TECHNICKÁ UNIVERZITA V KOŠICIACH
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Ján Kollár

METÓDY A PROSTRIEDKY PRE VÝKONNÉ PARALELNÉ VÝPOČTY

Recenzenti: prof. Ing. Štefan Hudák, DrSc.
prof. Ing. Milan Jelšina, CSc.

Vydanie tejto monografie bolo podporené Fakultou elektrotechniky a informatiky Technickej univerzity v Košiciach a projektom VEGA No.1/8134/01 – Väzba procesného funkcionálneho jazyka na MPI.

elfa, s.r.o., Košice
ISBN xx-xxxxx-xx-x

Obsah

Zoznam obrázkov

Predhovor

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. Vytváranie matematických modelov zložitých systémov bez fyzikov, biológov, matematikov a ďalších špecialistov v aplikačných oblastiach nie je mysliteľné. Na druhej strane, vytváranie programov, ktorých vykonaním možno simulovať daný systém na výkonných lokálnych sieťach počítačov – počítačových klastroch, alebo na superpočítačoch, je záležitosťou programátorov.

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.

Preto táto monografia je zameraná predovšetkým na metódy, na základe ktorých možno pristúpiť k analýze problémov paralelnej povahy, k ich dekompozícii a k voľbe efektívnych implementačných prostriedkov.

Vzhľadom na to, že súčasné superpočítače a počítačové klastre patria do kategórie distribuovaných počítačových systémov, značná časť knihy je venovaná štandardu MPI – Message Passing Interface, ktorý nielenže umožňuje tvorbu efektívnych aplikácií, ale svojou definíciou charakterizuje možnosti efektívnej implementácie. Orientácia na štandard MPI je z tohto pohľadu iba upozornením na charakteristické implementačné postupy v systémoch založených na odovzdávaní správ, a môže byť prvotnou informáciou pre čitateľa, ktorý chce v tejto oblasti pracovať.

Autor

Úvod

Výkonné paralelné výpočty sa uplatňujú v súvislosti s požiadavkou na predpovedanie vlastností systémov.

Napríklad pri konštrukcii áut alebo lietadiel nestačí vychádzať iba z nového návrhu na základe empirických poznatkov o predošlej verzii výrobku a vyvinúť iba konštrukčný postup a k nemu zodpovedajúcu technológiu výroby. Je potrebné rozsiahle testovanie matematického modelu, ktorý overí podstatné vlastnosti systému – auta alebo lietadla, a to predtým, než sa prikročí k výrobe prototypu. Je zrejmé, že cieľom modelu v tomto prípade je dosiahnutie požadovaných vlastností systému a zníženie nákladov vo výrobe a počas prevádzky systému.

Existujú aj systémy, u ktorých modelovanie a simulácia je nevyhnutnosťou, pretože ich prototyp nemožno vôbec skonštruovať. K takýmto systémom patria fyzikálne systémy reálne existujúce v prírode, ktorých prejavmi sú morské prúdy, prudké zmeny počasia, povodne, lavíny, atď. Cieľom modelovania týchto systémov je predchádzať ich nežiadúcemu účinku, a to na základe predpovede ich vývoja.

K rovnakému typu systémov patria biologické systémy, ktoré svojou podstatou sú najťažšie modelovateľné, pretože majú zložitú mikroštruktúru, pri ktorej vystihnúť ich podstatné vlastnosti modelom je mimoriadne zložité a niekedy aj nemožné – empirické postupy v medicíne sú toho dôkazom.

Vo všeobecnosti sa metodológia paralelného programovania 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 systémov reálneho času je často používaný pojem súbežné programovanie (concurrent programming) namiesto pojmu paralelné programovanie (parallel programming).

Cieľom súbežného programovania je špecifikácia komunikujúcich sekvenčných procesov v systéme, pričom nie je podstatné, aby výsledný systém bol implementovaný na paralelnom počítači. Metodológia programovania systémov reálneho času, hoci paralelná, nevylučuje totiž uplatnenie aj pseudoparalelného vykonávania na jednoprocesorovej architektúre počítača, ktorý má operačný systém s prideľovaním času.

Paralelizmus využívaný v systémoch reálneho času je zvyčajne hrubozrnnejší ako pri paralelných výpočtoch. Na druhej strane, vzhľadom na potrebu uspokojenia požiadaviek používateľa na čas, systémy reálneho času patria do kategórie časovo kritických systémov, čo vyžaduje potrebu modelovania, analýzy a verifikácie každého takéhoto systému, najčastejšie na báze Petriho sietí.

Podľa toho, na akej cieľovej architektúre je systém reálneho času vykonávaný, hovoríme o multiprogramovaní, viacnásobnom spracovaní alebo distribuovanom programovaní. Vzťah pseudoparalelizmu resp. paralelizmu, spôsobu programovania a cieľovej architektúry je prehľadne uvedený v tabuľke 1.

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

Tabuľka 1: Vzťah paralelizmu (P), programovania (PR) a architektúry (A)

Cieľ výkonných paralelných výpočtov, ktorým je táto kniha venovaná, je však iný ako pri systémoch reálneho času.

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 teda:

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

Už zo samotnej požiadavky na spracovanie veľkého množstva údajov často vyplýva nevyhnutnosť využitia paralelnej počítačovej architektúry, pretože je známe, že zvýšeniu efektívnosti výpočtu bráni nielen nedostatok procesorov, ale aj nedostatok pamäti.

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. častí kódu, ktoré sú vykonávateľné iba sekvenčne.

V ideálnom prípade, ak použijeme p procesorový systém, chceme, aby program bol vykonávaný p krát rýchlejšie. V praxi je to však ťažko dosiahnuteľné s ohľadom na uvedené náklady.

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.

Táto kniha má nasledujúce členenie.

V prvej kapitole sú uvedené základné charakteristiky paralelných počítačových architektúr, a to z pohľadu využiteľnosti ich paralelizmu pre riešenie paralelných úloh.

V druhej kapitole sú uvedené základné vlastnosti paralelných problémov a paralelných algoritmov, na základe ktorých možno teoreticky posúdiť, či daný problém je možné riešiť paralelným spôsobom.

Tretia kapitola je venovaná meraniu výkonnosti paralelných výpočtov. Cieľom tejto kapitoly nie je iba ukázať spôsob výpočtu výkonnosti na základe merania časových intervalov pri vykonávaní programu, ale aj o nadobudnutie praktickejšieho pohľadu na paralelné výpočty, ktorý je užitočný pre dekompozíciu problému.

Praktickou stránkou dekompozície paralelných problémov sa zaoberá štvrtá kapitola, v ktorej je uvedený prístup k odhaľovaniu paralelizmu problému na báze štyroch základných typov paralelizmu – jednoduchého, prúdového, masívneho a expanzívneho.

Programovým modelom sú venované posledné dve kapitoly. V piatej kapitole je uvedený model programovania na báze údajového paralelizmu, vo väzbe na architektúry SIMD. Neznamená to však, že údajový paralelizmus nemožno využiť aj pre druhý programový model, v súčasnosti najatraktívnejší, a to model odovzdávania správ, ktorému je venovaná rozsiahla šiesta kapitola.

Model odovzdávania správ je v poslednej šiestej kapitole vysvetľovaný na základe štandardu MPI – Message Passing Interface, ktorý je v súčasnosti všeobecne akceptovaným štandardom pre programovanie na báze odovzdávania správ. Navyše, tento štandard je v súčasnosti efektívne implementovaný a používa sa pre výkonné paralelné výpočty na mnohých počítačových klastroch a superpočítačoch vo svete. Cieľom tejto kapitoly však nie je detailná informácia o

všetkých procedúrach interfejsu MPI, tú možno nájsť v príručke MPI. Ide skôr o charakteristiku prístupov, ktoré sú v súčasnosti akceptované ako štandardné a zároveň implementačne mimoriadne efektívne.

Samostatnú stránku výkonných paralelných výpočtov tvoria jednak matematické metódy a jednak paralelné algoritmy. Tieto sú viazané v značnej miere na konkrétne aplikácie, nie na všeobecne použiteľné prístupy z pozície programovania výkonných paralelných výpočtov. I keď týmto dvom stránkam výkonných paralelných výpočtov, ktoré súvisia s ich interdisciplinárnym charakterom, sa táto kniha nevenuje, treba povedať, že pri riešení konkrétneho problému sú východiskami, na základe ktorých možno nájsť efektívne paralelné riešenie príslušného problému.

Kapitola 1

Paralelné architektúry

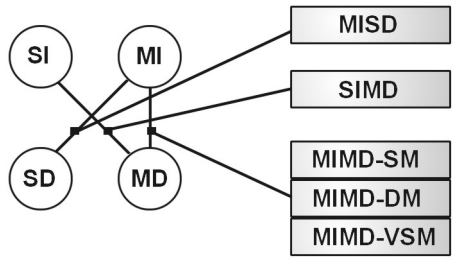
Pretože bez uvažovania vzťahu paralelného problému a paralelnej architektúry nie je možné správne problém ani dekomponovať, tobôž ho implementovať, je potrebné poznať základné charakteristiky paralelných architektúr. Ide pritom viac o pohľad z pozície programátora, ktorý sa musí rozhodnúť pre výber určitej architektúry, než o detailný opis štruktúry a funkcie jednotlivých druhov architektúr.

1.1 Flynnova klasifikácia paralelných architektúr

Na obr.1.1 je uvedená Flynnova klasifikácia paralelných počítačových architektúr, 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 inštrukcií (Multiple Data – MD). Takto by sme dostali štyri základné druhy architektúr: SISD, SIMD, MISD a MIMD.

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.



Obr. 1.1: Flynnova klasifikácia paralelných počítačových architektúr

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)

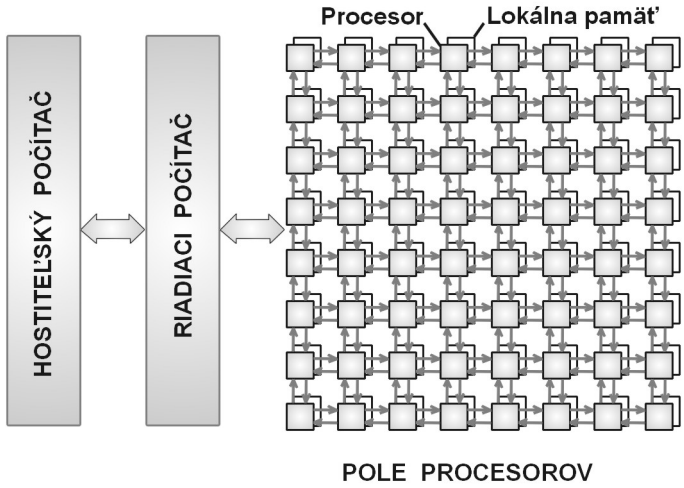
1.2 Architektúra SIMD

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.

Štruktúra architektúry SIMD ja uvedená na obr.1.2.

Výhodou architektúry SIMD je to, že je vhodná pre riešenie masívne paralelných problémov, kde tá istá operácia je vykonávaná na veľkom počte rozličných



Obr. 1.2: Štruktúra architektúry SIMD

objektov, napr. pri spracovaní obrazov, kde aktivuje spracovanie každého bodu obrazu naraz a rovnakým spôsobom.

Jej 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.

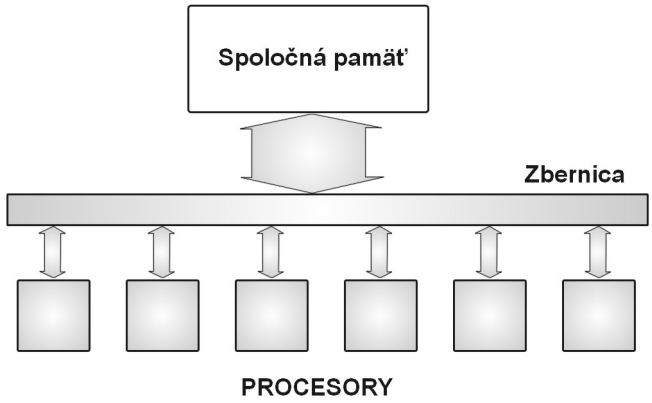
1.3 Architektúra MIMD

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.

