Scatter. Proces k odosiela správu D_k, pre $k = 0 \ldots n - 1$, a správu v tvare $D_0,D_1, \ldots ,D_{(n-1)}$ prijme hlavný proces.

MPI AllGather Na rozdiel od MPI_Gather výslednú správu prijme každý proces.

Spôsob práce pri skupinovej komunikácii možno ukázať na príklade použitia skupinovej operácie MPI_Bcast, ktorej interfejs je v tomto tvare:

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

a ktorej parametre majú tento význam:

INOUT	buf	začiatočná adresa bafra
IN	count	počet položiek v bafri
IN	datatype	typ položky
IN	root	poradové číslo hlavného procesu
IN	comm	komunikátor

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 MPI a ich vzťah k typom jazyka C sú uvedené v nasledujúcej tabuľke:

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	

Zobrazenie typu
Každý typ MPI napodobňuje typ jazyka jednotným spôsobom založeným na nasledujúcej definícii Typemap – zobrazenia typu:
Typemap = $\{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\}$

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 poctu polí, ktorých dĺžky treba určiť, podľa poctu relatívnych posunov a podľa poctu rôznych typov a to takto:
spojitý typ (contiguous) – je definovaný jednou hodnotou dĺžky pola, žiadnym posunom a jedným údajovým typom, z ktorého je konštruovaný.
vektor s obkrokmi (strided vector) – je definovaný jednou hodnotou dĺžky pola, jednou hodnotou posunu a jedným údajovým typom, z ktorého je konštruovaný.
indexový typ (indexed) – je definovaný viacerými dĺžkami polí, viacerými posunmi a jedným údajovým typom.
štruktúra (structure) – všetky tri parametre procedúry MPI pre konštrukciu štruktúry môžu byt viaceré.

Základný postup pri definícii nového typu je takýto:

1. Je potrebné definovať meno pre nový údajový typ, napr. datatype pomocou 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) Procedúra pre zaznamenanie nového typu má teda interfejs v tvare:

int MPI_Type_commit(MPI_Datatype *datatype)

Nový typ datatype možno používať dovtedy, kým sa tento typ neuvoľní pomocou procedúry

int MPI_Type_free(MPI_Datatype *datatype)

10. 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.

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 $0, \ldots, n-1$ umožňuje aplikáciu rôznych množinových operácií, na základe ktorých možno vytvárať nové komunikátory, rozdeľovať komunikátor, a pod.

Intrakomunikátory
Nový intrakomunikátor možno vytvoriť dvoma základnými spôsobmi:

- vytvorením duplikátu existujúceho komunikátora pomocou procedúry **MPI_Comm_dup**.
- rozdelením komunikátora na viacero komunikátorov pomocou procedúry **MPI_Comm_split**

Interfejs procedúry *MPI_Comm_dup* je v tvare:
*int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)*
kde

IN	comm	komunikátor
OUT	newcomm	duplikát komunikátora

Interfejs procedúry *MPI_Comm_split* je v tvare:
int MPI_Comm_split(MPI_Comm comm, int color,
*int key, MPI_Comm *newcomm)*
kde

IN	comm	komunikátor (handle)
IN	color	riadenie priradenia podmnožín (celé číslo)
IN	key	riadenie priradenia poradových čísel (celé číslo)
OUT	newcomm	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 poctu procesov vo vzdialenej skupine (intrakomunikátore) interkomunikátora
MPI_Comm_test_inter zistenie, či komunikátor je intrakomunikátorom alebo interkomunikátorom

Procedúra pre vytvorenie interkomunikátora má nasledujúci interfejs:

int MPI_Intercomm_create(MPI_Comm local_ comm,
int local_leader, MPI_Comm peer_comm,
int remote_leader, int tag,
*MPI_Comm *newintercomm)*

kde význam jednotlivých parametrov je takýto:

IN	local_comm	lokálny intrakomunikátor
IN	local_leader	poradové číslo vedúceho procesu v lokálnom intrakomunikátore
IN	peer_comm	dozerajúci komunikátor (obvykle MPI_COMM_WORLD)
IN	remote_leader	poradové číslo vedúceho procesu iného intrakomunikátora v dozerajúcom komunikátore
IN	tag	značka
OUT	newintercomm	nový interkomunikátor