1.3.1 SM – Architektúra MIMD so spoločnou pamäťou

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

Štruktúra architektúry MIMD so spoločnou pamäťou je uvedená na obr.1.3.



Obr. 1.3: 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 dostatočne 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 na výpočte brzdí komunikácia so spoločnou pamäťou.

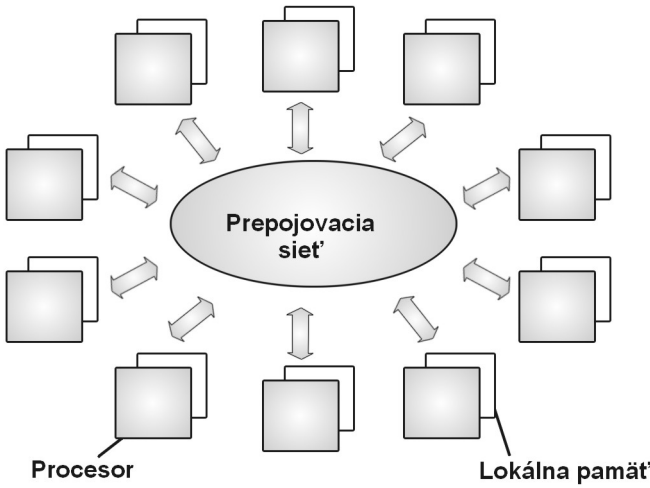
Príkladom architektúry MIMD so spoločnou pamäťou je počítač SGI Power Challenge.

1.3.2 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 výlučne prostredníctvom správ, keďže tu neexistuje žiadna spoločne používaná pamäť.

Štruktúra architektúry MIMD s distribuovanou pamäťou je uvedená na obr.1.4.



Obr. 1.4: Štruktúra architektúry DM – MIMD s distribuovanou pamäťou

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 (napr. v hyperkocke)

Súčasné technológie poskytujú možnosti prispôsobenia prepojovacej siete a tým aj celej architektúry podľa oblasti aplikácie, napr. pomocou prepínacích obvodov, smerovacích obvodov (nepriame prepojenie procesorov), a to buď v jednoskriňovej verzii alebo na báze klastrov počítačov.

1.3.3 VSM – Architektúra MIMD s virtuálnou spoločnou pamäťou

Architektúra MIMD s virtuálnou spoločnou pamäťou má tieto charakteristické rysy:

- Koncepčne 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.

1.4 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.

Kapitola 2

Paralelné problémy a paralelné algoritmy

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 byť paralelný, v opačnom prípade nie je možné dosiahnuť zrýchlenie výpočtu. Preto je potrebné dobre sa orientovať vo vzťahu paralelných problémov a paralelných algoritmov, ktorý je uvedený v tejto kapitole.

2.1 Inherentný paralelizmus

> 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.

V príklade 2.1.1 je problém sčítania riešený sekvenčným algoritmom a ten istý problém paralelným algoritmom.

Príklad 2.1.1

Problém – Sčítanie hodnôt: $x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8$

Sekvenčný algoritmus: $(((((((x_1 + x_2) + x_3) + x_4) + x_5) + x_6) + x_7) + x_8)$

Paralelný algoritmus $(((x_1 + x_2) + (x_3 + x_4)) + ((x_5 + x_6) + (x_7 + x_8)))$

Hoci sekvenčný a paralelný algoritmus je funkčne rovnaký, paralelným algoritmom je potrebné venovať zvýšenú pozornosť, pretože môžu viesť k vyčerpaniu zdrojov, a to najmä

- k preplneniu pamäti (v súlade s tézou paralelného výpočtu, uvedenou v časti 2.4) a
- k aritmetickému pretečeniu.

Príklad 2.1.2 je ukážkou vzniku aritmetického pretečenia pri paralelnom algoritme.

Príklad 2.1.2

Ak výraz $((30000 - 10000) - 20000) - 20000$ počítame paralelne, teda ak ho upravíme do paralelného tvaru

$$((30000 - 10000) - (20000 + 20000))$$

tento výpočet vedie k aritmetickému pretečeniu (40000), samozrejme za predpokladu, že prípustný rozsah celých čísel je iba $[-32768 \ldots 32767]$.

2.2 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 $\mathcal{O}(\log n)$ krokov, kde n je veľkosť problému.

> Ohraničený paralelizmus algoritmu je vyjadrený paralelným časom výpočtu, berúc do úvahy (ohraničené) zdroje počítačového systému.

2.3 Efektívne a optimálne paralelné algoritmy

Nech k je celočíselná konštanta a n je veľkosť problému.

> Potom efektívny paralelný algoritmus je vykonávaný v polylogaritmickom čase ($\mathcal{O}(\log^k n)$) 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), a samozrejme
- NP-úplné problémy.

2.4 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 $F(n)^{\mathcal{O}(1)}$ je zhodná s triedou problémov, ktoré možno riešiť sekvenčne v pamäti veľkosti $F(n)^{\mathcal{O}(1)}$.

Napr. paralelný výpočet podľa paralelného algoritmu reprezentovaného výrazom

$$(((x_1 + x_2) + (x_3 + x_4)) + ((x_5 + x_6) + (x_7 + x_8)))$$

sa realizuje v 3 krokoch (najprv možno počítať všetky štyri najvnútornejšie výrazy paralelne), sekvenčný výpočet vyžaduje pamäť (zásobník) veľkosti 3, vo všeobecnosti teda $\mathcal{O}(\log n)$, kde n je v tomto prípade 8.

Existuje však aj ekvivalencia času sekvenčného výpočtu a pamäti paralelného výpočtu. V tomto prípade, ak čas sekvenčného výpočtu je 7, pamäť paralelného výpočtu je 8 (v prvom kroku je potrebné mať v pamäti všetkých 8 hodnôt), teda $\mathcal{O}(n)$.

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.

2.5 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.

Kapitola 3

Meranie výkonnosti paralelných programov

V tejto kapitole sú uvedené charakteristiky paralelných programov, ktoré možno vyhodnotiť pri vykonávaní programu. Ich vyhodnotenie môže nielen upozorniť na nesprávnu paralelizáciu problému, napr. voľbu nevhodných algoritmov a chybnú dekompozíciu, ale aj na nevhodnú štruktúru a parametre architektúry, teda na nedostatočné zdroje.

3.1 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 problénmu veľkosti n. Zrýchlenie $S(n,p)$ a efektívnosť $E(n,p)$ sú definované nasledujúcimi vzťahmi:

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

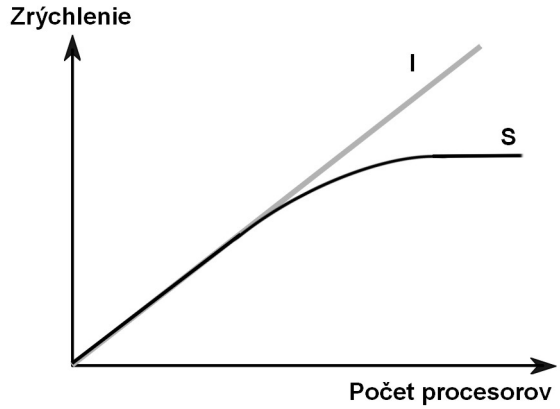
$$E(n,p) = \frac{S(n,p)}{p}$$

Možno si všimnúť, že aj $T(n,1)$ aj $T(n,p)$ možno merať.

3.2 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.3.1, 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.



Obr. 3.1: Ideálna (I) a skutočná (S) závislosť zrýchlenia od počtu procesorov

3.3 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:

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

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

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.

3.4 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 $0 \leq \alpha \leq 1$. Potom, maximálne zrýchlenie dosiahnuteľné na paralelnom počítači s p procesormi je ohraničené vzťahom
>
> $$S(n,p) = \frac{1}{\alpha + (1-\alpha)/p} \leq \frac{1}{\alpha}$$

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

Kvôli lepšej orientácii v súvislostiach sekvenčnej a paralelnej časti vykonávaného kódu uvedieme aj odvodenie Amdahlovho pravidla.

3.4.1 Odvodenie Amdahlovho pravidla

Pre zjednodušenie predpokladajme, že všetky náklady paralelného výpočtu spôsobené paralelizáciou, vrátane nákladov spôsobených sekvenčnou časťou kódu možno spojiť dovedna. Čiže 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

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

Definujme

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

a tiež

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

Potom môžeme odvodiť

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

3.4.2 Nedostatok Amdahlovho pravidla

Pri odvodení Amdahlovho pravidla sme vychádzali z predpokladu, že platí $T^S(n) = T^S$ (t.j. hodnota α 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 počte 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 počte procesorov a Amdahlovo pravidlo stráca značne svoj význam.

3.5 Alternatívna forma Amdahlovho pravidla

Alternatívna forma Amdahlovho pravidla je takáto:

> Nech $\overline{\alpha}$ označuje čas potrebný na vykonanie sekvenčnej časti kódu v paralelnom systéme s p procesormi. Maximálne dosiahnuteľné zrýchlenie je potom ohraničené nasledujúcim vzťahom

$$S'(n,p) = p\ (1-\overline{\alpha}) + \overline{\alpha} \leq p$$

$S'(n,p)$ sa zvykne nazývať *škálovaným zrýchlením*. Je rovné podielu $T'(n,1)$ a $T'(n,p)$, kde $T'(n,1)$ je čas v ktorom by sa paralelný program vykonal na jednom procesore za predpokladu dostatku dostupných zdrojov (pamäti) počítača.

Odvodenie alternatívnej formy Amdahlovho pravidla je ponechané na čitateľa. Je pritom potrebné brať do úvahy nasledujúce vzťahy:

$$\begin{array}{lcl} T'(n,p) & = & T'^P(n) + T'^S(n) \\ T'(n,1) & = & p.T'^P(n) + T'^S(n) \\ \overline{\alpha} & = & \frac{T'^S(n)}{T'(n,p)} \end{array}$$

V prípade škálovateľných aplikácií opäť platí, že $\overline{\alpha}$ je často veľmi malá hodnota, a preto pri vysokom počte procesorov možno dosiahnuť vysoké škálované zrýchlenie.

3.6 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 byť predsa len rýchlejší ako algoritmus, ktorý je paralelizovateľný.

Kapitola 4

Dekompozícia paralelných problémov

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.

4.1 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 (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

4.2 Druhy paralelizmu

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

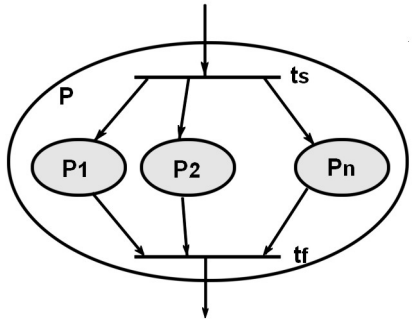
- Jednoduchý paralelizmus
- Prúdový paralelizmus
- Expanzívny paralelizmus
- Masívny paralelizmus

4.3 Jednoduchý paralelizmus

Problém P možno rozdeliť na množinu n podproblémov P_1, P_2, ..., P_n, ktoré možno riešiť nezávisle, počnúc v čase t_s a končiac v čase t_f, podľa obr.4.1

4.3.1 Charakteristika jednoduchého paralelizmu

- Vykonávané úlohy (procesy), ktoré riešia podproblémy P_i sú počítané nezávisle a zvyčajne 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 vyváženého priraďovaniu procesov procesorom, ak počet procesorov p je menší ako veľkosť problému n.



Obr. 4.1: Jednoduchý paralelizmus problému P

- 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.

4.3.2 Jednoduchá dekompozícia

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

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

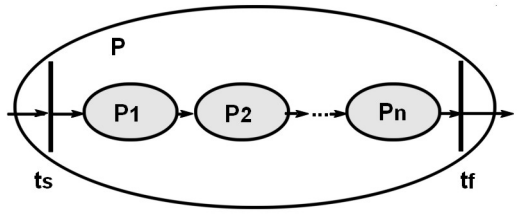
Preto pri jednoduchej dekompozícii požadujeme približne rovnaké časové intervaly $T(P_i)$ riešenia jednotlivých podproblémov, t.j.

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

4.4 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 obr.4.2.



Obr. 4.2: Prúdový paralelizmus problému P

4.4.1 Charakteristika prúdového paralelizmu

- Úlohy (procesy) počas vykonávania riešia rozdielne problémy P_i, pričom každý procesor spracováva údaje toho istého typu. Dekompozícia problému obsahujúceho prúdový paralelizmus sa nazýva funkcionálna dekompozícia.

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

- Navyše prúdové spracovanie je pre menší počet procesorov efektívne iba v tom prípade, 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, ako sú napr. počítačové klastre, keďže ich využitie je univerzálnejšie ako architektúr MISD.

4.4.2 Funkcionálna dekompozícia

Za predpokladu, že $T(P_i)$ je časový interval na riešenie podproblému P_i 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

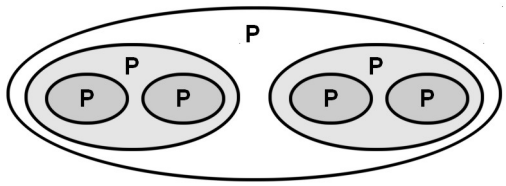
   $$T(P_1) \approx T(P_2) \approx \ldots \approx T(P_n)$$

   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.

4.5 Expanzívny paralelizmus

Problém P je riešiteľný riešením množiny n tých istých problémov rekurzívne, viď obr.4.3. V špeciálnom prípade, ak $n = 2$, potom sa takáto metóda nazýva metóda *rozdeľuj a panuj*.



Obr. 4.3: Expanzívny paralelizmus problému P

4.5.1 Charakteristika expanzívneho paralelizmu

- Tento paralelizmus môže byť buď hrubozrnný alebo jemnozrnný, v závislosti od počtu 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).

4.5.2 Hierarchická dekompozícia

Dekompozícia expanzívne paralelného problému sa nazýva hierarchická dekompozícia.

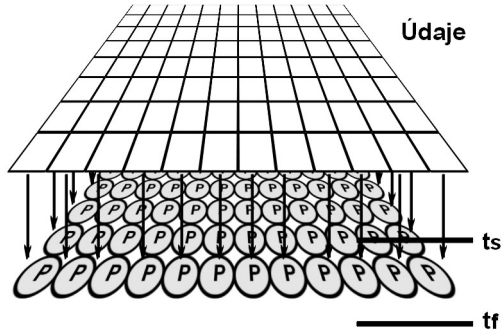
V prípade, že $T(P)$ je časový interval na vykonanie úlohy pre riešenie problému P, ktorý je riešený rekurzívne dvoma tými istými úlohami, a to do m úrovní, teda ak celková veľkosť problému je $n = 2^m - 1$, celkový paralelný čas je $\mathcal{O}(\log n)$. Ako uvidíme, pri architektúrach SIMD je možné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

4.6 Masívny paralelizmus

Masívne paralelný problém pozostáva z rozsiahlej množiny problémov P, ktoré sú funkčne identické a spracovávajú paralelne rozsiahlu množinu údajov v tom istom čase, podľa obr.4.4. Preto sa tento druh paralelizmu nazýva tiež údajovým paralelizmom.



Obr. 4.4: Masívny paralelizmus problému P

4.6.1 Charakteristika masívneho paralelizmu

- 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 množiny údajov.
- Pritom jednotlivé množiny údajov sa môžu aj navzájom prekrývať.

4.6.2 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.

4.7 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 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.

4.8 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 didtribuovanou pamäťou, ktoré sú realizované buď ako superpočítače alebo ako klastre počítačov v lokálnej sieti.

Kapitola 5

Programový model údajového paralelizmu

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 dosahli vysokú rýchlosť komunikácie a sú pre ne dostupné aj výkonné paralelné jazyky (napr. HPF - High Performance Fortran).

5.1 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 $i = 1 \ldots n$.

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.

5.1.1 Sekvenčný algoritmus pre nezávislé množiny údajov

Predpokladajme, že pole má $n = 1000$ prvkov. Sekvenčný algoritmus pre nezávislé množiny údajov je takýto:

```
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
```

kde f_i is ľubovoľná funkcia, ktorej definícia pre každú iteráciu je rovnaká.

Napr. $f_i\ (N[i])$ by mohla byť definovaná výrazom $2 * i * N[i]$, ak typ_1 a typ_2 je INTEGER (vtedy $M[i]$ budú jednoprvkové množiny).

Zložitosť sekvenčného algoritmu je $\mathcal{O}(n)$, v uvedenom prípade 1000 sekvenčných krokov (1000 iterácií cyklu).

5.1.2 Paralelný algoritmus pre nezávislé množiny údajov

Ak použijeme v modeli údajového paralelizmu jazykovú konštrukciu paralelného jazyka FORALL . . . IN PARALLEL DO, paralelný algoritmus, funkčne zhodný s uvedeným sekvenčným, je takýto:

```
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
```

Neformálna sémantika príkazu FORALL je nasledovná:

1. Najprv sa generujú hodnoty i v rozsahu $1 \ldots n$.

2. Potom sa vykoná telo príkazu FORALL pre všetky hodnoty i, ktoré boli generované v prvom kroku, na samostatných procesoroch.

Je zrejmé, že zložitosť paralelného algoritmu je $\mathcal{O}(1)$ – výpočet sa vykoná v jednom kroku. Ak je k dispozícii 1000 procesorov, zrýchlenie pri paralelnom výpočte je 1000.

5.1.3 Paralelné násobenie matíc

Ďalším príkladom uplatnenia údajoveho paralelizmu pri programovaní je paralelné násobenie matíc, ktoré je definované nasledujúcim algoritmom.

```
VAR A, B, C : ARRAY[1..n, 1..n] OF REAL;
      i, j, k : INTEGER;
BEGIN
  (* jeden krok inicializacie *)
  FORALL i = [1 .. n] AND j = [1 .. n] IN PARALLEL DO
    C[i, j] := 0.0
  END FORALL
  (* n krokov sumarizacie *)
  FOR k := 1 TO n DO
    FORALL i = [1 .. n] AND j = [1 .. n] IN PARALLEL DO
      C[i, j] := C[i, j] + A[i, k] * B[k, j]
    END FORALL
  END FOR
END
```

Zložitosť paralelného násobenia matíc je $\mathcal{O}(n)$ (za cenu n^2 násobného zvýšenia nárokov na pamäť).

Uvedený paralelný algoritmus možno ešte zlepšiť paralelným algoritmom sčítania (namiesto postupného pripočítavania hodnôt $A[i,k]*B[k,j]$ do premennej $C[i,j]$), a dosiahnuť v konečnom dôsledku zložitosť $\mathcal{O}(\log n)$.

5.1.4 Zníženie pamäťových nárokov

V určitých prípadoch možno nezávislé množiny údajov spracovávať s použitím jediného poľa, napríklad množina údajov indexovaných párnymi indexami je disjunktná s množinou údajov indexovaných nepárnymi indexami, ako je to v nasledujúcom príklade.

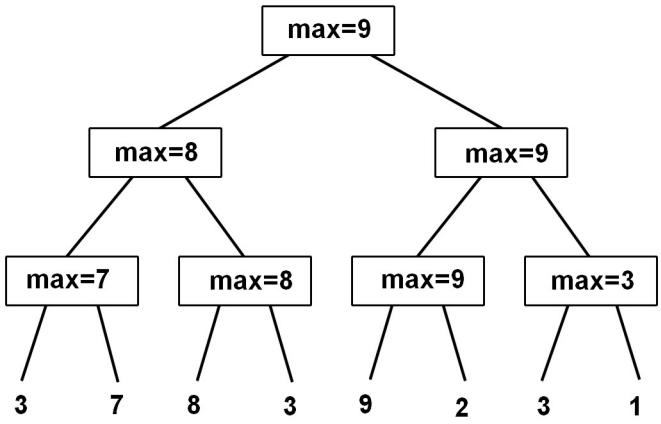
```
VAR N : ARRAY[1..2 * n] OF INTEGER;
      i : INTEGER;
BEGIN
  FORALL i = [1 .. n] IN PARALLEL DO
    N[2 * i] := N[2 * i] * 2;
    N[2 * i - 1] := N[2 * i - 1] div 2;
  END FORALL
END
```

5.2 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.

5.2.1 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.5.1



Obr. 5.1: Hľadanie maximálnej hodnoty

Klasické rekurzívne riešenie je dané optimálnym paralelným algoritmom, založeným na rekurzívnom volaní funkcie *maximum*.

```
TYPE T = ARRAY[1..n] OF INTEGER;

FUNCTION maximum(A : T; i, j : INTEGER) : INTEGER;
VAR max1, max2, k : INTEGER;
BEGIN
  IF i < j THEN
    FORALL k = [1..2] IN PARALLEL DO
      max1 := maximum(A, i, (i + j) div 2);
      max2 := maximum(A, (i + j) div 2 + 1, j)
    END FORALL;
    RETURN max(max1, max2)
  ELSE
    RETURN A[i]
  FI
END maximum;
```

Napriek teoretickej zložitosti $\mathcal{O}(\log n)$ 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.

5.2.2 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,$(n + 1)$,...,$(2n - 1)$.

Nech teda vstupné údaje sú

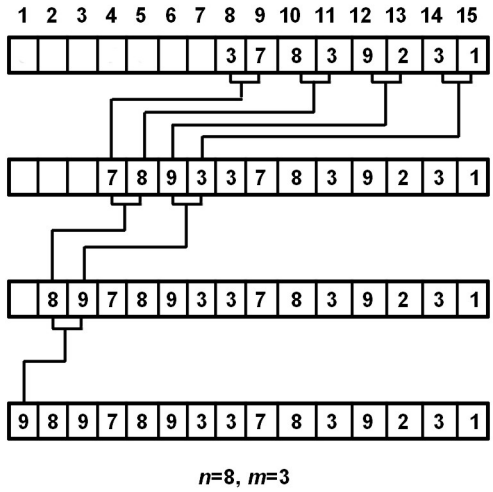
$$A[n], A[n + 1], \ldots, A[2 * n - 1]$$

Potom metóda vyváženého stromu je definovaná nasledujúcim algoritmom.

VAR A : ARRAY$[1..2 * n - 1]$ OF INTEGER;

FOR $k := m - 1$ STEP -1 TO 0 DO
  FORALL $j = [2^k \ .. \ 2^{k+1} - 1]$ IN PARALLEL DO
    $A[j] :=$ **max** $(A[2 * j], \ A[2 * j + 1])$
  END FORALL
END FOR

Vykonávanie je ilustrované na obr.5.2. Výslednou maximálnou hodnotou je $A[1]$.



Obr. 5.2: Metóda vyváženého stromu

5.2.3 Zníženie počtu procesorov

Vezmime do úvahy algoritmus pre nájdenie maxima z n čísel pomocou metódy vyváženého stromu.

Tento algoritmus sa vykoná v čase $\mathcal{O}(\log n)$ 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 počtu 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 sekvenčne 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 $\mathcal{O}(\log n)$. 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).

5.2.4 Metóda binárneho stromu

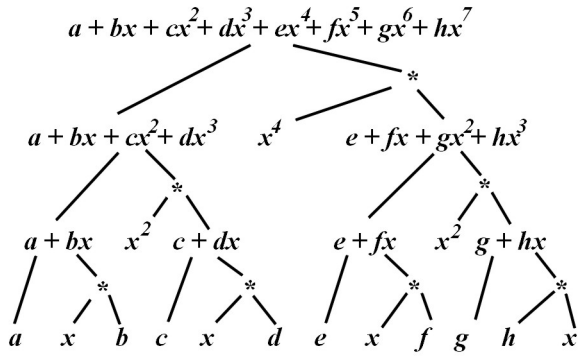
Metóda binárneho stromu je ďalšou z mnohých metód, ktoré sú používané pri využití expanzívneho paralelizmu v programovom modeli údajového paralelizmu. Táto metóda na rozdiel od predošlej nevyžaduje, aby bol strom výpočtu vyvážený.

Pre ilustráciu predpokladajme, že je potrebné vypočítat 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.5.3.



Obr. 5.3: Metóda binárneho stromu

Kapitola 6

Programový model odovzdávania 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 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 jednoskriňových 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.

6.1 Programový model SPMD

Charakteristické pre programový model SPMD sú jeho nasledujúce vlastnosti:

- Tá istá časť programu alebo rôzne časti programu môžu byť 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).

6.1.1 Štart systému LAM/MPI

Pred štartom systému LAM/MPI je potrebné v súbore, napr. `hostfile.def`, definovať uzly, t.j. počítače lokálnej počítačovej siete, to buď vo forme ich fyzických adries IP, alebo pomocou symbolických mien.

Príklady dvoch možností definície súboru `hostfile.def`, ako aj priradenie čísla uzla `n`, sú uvedené v nasledujúcej tabuľke.

Priradené číslo uzla `n`	Obsah súboru `hostfile.def` v prípade	
	fyzického označenia uzlov	symbolického označenia uzlov
`0`	`147.232.34.41`	`node1.cluster`
`1`	`147.232.34.42`	`node2.cluster`
`...`	`...`	`...`
`9`	`147.232.34.50`	`node10.cluster`

Číslo uzla je priradené fyzickým uzlom v závislosti od ich poradia, v akom sú uvedené v súbore `hostfile.def`.

Za predpokladu, že súbor `hostfile.def` je vytvorený, je možné zaviesť systém LAM príkazom `lamboot`.

Ešte predtým je však dobre overiť, či všetky definované uzly sú dostupné, a to príkazom `recon`.

```
> recon -v hostfile
recon: -- testing n0 (node1.cluster)
recon: -- testing n1 (node2.cluster)
.........................................
recon: -- testing n9 (node10.cluster)
-----------------------------------------------------------
Woo hoo!
recon has completed successfully.  This means ...
>
```

Teraz možno odštartovať LAM pomocou príkazu `lamboot`:

```
> lamboot -v hostfile
LAM 6.3.1/MPI 2 C++/ROMIO - University of Notre Dame
Executing hboot on n0 (node1.cluster)...
Executing hboot on n1 (node2.cluster)...
.............................................
Executing hboot on n9 (node10.cluster)...
topology done
>
```

Po štarte systému LAM možno kedykoľvek overiť komunikáciu s jednotlivými uzlami pomocou príkazu `tping`, napr. pre uzol `n1` v tvare

```
> tping n1
  1 byte from n1 (o): 0.002 secs
  1 byte from n1 (o): 0.001 secs
  1 byte from n1 (o): 0.001 secs
^C
3 messages, 3 bytes (0.003K), 0.005 secs (1.204K/sec)
roundtrip min/avg/max: 0.001/0.002/0.002
>
```

Počas vykonávania paralelného programu (spôsob štartu programu uvedieme nižšie) možno zistiť stav výpočtu: stav vykonávaných procesov príkazom `mpitask`

a stav správ príkazom `mpimsg`. Taktiež možno zrušiť vykonávanie procesov a vymazať odovzdávané správy príkazom `lamclean`.

Na uvoľnenie systému LAM/MPI slúži príkaz `lamwipe`:

```
> lamwipe -v hostfile
LAM 6.3.1 - University of Notre Dame
Executing tkill on n0 (node1.cluster)...
Executing tkill on n1 (node2.cluster)...
.............................................
Executing tkill on n9 (node10.cluster)...
>
```

6.1.2 Používanie systému LAM/MPI pre model SPMD

Základná schéma programu v modeli SPMD (single program – multiple data) je daná obsahom súboru, nazvime ho `simple.c`, teda použijeme jazyk C.

```
#include <mpi.h>
................................................
..... Deklarácie lokálnych premenných ............
................................................
MPI_Init(&argc, &argv);
................................................
....... Časť kódu v ktorej je možné použiť .......
....... volania procedúr MPI  ...................
................................................
MPI_Finalize();
```

Pritom `#include <mpi.h>` umožňuje pripojenie knižnice procedúr MPI, ktoré možno v programe používať v časti kódu medzi volaním procedúry `MPI_Init` a `MPI_Finalize`.

Po kompilácii programu `simple.c` pomocou príkazu

```
> hcc -o simple simple.c -lmpi
>
```

vznikne vykonateľný program `simple`, ktorý možno odštartovať príkazom `mpirun`, za prepokladu, že bol predtým odštartovaný systém LAM príkazom `lamboot` a nebol uvoľnený príkazom `lamwipe`.

Štart programu príkazom `mpirun` znamená, že vykonávateľný program zo zdrojového uzla, definovaného v parametroch príkazu `mpirun` bude najprv prenesený na množinu uzlov (opäť definovanú v parametroch), na ktorej bude potom vykonávaný paralelne, resp. pseudoparalelne. Každá kópia vykonávateľného programu je procesom, ktorý používa lokálne premenné, t.j. uložené v pamäťovej oblasti, neprekrývajúcej sa s pamäťovými oblasťami iných procesov.

Ak napr. odštartujeme vykonávateľný program `simple` pomocou príkazu `mpirun` v nasledujúcej forme:

```
> mpirun -v n1,0,0,9,6,6,7,6 -s n5 simple
```

musí vykonávateľný program `simple` existovať na zdrojovom uzle `n5`, ktorý je fyzicky uzlom `node6.cluster` na základe našej definície súboru `hostfile.def`.

Celkový počet paralelných procesov je v tomto prípade 8, avšak dva procesy na uzle `n0` a ďalšie tri procesy na uzle `n6` sú vykonávané v pseudoparalelnom režime.

Celkový počet procesov môže počas vykonávania každý proces zistiť volaním procedúry

```
int MPI_Comm_size(MPI_Comm comm, int *size);
```

pričom parametre majú nasledujúci význam:

IN	`comm`	je komunikátor, t.j. skupina procesor, v rámci ktorej procesy navzájom môžu komunikovať
OUT	`size`	počet procesov v komunikátore

IN označuje vstupný parameter – prostredníctvom neho možno volanej procedúre MPI hodnotu dodať, OUT označuje výstupný parameter – prostredníctvom neho možno od volanej procedúry MPI hodnotu získať a zriedkavo používaný parameter INOUT je parameter označujúci možnosť dodania hodnoty a po ukončení vykonávania procedúry MPI aj možnosť získania novej hodnoty.

Mimoriadne dôležité je to, že každý proces môže sám seba identifikovať volaním procedúry

```
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

pričom parametre majú nasledujúci význam:

IN	`comm`	komunikátor
OUT	`rank`	poradové číslo procesu v komunikátore

To znamená, že volaním procedúry `MPI_Comm_rank` proces vie počas vykonávania zistiť svoje (vnútorné) poradové číslo (rank). Závislosť týchto poradových čísel procesov od uzlov je daná postupnosťou, v akej sú uzly uvedené v príkaze `mpirun`.

Napr. pre postupnosť uzlov `n1,0,0,9,6,6,7,6` použitú v predošlom príklade je táto závislosť nasledujúca:

poradové číslo								
uzla	n1	n0	n0	n9	n6	n6	n7	n6
procesu	0	1	2	3	4	5	6	7

Na základe definície súboru `hostfile.def` a postupnosti uzlov v príkaze `mpirun` vieme dokonca priradiť poradové čísla procesov fyzickým uzlom v sieti. To môže mať niekedy význam v heterogénnej sieti počítačov s nerovnakou výkonnosťou, keď poradie uzlov v príkaze `mpirun` môže mať vplyv na vyváženosť výpočtu. Napr. proces 3 je vykonávaný na uzle n9, teda na fyzickom uzle `node10.cluster`.

Ak je však sieť homogénna a je potrebné vykonať výpočet na všetkých uzloch v sieti, potom stačí použiť príkaz `mpirun` v tvare

```
> mpirun -v n0-9 -s n5 simple
```

alebo ešte jednoduchšie

```
> mpirun -v N -s n5 simple
```

Obidva vyššie uvedené tvary zodpovedajú tvaru

```
> mpirun -v n0,1,2,3,4,5,6,7,8,9 -s n5 simple
```

Cieľom rozvrhovania paralelného výpočtu v systéme LAM/MPI však nie je prideľovanie procesov na konkrétne fyzické uzly počítačového klastra, a už vôbec nie na fyzické procesory superpočítača. Naopak, cieľom je pracovať na vyššej úrovni abstrakcie, t.j. na základe identifikácie vnútorných poradových čísel procesov v komunikátoroch – skupinách vzájomne komunikujúcich procesov.

Pritom je možné pre testovanie záťaže uzlov a riadenie výpočtu využívať aj prostriedky merania času. Funkcia

```
double MPI_Wtime(void)
```

zistí hodnotu aktuálneho času výpočtu v sekundách, takže časový interval na vykonanie fragmentu programu možno merať napr. takto:

```
double starttime, endtime;
starttime = MPI_Wtime();
.....................................
.... meraný fragment programu ....
.....................................
endtime = MPI_Wtime();
printf(''Vykonanie fragmentu trvalo %f sekund.\n'',
                                      endtime-starttime);
```

6.2 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 byť dlhé a neodosielané priliš často.

- Prekrytím komunikácie a výpočtu v čase pomocou neblokujúcich operácií pre odovzdávanie správ.

6.3 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.

Komunikačná výkonnosť pri prenose správy, pozostávajúcej z obálky konštantnej veľkosti E (ktorá zabezpečuje jednak správne smerovanie správy k určenému cieľu a po jej prijatí rozpoznanie zdroja správy) a údajov používateľa D (ktorých veľkosť môže byť samozrejme rôzna) je závislá na

- Latentnosti – času štartu prenosu správy T_L
- Priepustnosti – (asymptotickej) prenosovej rýchlosti (v megabytoch za sekundu – Mbyte/s)
- Podielu veľkosti údajov D a veľkosti obálky E správy.

Celkový čas prenosu správy je T_M, podľa vzťahu

$$T_M = T_L + T_E + T_D$$

kde T_L je latentnosť, T_E je čas prenosu obálky správy a T_D je čas prenosu údajov, ktorými naplní správu používateľ predtým, než dôjde k prenosu. Prenos správy v čase je znázornený na obr.6.1, podľa ktorého je zrejmé, že celkový čas prenosu správy je tým menší, čím je menšia latentnosť a čím je vyššia priepustnosť, t.j. čím je kvalitnejší prenosový kanál prepojovacej siete.

Pretože obálka správy je nevyhnutná pre zabezpečenie správnej funkčnosti odovzdávania správy, pri daných parametroch prepojovacej siete má používateľ možnosť ovplyvniť dĺžku správ. Ukážeme, že je výhodnejšie odoslať menej dlhších správ ako viac kratších.

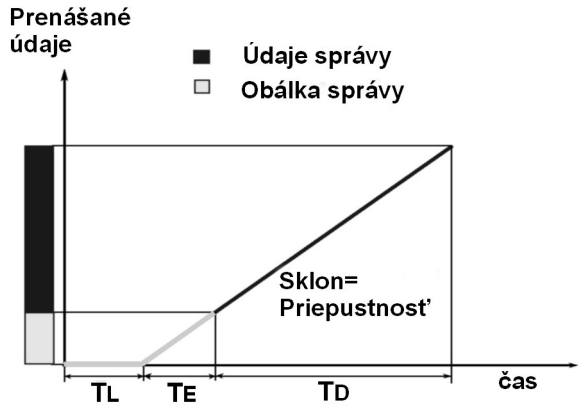
Predpokladajme správu s obálkou pevnej dĺžky E, obsahujúcu užitočné údaje D.

Ak je táto správa odoslaná, prenosový kanál je na čas $T_M^{(D)} = T_L + T_E + T_D$ obsadený. Ak rozdelíme údaje D na dve rovnaké časti, prepojovacia sieť spracováva jednu správu obsahujúcu užitočné údaje veľkosti $D/2$ v čase

$$T_M^{(D/2)} = T_L + T_E + T_D/2$$

Preto celkový čas potrebný na prenos dvoch správ je

$$T_M^{(D/2+D/2)} = 2T_M^{(D/2)} = 2(T_L + T_E) + T_D > T_M^{(D)}$$



Obr. 6.1: Prenos správy

Z toho je zrejmé, že je lepšie poslať jednu správu veľkosti D (v skutočnosti veľkosti $E + D$) ako dve správy veľkosti $D/2$ (v skutočnosti každú veľkosti $E + D/2$) zaťažené navyše dvojnásobnou latentnosťou.

6.4 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, atď. 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.

Z hľadiska programátora sú dôležité tri časové okamihy: čas volania procedúry MPI, čas návratu a čas vykonania aktivovanej operácie (napr. odovzdania správy). Čas začiatku operácie nie je potrebné brať do úvahy samostatne, pretože ak zanedbáme latentnosť, možno považovať čas volania procedúry MPI za zhodný s časom začiatku operácie.

Na druhej strane, k návratu z volanej procedúry môže dôjsť aj skôr, než bola operácia ukončená. Napr. čas odovzdania správy (ukončenia operácie odoslania správy) môže byť aj väčší ako času návratu z príslušnej procedúry MPI. Dokonca nie je nevyhnutné, aby správa bola v tomto prípade odovzdaná prijímajúcemu procesu aj fyzicky. To znamená, že 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.

Z hľadiska počtu procesov zúčastnených na komunikácii, rozoznávame

Komunikáciu medzi dvoma procesmi: (Point-to-Point communication) Táto komunikácia sa uskutočňuje medzi dvoma procesmi – odosielajúcim a prijímajúcim procesom a môže byť

- blokujúca: Ak návrat z procedúry MPI indikuje, že používateľ môže opätovne použiť zdroje špecifikované vo volaní (napr. bafer obsahujúci správu).
- 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.

Skupinovú komunikáciu: Každý proces v skupine procesov musí volať tú istú procedúru MPI.

6.5 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 ba-

fer 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 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.

Pri odoslaní správy odosielajúcim procesom sa obálka pripája k údajovej časti, nachádzajúcej sa v bafri a identifikuje správu. Pri prijatí správy sa do bafra prijímajúceho procesu ukladá opäť iba údajová časť.

Údajovú časť tvoria položky, každá toho istého typu. Typ položky je jedným z parametrov procedúr MPI pre odovzdávanie správ. Tento typ je však typom MPI, nie typom jazyka, v ktorom sa pripravuje program. Napr. pri programovaní v jazyku C, namiesto typu `char` jazyka C je potrebné uviesť typ MPI v tvare `MPI_CHAR`. Typy MPI zabezpečujú možnosť komunikácie v heterogénnej sieti, pretože bez ohľadu na rozdielnosť reprezentácie hodnôt typu jazyka na rôznych uzloch je reprezentácia typu MPI rovnaká.

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`	

Typ `MPI_PACKED` má špeciálny význam, ktorý nemá ekvivalent v jazyku C.

6.5.1 Poradie správ

Zprávy sa nepredbiehajú. Ak proces odošle dve správy po sebe tomu istému procesu, ktorý ich prijíma, sú prijaté v rovnakom poradí ako boli odoslané. Alebo inak, ak prijímajúci proces spracováva prvú správu, druhú nemôže začať prijímať. Táto vlastnosť zabezpečuje, že odovzdávanie správ je deterministické.

Na druhej strane, prijímajúci proces môže mať požiadavku na prijatie ľubovoľnej správy, na základe toho, že pri volaní procedúry MPI pre prijatie správy je možné namiesto konkrétneho poradového čisla zdrojového procesu uviesť špeciálny parameter (`MPI_ANY_SOURCE`) označujúci ľubovoľný zdrojový proces. Vtedy už kód programu pre odovzdávanie správ nie je deterministický.

Obidva uvedené prípady sa týkajú jednovláknových procesov, t.j. takých, pri ktorých dochádza k sekvenčnému vykonávaniu procesov na jednotlivých procesoroch.

Vzhľadom na organizáciu výpočtu, spočívajúcu v zoskupovaní procesov do špeciálnych nezávislých skupín procesov, zvaných komunikátory a možnosti komunikácie procesov v rámci komunikátorov aj medzi komunikátormi, je možné vytvárať aj viacvláknové procesy, pri ktorých dochádza k paralelnému vykonávaniu kódu.

Ak je proces viacvláknový, potom operácie v dvoch paralelných vláknach výpočtu nemajú určené logické usporiadanie v čase svojou fyzickou polohou v programe. Ak správy sú odoslané v dvoch paralelných vláknach, môžu byť prijaté jedným sekvenčným procesom v ľubovoľnom poradí a naopak, pre dve správy odoslané sekvenčným procesom po sebe neexistuje žiadna záruka, že budú v tomto poradí prijaté dvoma procesmi v dvoch rôznych paralelných vláknach. Preto v prípade viacvláknových procesov je odovzdávanie správ nedeterministické.

Z praktického hľadiska je možnosť nedeterministického vykonávania paralelného výpočtu pozitívnou vlastnosťou vtedy, ak ju programátor dokáže využiť, a negatívnou vtedy, ak si túto vlastnosť neuvedomí a dôjde k nedeterministickému výpočtu neočakávane.

Pri komunikácii medzi dvoma procesmi existujú štyri režimy odosielania správ: štandardný, bafrovaný, synchronizovaný a režim pripravenosti (prijatia správy). Režim prijímania správ je však iba jeden.

Vo všetkých prípadoch môže ísť o komunikáciu blokujúcu alebo neblokujúcu.

6.5.2 Blokujúca komunikácia medzi dvoma procesmi

Pri blokujúcej komunikácii 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. Na zistenie tejto chyby, a tým aj tejto situácie existujú v štandarde MPI prostriedky, ktoré môže programátor využiť.

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.

Operácia odovzdania správy v bafrovanom režime končí bezprostredne po volaní procedúry MPI, pretože údaje odosielané používateľom sú umiestňované do bafra alokovaného používateľom.

Operácie odovzdania správy v štandardnom režime a v režime pripravenosti končia dvojako: buď ako v synchrónnom, alebo ako v bafrovanom režime. Výber možnosti ukončenia je závislé na zdrojoch systému, na dĺžke správy, apod. a programátor ho nemá možnosť ovplyvniť.

Pretože iba bafrované blokujúce odoslanie správ je lokálne, ostatné režimy môžu spôsobiť zablokovanie výpočtu – čakanie na udalosť od iného procesu, ku ktorej však nemôže dôjsť.

Operácia pre prijatie správy pracuje iba v jednom režime a môže byť zodpovedajúcou operáciou operácii pre odoslanie správy v každom zo štyroch režimov. Blokujúca operácia pre prijatie správy končí vtedy, ak bafer prijímajúceho procesu obsahuje prijaté údaje.

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.

6.5.3 Procedúry MPI pre blokujúce odovzdávanie správ

Procedúra MPI pre štandardné odoslanie správy má nasledujúci interfejs:

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
```

Parametre procedúry majú nasledujúci význam:

IN	`buf`	začiatočná adresa bafra odosielajúceho procesu
IN	`count`	počet odosielaných prvkov z bafra (nezáporné celé číslo)
IN	`datatype`	údajový typ položky bafra
IN	`dest`	poradové číslo cieľového procesu
IN	`tag`	značka správy
IN	`comm`	komunikátor

Procedúra MPI pre bafrované odoslanie správy má nasledujúci interfejs:

```
int MPI_Bsend(void* buf, int count,MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm)
```

Pred volaním tejto procedúry je potrebné pripojiť bafer, pomocou procedúry

```
int MPI_Buffer_attach(void* buffer, int size)
```

kde

IN	`buffer`	je začiatočná adresa bafra
IN	`size`	je veľkosť bafra v bytoch

Po ukončení práce z bafrom možno bafer odpojiť pomocou procedúry

```
int MPI_Buffer_detach(void* buffer_addr, int* size)
```

kde

OUT	`buffer`	je začiatočná adresa bafra
OUT	`size`	je veľkosť bafra v bytoch

Procedúra MPI pre synchrónne odoslanie správy má nasledujúci interfejs:

```
int MPI_Ssend(void* buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm)
```

Procedúra MPI pre odoslanie správy v režime pripravenosti má nasledujúci interfejs:

```
int MPI_Rsend(void* buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm)
```

Procedúra MPI pre prijatie správy má nasledujúci interfejs:

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm,
             MPI_Status *status)
```

Jej parametre majú tento význam:

OUT	`buf`	začiatočná adresa bafra prijímajúceho procesu
IN	`count`	počet položiek v bafri prijímajúceho procesu
IN	`datatype`	údajový typ položky bafra
IN	`source`	poradové číslo zdrojového (odosielajúceho) procesu
IN	`tag`	značka správy
IN	`comm`	komunikátor
OUT	`status`	stav

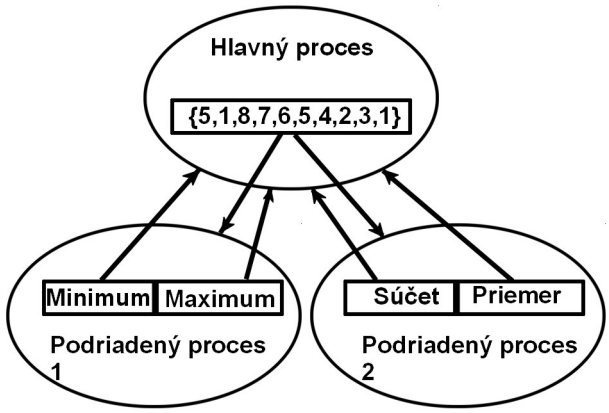
Vzhľadom na to, že pri prijímaní správy možno použiť namiesto konkrétneho poradového čísla zdrojového procesu hodnotu `MPI_ANY_SOURCE`, ktorá umožní prevziať správu od ľubovoľného zdrojového procesu, a tiež namiesto konkrétnej hodnoty značky hodnotu `MPI_ANY_TAG`, čo umožní prijať správu ľubovoľne označkovanú, stav obsahuje okrem informácie o tom, či prenos správy dopadol bez chyby alebo s identifikáciou chyby aj informáciu o konkrétnych hodnotách poradového čísla zdrojového procesu a značky správy, ktorá bola prijatá.
Preto stav obsahuje tieto položky:

`status.MPI_SOURCE`	poradové číslo zdrojového procesu
`status.MPI_TAG`	značka správy
`status.MPI_ERROR`	kód chyby

6.5.4 Príklad komunikácie medzi dvomi procesmi

Jednoduchý príklad komunikácie medzi hlavným procesom a dvoma podriadenými procesmi ukazuje nasledujúci príklad viď obr.6.2. Cieľom je definovať pole hodnôt $\{5, 1, 8, 7, 6, 5, 4, 2, 3, 1\}$ v hlavnom procese (s poradovým číslom 0) a poslať ho dvom podriadeným procesom s poradovými číslami 1 a 2.

Program je teda určený pre tri procesy, ktoré komunikujú v komunikátore `MPI_COMM_WORLD`, čo je systémom daný základný komunikátor, pričom komunikácia vždy prebieha medzi dvoma procesmi, nikdy nie v skupine všetkých troch procesov.



Obr. 6.2: Príklad komunikácie medzi dvoma procesmi

```
/*
 * Komunikacia medzi dvoma procesmi v systeme 3 procesov.
 */
#include <mpi.h>
#define BUFSIZE 10
int main(argc, argv)
int argc; char *argv[];
{ int size, rank;
  int slave;
  int buf[BUFSIZE];
  int n, value;
  float rval;
  MPI_Status status;
/* Inicializacia MPI  */
  MPI_Init(&argc, &argv);
/* Zistenie poctu procesov v komunikatore MPI_COMM_WORLD */
  MPI_Comm_size(MPI_COMM_WORLD, &size);
/* Zistenie poradoveho cisla procesu 0,1,2 */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank==0) { /* Hlavny proces 0 */
```

```
  buf[0]=5; buf[1]=1; buf[2]=8; buf[3]=7; buf[4]=6;
  buf[5]=5; buf[6]=4; buf[7]=2; buf[8]=3; buf[9]=1;
  printf("\n Odosielanie {5,1,8,7,6,5,4,2,3,1}");
  for (slave=1;slave < size;slave++) {
    printf("\n od hlavneho procesu %d procesu %d",rank,slave);
    MPI_Send(buf, 10, MPI_INT, slave, 1, MPI_COMM_WORLD);
  }
  printf("\n\n Prijimanie vysledkov od podriadenych procesov");
  MPI_Recv(&value, 1, MPI_INT, 1, 11, MPI_COMM_WORLD, &status);
  printf("\n Minimum %4d od procesu 1",value);
  MPI_Recv(&value, 1, MPI_INT, 2, 21, MPI_COMM_WORLD, &status);
  printf("\n Sucet   %4d od procesu 2",value);
  MPI_Recv(&value, 1, MPI_INT, 1, 12, MPI_COMM_WORLD, &status);
  printf("\n Maximum %4d od procesu 1",value);
  MPI_Recv(&rval, 1, MPI_FLOAT, 2, 22, MPI_COMM_WORLD, &status);
  printf("\n Priemer %4.2f od procesu 2",rval);
} else {
  if (rank==1) { /* proces 1 pre min a max */
    MPI_Recv(buf, 10, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
    value=100;
    for (n=0;n<BUFSIZE;n++) {
      if (value>buf[n]) { value=buf[n]; }
    }
    MPI_Send(&value, 1, MPI_INT, 0, 11, MPI_COMM_WORLD);
    value=0;
    for (n=0;n<BUFSIZE;n++) {
      if (value<buf[n]) { value=buf[n]; }
    }
    MPI_Send(&value, 1, MPI_INT, 0, 12, MPI_COMM_WORLD);
  } else { /* proces 2 pre sucet a priemer */
    MPI_Recv(buf, 10, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
    value=0;
    for (n=0;n<BUFSIZE;n++) {
      value=value+buf[n];
    }
    MPI_Send(&value, 1, MPI_INT, 0, 21, MPI_COMM_WORLD);
    rval= (float) value / BUFSIZE;
    MPI_Send(&rval, 1, MPI_FLOAT, 0, 22, MPI_COMM_WORLD);
  }
```

```
}
MPI_Finalize();  return(0);
}
```

Výstup programu je v nasledujúcom tvare:

```
Odosielanie  {5,1,8,7,6,5,4,2,3,1}
od hlavneho procesu 0 procesu 1
od hlavneho procesu 0 procesu 2

Prijimanie vysledkov od podriadenych procesov
Minimum     1 od procesu 1
Sucet      42 od procesu 2
Maximum     8 od procesu 1
Priemer  4.20 od procesu 2
```

6.5.5 Neblokujúca komunikácia medzi dvoma procesmi

Cieľom neblokujúcej komunikácie je zvýšenie výkonnosti prostredníctvom prekrytia výpočtu a komunikácie. K zvýšeniu výkonnosti výpočtu však nedochádza pri použití neblokujúcej komunikácie automaticky v každom prípade.

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. Ak chce programátor opätovne použiť tento bafer, musí sa o úplnom prenose správy presvedčiť volaním špeciálnej procedúry MPI, ktorá testuje ukončenie odoslania správy.

Podobne ako pri odosielaní správy, aj k návratu z procedúry pre neblokujúce prijatie správy dochádza bezprostredne po jej volaní, t.j. predtým, než je správa prenesená do bafra prijímajúceho procesu. Preto treba volať špeciálnu procedúru, ktorá testuje, či došlo k úplnému prenosu prijímanej správy.

Neblokujúce odosielanie správ možno kombinovať s blokujúcim prijímaním správ a naopak.

Pri modeli odovzdávania správ je komunikácia inicializovaná procesom odosielajúcim správu. Ak odosielanie správ má za následok vyčerpanie systémových zdrojov, dôjde k chybe, ktorú možno zistiť na základe jej kódu.

Vo všeobecnosti platí, že náklady na komunikáciu sú menšie, ak sa už vyskytlo volanie procedúry pre prijatie správy v momente, keď dôjde k volaniu procedúry MPI pre odoslanie správy.

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, pretože časté opakované testovanie na ukončenie operácie môže znižovať výkonnosť výpočtu väčšmi ako použitie blokujúcich operácií.

6.5.6 Procedúry MPI pre neblokujúce odovzdávanie správ

Procedúra MPI pre štandardné neblokujúce odoslanie správy má nasledujúci interfejs.

```
int MPI_Isend(void* buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Status *status,MPI_Request *request)
```

kde

OUT	`request`	je komunikačná požiadavka

Procedúra MPI pre bafrované odoslanie správy je `MPI_Ibsend`, pre synchrónne odoslanie správy je `MPI_Issend` a pre odoslanie správy v režime pripravenosti `MPI_Irsend`. Vo všetkých troch prípadoch parametre sú rovnaké, ako pri procedúre `MPI_Isend`.
Procedúra pre neblokujúce prijatie správy má interfejs v tvare:

```
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Status *status, MPI_Request *request)
```

kde komunikačná požiadavka má rovnaký význam ako pri neblokujúcich procedúrach pre odoslanie správ, t.j. evidovať stav vykonávania príslušnej neblokujúcej procedúry MPI.

Táto evidencia je parametrom procedúry `MPI_Test` testujúcej, či došlo k ukončeniu komunikácie, ako aj procedúry `MPI_Wait` čakajúcej, kým nedôjde k ukončeniu komunikácie.
Procedúra `MPI_Test` má nasledujúci interfejs:

```
int MPI_ Test(MPI_Request *request, int *flag, MPI_Status *status)
```

kde

INOUT	`request`	komunikačná požiadavka
OUT	`flag`	má hodnotu true, ak je operácia ukončená

Táto procedúra testuje, či správa bola odoslaná, resp. prijatá ($flag = true$). Ak áno, nastaví požiadavku `request` na hodnotu `MPI_REQUEST_NULL`. Je to lokálna operácia.

Procedúra `MPI_Wait` čaká, kým sa komunikácia neukončí a nastaví požiadavku request na hodnotu `MPI_REQUEST_NULL`. Nie je to lokálna operácia. Jej interfejs je v nasledujúcom tvare:

```
int MPI_Wait(MPI_ Request *request, MPI_Status *status);
```

kde

INOUT `request` communication request (handle)

To znamená, že ak bezprostredne za volaním neblokujúcej procedúry pre odoslanie správy nasleduje v programe volanie `MPI_Wait`, účinok je rovnaký, ako použitie zodpovedajúcej blokujúcej procedúry pre odoslanie správy.

6.5.7 Príklad neblokujúcej komunikácie

Nasledujúci fragment ukazuje možnosť použitia neblokujúcich procedúr v prípade, ak proces 0 môže urobiť nejaký užitočný výpočet (`vypocet 0`), pri ktorom nie je nevyhnutné, aby bolo odoslanie správy z bafra `buffer` ukončené. Takisto operácia prijatia správy druhým procesom môže byť aktivovaná, avšak nie je nevyhnutné, aby bola ukončená počas výpočtu, označeného ako `vypocet 1`.

```
.....................
if (rank == 0) {
  MPI_Isend(&buffer, 10, MPI_CHAR, 1, 20, MPI_COMM_WORLD,
            &request);
....... vypocet 0 ........
  MPI_Wait(&request, &status);
} else {
  MPI_Irecv(&buffer, 10, MPI_REAL, 0, 20, MPI_COMM_WORLD,
            &status, &request);
....... vypocet 1 .......
  MPI_Wait(&request, &status);
}
.....................
```

6.6 Skupinová komunikácia

Pri skupinovej komunikácii všetky procesy v skupine procesov tvoriacich komunikátor volajú tú istú procedúru MPI. Každý z procesov môže byť 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.

6.6.1 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 byť 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 p_n, 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 byť 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

6.6.2 Príklad skupinovej komunikácie

Predpokladajme, že komunikátor `MPI_COMM_WORLD` obsahuje 4 procesy a že hlavný proces má poradové číslo 0. Hlavný process rozošle správu obsahujúcu čísla $\{5, 1, 8, 7, 6, 5, 4, 2, 3, 1\}$ všetkým procesom, teda procesom 0,1,2 a 3. Treba si všimnúť, že iba proces 0 naplní svoj bafer, a zároveň všetky procesy volajú procedúru `MPI_Bcast` s hodnotou root=0. Preto proces 0 je hlavným procesom, t.j. tým, ktorý rozošle správu. Po prijatí tej istej správy štyrmi procesmi, každý proces vykonáva samostatný výpočet: proces 0 počíta minimum, proces 1 maximum, proces 2 súčet a proces 3 priemer. Výsledky procesy 1, 2 a 3 odosielajú procesu 0.

```
/*
 * Skupinova komunikacia medzi styrmi procesmi
 */
#include <mpi.h>
#define BUFSIZE 10

int main(argc, argv)
int argc; char *argv[];
{ int size, rank;
  int buf[BUFSIZE]={0,0,0,0,0,0,0,0,0,0};
  int n, value;
  float rval;
  MPI_Status status;
/* Inicializacia MPI  */
  MPI_Init(&argc, &argv);
/*
 * Zistenie poctu procesov
 */
  MPI_Comm_size(MPI_COMM_WORLD, &size);
 if (size==4) { /* Spravny pocet procesov */
/*
 * Urcenie poradoveho cisla procesu
 */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank==0) { /* definicia hodnot bafra len pre proces 0 */
    buf[0]=5; buf[1]=1; buf[2]=8; buf[3]=7; buf[4]=6;
    buf[5]=5; buf[6]=4; buf[7]=2; buf[8]=3; buf[9]=1;
    printf("\n Rozosielanie {5,1,8,7,6,5,4,2,3,1}");
  }
 /* vysielanie vsetkymi procesmi s hodnotou root=0 */
  MPI_Bcast(buf,10,MPI_INT,0,MPI_COMM_WORLD);
  if (rank==0) { /* hlavny proces */
    printf("\n Vysledky vypoctu");
    value=100;
    for (n=0;n<BUFSIZE;n++) {
      if (value>buf[n]) { value=buf[n]; }
    }
    printf("\n Minimum  %4d  procesom 0 ",value);
    MPI_Recv(&value, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
```

```
    printf("\n Maximum  %4d procesom 1",value);
    MPI_Recv(&value, 1, MPI_INT, 2, 0, MPI_COMM_WORLD, &status);
    printf("\n Sucet    %4d procesom 2",value);
    MPI_Recv(&rval, 1, MPI_FLOAT, 3, 0, MPI_COMM_WORLD, &status);
    printf("\n Priemer %4.2f procesom 3\n",rval);
  } else if (rank==1) { /* maximum */
      value=0;
      for (n=0;n<BUFSIZE;n++) { if (value<buf[n]) { value=buf[n]; }
    }
    MPI_Send(&value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
  } else if (rank==2) { /* sucet */
      value=0;
      for (n=0;n<BUFSIZE;n++) { value=value+buf[n];     }
      MPI_Send(&value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
  } else if (rank==3) { /* ave slave */
      value=0;
      for (n=0;n<BUFSIZE;n++) { value=value+buf[n];     }
      rval= (float) value / BUFSIZE;
      MPI_Send(&rval, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
  }
 }  MPI_Finalize(); return(0);
}
```

Výsledok výpočtu je v tvare:

```
Rozosielanie {5,1,8,7,6,5,4,2,3,1}

Vysledky vypoctu
Minimum        1  procesom 0
Maximum        8  procesom 1
Sucet         42  procesom 2
Priemer     4.20  procesom 3
```

6.7 Nové údajové typy

V štandarde MPI jednotný prenos správ v heterogénnej sieti počítačov možno dosiahnuť zbalením správy pred jej odoslaním a rozbalením správy po jej prijatí. Výhodou tohto spôsobu je to, že používateľ nepotrebuje vytvoriť nový typ MPI, stačí použiť tri procedúry:

MPI_Pack_size slúži na výpočet veľkosti bafra, do ktorého bude správa zbalená pred odoslaním.

MPI_Pack slúži na zbalenie správy pred jej odoslaním. Toto zbalenie (packing) znamená, že správa pripravená pre odoslanie sa pretransformuje do nového formátu a uloží do nového bafra s položkami typu `MPI_PACKED`, pričom dlžka správy narastie. V tomto novom formáte sa potom odosiela.

MPI_Unpack slúči na rozbalenie (unpacking) správy po prijatí, čo je opačná akcia, ako pri zbalení.

Je zrejmé, že prenos správ zaťažený zbaľovaním a rozbaľovaním je pomalší ako priamy prenos, keď typ položky odovzdávanej správy, nazývaný tiež typom MPI je parametrom príslušnej procedúry MPI. V štandarde MPI tento typ MPI však nie je priamo typom jazyka (C, FORTRAN). Koncepcia údajových typov MPI umožňuje vývoj prenositeľných aplikácií v heterogénnych sieťach, pretože typ MPI napodobňuje typy jazyka, t.j. prispôsobuje sa ich rozdielnej reprezentácii.

Okrem základných typov (položiek správ) definovaných v štandarde MPI, ktoré sú uvedené v časti 6.5, možno definovať aj nové, t.j. konštruované typy MPI.

6.7.1 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 $type_i$.
Napríklad zobrazenie typu `MPI_INT` je definované takto:

$$Typemap = \{(int, 0)\}$$

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:

$$\begin{aligned} \mathbf{lb}(Typemap) &= min\ \{disp_j\}, \text{ak žiadny typ nie je typom } \texttt{MPI\_LB} \\ \mathbf{ub}(Typemap) &= max\ \{(disp_j + sizeof(type_j))\} + \varepsilon, \\ &\quad \text{ak žiadny typ nie je typom } \texttt{MPI\_UB} \end{aligned}$$

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

Rozsah nového typu je potom definovaný nasledovne:

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

Programátor sa nemusí starať o vnútornú štruktúru zobrazenia typov MPI do detailov, toto zobrazenie sa však vytvára na základe definície nového typu, k čomu slúžia procedúry MPI, uvedené neskôr.

Ak typy $type_i$, z ktorých sa konštruuje nový typ, vyžadujú zaokrúhlenie nahor na hodnotu adresy, ktorá je násobkom nejakej konštanty, potom sa pri definícii zobrazenia nového typu automaticky pridá najmenšia hodnota ε_i, ktorá to zabezpečí pre všetky typy uvedené v zobrazení konštruovaného typu.

Napr. ak $Typemap = \{(double, 0), (char, 8)\}$, potom $\mathbf{lb} = 0$. Ak navyše typ *double* vyžaduje zaokrúhlenie nahor na hodnotu, ktorá je násobkom 8, potom

$$\mathbf{ub} = max\ \{0 + 8, 8 + 1\} + \varepsilon = 9 + \varepsilon = 16$$

a rozsah nového typu je definovaný hodnotou $extent = 16$, pretože bolo potrebné pridať hodnotu $\varepsilon = 7$.

6.7.2 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:

spojitý typ (contiguous) – je definovaný jednou hodnotou dĺžky poľa, žiadnym posunom a jedným údajovým typom, z ktorého je konštruovaný.

vektor s obkrokmi (strided vector) – je definovaný jednou hodnotou dĺžky poľa, 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 byť 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)
```

6.7.3 Konštruktor spojitého typu

Na konštrukciu nového spojitého typu na základe starého typu (t.j. predtým definovaného typu položky, na základe ktorého je nový spojitý typ konštruovaný) slúži konštruktor `MPI_Type_contiguous` s nasledujúcim interfejsom:

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
                        MPI_Datatype *newtype);
```

pričom význam parametrov je takýto:

IN	`count`	počet opakovaní starého typu
IN	`oldtype`	starý typ
OUT	`newtype`	nový typ

Príklad 6.7.1

Predpokladajme, že zobrazenie starého typu $Typemap$ a jeho rozsah $extend$ sú nasledovné:

$$Typemap = \{(double, 0), (char, 8)\};\ extent = 16$$

Potom na základe volania procedúry MPI – konštruktora spojitého typu v tvare

```
MPI_Type_contiguous (3, oldtype, &newtype);
```

je konštruovaný nový spojitý typ so zobrazením

$$\begin{aligned} Typemap \quad = \quad & \{\ (double, 0), (char, 8), (double, 16), (char, 24), \\ & (double, 32), (char, 40)\ \} \end{aligned}$$

6.7.4 Konštruktor vektora s obkrokmi

Na konštrukciu nového typu vektor s obkrokmi slúži konštruktor `MPI_Type_vector` s nasledujúcim interfejsom:

```
int MPI_Type_vector(int count, int blocklength, int stride,
              MPI_Datatype oldtype, MPI_Datatype *newtype);
```

kde význam parametrov je takýto:

IN	`count`	počet blokov
IN	`blocklength`	počet prvkov v každom bloku
IN	`stride`	obkrok – počet prvkov medzi začiatkami dvoch následných blokov
IN	`oldtype`	starý typ
OUT	`newtype`	nový typ

Príklad 6.7.2

Predpokladajme, že zobrazenie starého typu *Typemap* a jeho rozsah *extend* sú nasledovné:

$$Typemap = \{(double, 0), (char, 8)\};\ extent = 16$$

Potom na základe volania konštruktora vektora s obkrokmi v tvare

```
MPI_Type_vector(2,3,4,oldtype, &newtype);
```

vznikne nový typ so zobrazením

$$\begin{aligned} Typemap \quad = \quad & \{\ (double, 0), (char, 8), (double, 16), (char, 24), \\ & (double, 32), (char, 40), (double, 64), (char, 72), \\ & (double, 80), (char, 88), (double, 96), (char, 104)\ \} \end{aligned}$$

Ďalší príklad ukazuje možnosť zobrazenia prvého stĺpca matice rozmeru 3×4 (R=3 riadky a S=4 stĺpce) do vektora s obkrokmi.

Príklad 6.7.3

Nasledujúci fragment programu umožňuje použiť ako parameter procedúr pre odosielanie, resp. prijímanie správ nový typ `newtype` ktorý je vektorom z obkrokmi veľkosti S=4, obsahujúcim tri bloky, každý obsahuje jeden prvok.

```
int R=3, S=4;
MPI_Datatype newtype;
MPI_Type_vector(R, 1, S, MPI_INT, &newtype);
MPI_Type_commit(&newtype);
```

To znamená, že prenosový kanál je priamo napojený na prvý stĺpec matice pri odosielaní aj prijímaní správy, a teda je možné tento prvý stĺpec preniesť selektívne, t.j. bez potreby prenosu ostatných stĺpcov matice.

6.7.5 Konštruktor indexového typu

Nový indexový typ je konštruovaný konštruktorom `MPI_Type_indexed`, ktorého interfejs je v tvare:

```
int MPI_Type_indexed (int count, int blocklengths[],
                int displacements[], MPI_Datatype oldtype,
                MPI_Datatype *newtype)
```

pričom

IN	`count`	je počet blokov
IN	`blocklengths`	pole dĺžok blokov (počty prvkov každého bloku)
IN	`displacements`	pole posunov blokov v násobkoch rozsahu starého typu
IN	`oldtype`	starý typ
OUT	`newtype`	nový typ

Príklad 6.7.4

Nech starý typ má svoje zobrazenie a rozsah definované nasledovne:

$$Typemap = \{(double, 0), (char, 8)\};\ extent = 16$$

Ďalej, nech počty blokov sú `B={3,1}` a ich posuny sú `D={4,0}`. Potom volaním konštruktora `MPI_Type_indexed` v tvare

```
MPI_Type_indexed(2,B,D,oldtype, &newtype);
```

bude konštruovaný nový indexový typ so svojim zobrazením v tvare:

$$\begin{aligned} Typemap \quad = \quad & \{\ (double, 64), (char, 72), (double, 80), (char, 88), \\ & (double, 96), (char, 104), (double, 0), (char, 8)\ \} \end{aligned}$$

6.7.6 Konštruktor štruktúry

Typ štruktúra je najzložitejším novým typom, keďže je konštruovaný z viacerých starých typov. Interfejs konštruktora pre konštrukciu štruktúry je v tvare:

```
int MPI_Type_struct (int count, int blocklengths[],
             MPI_Aint displacements[], MPI_Datatype oldtypes[],
             MPI_Datatype *newtype);
```

kde

IN	`count`	je počet blokov
IN	`blocklengths`	pole dĺžok blokov (počtov prvkov každého bloku)
IN	`displacements`	pole posunov každého bloku v bytoch
IN	`oldtypes`	pole starých typov prvkov v každom bloku
OUT	`newtype`	nový typ

Príklad 6.7.5

Nech pre starý typ TYP (ktorý bol už predtým konštruovaný) platí:

$$Typemap = \{(double, 0), (char, 8)\};\ extent = 16$$

Ďalej vezmime do úvahy dva ďalšie typy: `MPI_FLOAT` a `MPI_CHAR`. Priraďme do poľa `B` dĺžky blokov, do poľa `D` posuny blokov v bytoch a do poľa `T` tri typy nasledujúcim spôsobom:

```
B={2,1,3}; D={0,16,26}; T={MPI_FLOAT,TYP,MPI_CHAR}
```

Volaním konštruktora `MPI_Type_struct` v tvare

```
MPI_Type_struct(3,B,D,T, &newtype);
```

vznikne nový typ – štruktúra, obsahujúca tri bloky, ktorej zobrazenie je v tvare:

$$\begin{aligned} Typemap \quad = \quad & \{\ (float, 0), (float, 4), \\ & (double, 16), (char, 24), \\ & (char, 26), (char, 27), (char, 28)\ \} \end{aligned}$$

6.7.7 Výpočet parametrov konštruktorov

Pred volaním konštruktora je potrebné vypočítať jeho parametre, k čomu slúžia nasledujúce pomocné procedúry MPI.
Procedúra pre výpočet adresy pozície pamäťovej bunky v programe:

```
int MPI_Address(void *location, MPI_Aint *address)
```

ktorej parametre majú tento význam:

IN	`location`	pozícia v pamäti procesu odosielajúceho správu
OUT	`address`	adresa pozície (celé číslo)

a procedúra pre výpočet rozsahu typu:

```
int MPI_Extent(MPI_Datatype datatype, int *size)
```

kde

IN	`datatype`	typ
OUT	`size`	rozsah tohto typu (celé číslo)

Ďalej sú k dispozícii procedúry pre určenie posunov spodnej a hornej hranice typu; `MPI_Type_lb` pre spodnú a `MPI_Type_ub` pre hornú hranicu:

```
int MPI_Type_lb(MPI_Datatype datatype, MPI_Aint *displacement)
```

kde

IN	`datatype`	typ
OUT	`displacement`	posun spodnej hranice od začiatku v bytoch (celé číslo)

```
int MPI_Type_ub(MPI_Datatype datatype,MPI_Aint *displacement)
(integer)
```

kde

IN	`datatype`	typ
OUT	`displacement`	posun spodnej hranice od začiatku v bytoch

Napokon, niekedy možno s výhodou použiť špeciálne značky; `MPI_LB` pre spodnú hranicu a `MPI_UB` pre hornú hranicu typu. Tieto značky sú pseudotypy s nulovým rozsahom, t.j. platí

$$extent(\texttt{MPI\_LB}) = extent(\texttt{MPI\_LB}) = 0$$

Ak tieto značky sú v zobrazení typu definované, potom pre spodnú (**lb**) resp. hornú (**ub**) hranicu platí:

$$\begin{aligned} \mathbf{lb}(Typemap) &= min\ \{\ disp_j \mid type_j = \texttt{MPI\_LB}\ \} \\ \mathbf{ub}(Typemap) &= max\ \{\ disp_j \mid type_j = \texttt{MPI\_UB}\ \} \end{aligned}$$

Nasledujúci príklad ilustruje využitie procedúr pre výpočet parametrov pri konštrukcii štruktúry.

Príklad 6.7.6

```
typedef { double x;
          char y;
          float z[3]; } cell;

struct cell c[200];

MPI_Datatype celltype;
int blocklengths[4] = {1, 1, 3, 1};
MPI_Aint base;
MPI_Aint displacements[4];
MPI_Datatype types[4] = {MPI_DOUBLE, MPI_CHAR,
  MPI_FLOAT, MPI_UB};
MPI_Address(&c[0].x, &displacement[0]);
MPI_Address(&c[0].y, &displacement[1]);
MPI_Address(&c[0].z, &displacement[2]);
MPI_Address(&c[1].x, &displacement[3]);
base = displacement[0];
for (i = 0; i < 4; ++i) displacement[i] -= base;
MPI_Type_struct(4, blocklengths, displacements,
  types, &celltype);
MPI_Type_commit(&celltype);
```

Ak proces chce komunikovať, t.j. používať procedúry pre odosielanie a prijímanie správ, či už medzi dvoma procesmi, alebo v skupine procesov na základe položiek nového typu, musí predtým zaznamenať nový skonštruovaný typ do systému. Ak na základe tohto nového typu je konštruovaný ďalší nový typ, tento ostáva v platnosti aj vtedy, keď predošlý nový typ bol zo systému uvoľnený.

Na nový typ sa možno pozerať aj z hľadiska bafra pre odosielanie alebo prijímanie správ. Nový typ umožňuje selektívny výber časti bafra, alebo položky

bafra na prenosový kanál. To znamená, že komunikácia prebieha iba medzi vybratými časťami bafra odosielajúceho a prijímajúceho procesu a je teda rýchlejšia ako komunikácia na základe zbaľovania správ. Jedinou nevýhodou je to, že niekedy je potrebné definovať väčšie množstvo nových typov.

6.8 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 mať 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 mať 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, apod.

6.8.1 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

Príklad 6.8.1

Predpokladajme, že komunikátor `comm` je tvorený množinou šiestich procesov $\{0, 1, 2, 3, 4, 5\}$ a jednotlivé procesy volajú procedúru `MPI_Comm_split` podľa nasledujúcej tabuľky.

proces	tvar volania
0	`MPI_Comm_split(comm, 0, 0, &acomm)`
1	`MPI_Comm_split(comm, 2, 1, &bcomm)`
2	`MPI_Comm_split(comm, 1, 4, &ccomm)`
3	`MPI_Comm_split(comm, 1, 3, &ccomm)`
4	`MPI_Comm_split(comm, 1, 2, &ccomm)`
5	`MPI_Comm_split(comm, 0, 5, &acomm)`

V dôsledku uvedených volaní bude komunikátor `comm` rozdelený na tri komunikátory `acomm`, `bcomm` a `ccomm`, obsahujúcimi procesy, označené novými poradovými číslami podľa nasledujúcej tabuľky:

nový komunikátor	číslo procesu v novom komunikátore	pôvodné číslo procesu v komunikátore `comm`
acomm	0	0
	1	5
bcomm	0	1
ccomm	0	4
	1	3
	2	2

Pokiaľ novovytvorený intrakomunikátor nie je uvoľnený, procesy v rámci každého z nich môžu komunikovať a procesy medzi rôznymi intrakomunikátormi nekomunikujú.

Komunikátor možno uvoľniť procedúrou `MPI_Comm_free`, ktorj interfejs je v tomto tvare:

```
int MPI_Comm_free(MPI_Comm comm)
```

kde

IN `comm` komunikátor

Okrem uvedených dvoch možností vytvárania komunikátorov priamo z existujúceho komunikátora, je možné extrahovať z existujúceho komunikátora skupinu do premennej špeciálneho typu `MPI_Group` a to procedúrou s interfejsom

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

kde

IN	`comm`	komunikátor
OUT	`group`	extrahovaná skupina procesov z komunikátora

Táto extrakcia znamená oslobodenie skupiny procesov komunikátora od kontextu a uloženie informácie o množine procesov obsiahnutých v komunikátore do premennej `group`.

Na základe množinových operácií, uvedených nižšie, možno vytvoriť novú skupinu procesov a na základe takto vytvorenej novej skupiny procesov konštruovať nový komunikátor.

Naopak, skupinu procesov možno uvoľniť procedúrou MPI s interfejsom v tvare:

```
int MPI_Group_free(MPI_Group *group)
```

kde

INOUT	`group`	uvoľnovaná skupina procesov

V štandarde MPI sú definované interfejsy početných procedúr pre prácu so skupinami.

Napr. procedúra `MPI_Group_translate_ranks` umožňuje zistiť, ktorým poradovým číslam procesov v pôvodnej skupine zodpovedajú čísla procesov v novej skupine.

K inej kategórii patria množinové operácie so skupinami procesov, napr. `MPI_Group_union` pre vytvorenie novej skupiny zjednotením dvoch skupín, `MPI_Group_intersection` pre vytvorenie novej skupiny prienikom dvoch skupín, atď.

Po vytvorení novej skupiny týmito operáciami možno túto skupinu použiť pre vytvorenie nového komunikátora, a to pomocou procedúry `MPI_Comm_create`, ktorej interfejs je v tvare:

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group,
                    MPI_Comm *newcomm)
```

kde

IN	`comm`	komunikátor
IN	`group`	skupina, ktorá je podmnožinou skupiny tvoriacej komunikátor comm
OUT	`newcomm`	nový komunikátor

6.8.2 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

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

Príklad 6.8.2

Predpokladajme,že komunikátory `acomm` a `ccomm` boli vytvorené z komunikátora `com` spôsobom uvedeným v príklade 6.8.1, čiže nové čísla procesov v týchto komunikátoroch zodpovedajú číslam procesov v pôvodnom komunikátore `comm` podľa nasledujúcej tabuľky.

nový komunikátor	číslo procesu v novom komunikátore	pôvodné číslo procesu v komunikátore `comm`
acomm	0	0
	1	5
ccomm	0	4
	1	3
	2	2

Vzhľadom na to, že skupiny procesov v komunikátoroch `acomm` a `ccomm` sú podmnožinami skupiny procesov patriacich komunikátoru `comm`, možno zvoliť komunikátor `comm` za dozerajúci komunikátor.
Ak je potrebné, aby proces 1 obsiahnutý v intrakomunikátore `acomm` komunikoval s procesom 0 v intrakomunikátore `ccomm`, potom obidva procesy v intrakomunikátore `acomm` musia určiť svoj vedúci proces, t.j. proces 1 a musia vedieť, že proces 0 v intrakomunikátore `ccomm` má číslo 4 v dozorujúcom komunikátore `com`. Preto obidva procesy musia vykonať volanie

```
MPI_Intercomm_create(acomm, 1, comm, 4, 10, &intercomm)
```

napríklad so značkou 10.
Na druhej strane, všetky tri procesy v intrakomunikátore `ccomm` musia určiť svoj vedúci proces 0 a musia vedieť, že proces 1 v intrakomunikátore `acomm` má číslo 5 v dozorujúcom komunikátore `comm`. Preto tieto tri procesy musia vykonať volanie v tvare:

```
MPI_Intercomm_create(acomm, 0, comm, 5, 10, &intercomm)
```

s tou istou značkou.
Vtedy vznikne nový interkomunikátor `intercomm`, ktorý umožní komunikáciu medzi intrakomunikátormi `acomm` a `ccom` prostredníctvom dvoch procesov, tzv. vedúcich procesov, z ktorých jeden patrí jednému intrakomunikátoru a druhý druhému intrakomunikátoru.

Vytvorenie intrakomunikátora z interkomunikátora je operáciou užitočnou vtedy, ak je potrebné namiesto obmedzenej komunikácie medzi dvoma vedúcimi procesmi použiť skupinovú komunikáciu medzi všetkými procesmi patriacimi do dvoch intrakomunikátorov spojených interkomunikátorom.
Interfejs tejto procedúry je v tvare:

```
int MPI_Intercomm_merge(MPI_Comm intercomm,
  int high, MPI_Comm *newintracomm)
```

kde

IN	`intercomm`	interkomunikátor
IN	`high`	logická hodnota
OUT	`newintracomm`	nový intrakomunikátor

Logická hodnota určuje poradie, v ktorom sa zaradia skupiny procesov dvoch intrakomunikátorov v interkomunikátore do nového intrakomunikátora.

6.8.3 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

Ukážeme spôsob priradenia procesov pre prípad mriežkovej topológie.

Procedúra `MPI_Dims_create` umožňuje pre daný počet procesov a rozmerov (dimenzií) mriežky určiť veľkosť jednotlivých dimenzií a tým aj určiť základ pre rozloženie procesov v mriežke.

Interfejs tejto procedúry je v tvare:

```
int MPI_Dims_create(int nnodes, int ndims, int *dims)
```

kde

IN	`nnodes`	celkový počet uzlov mriežky (celé číslo)
IN	`ndims`	počet rozmerov mriežky (celé číslo)
INOUT	`dims`	pole veľkosti `ndims` obsahujúce počet uzlov v každom rozmere

Musí teda platiť

$$nnodes = \prod_{i, dims[i] \neq 0} dims[i]$$

Príklad 6.8.3

Nasledujúci fragment programu zistí počet procesov v komunikátore `MPI_COMM_WORLD` a pre dvojrozmernú mriežkovú topológiu určí počet uzlov v každom rozmere mriežky.

```
int dims[2]={0,0};  int size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Dims_create(size, 2, dims);
```

Za predpokladu, že zistený počet procesov je `size=6`, výsledná hodnota poľa `dims` je `dims[2]={3,2}`.

Počet uzlov pre riadky a stĺpce závisí od vstupnej hodnoty poľa `dims`. Ak všetky položky sú nulové, algoritmus rozloženia uzlov odvodí sám rozloženie tak, aby medzi uzlami mriežky boli najkratšie cesty.

Ak položka `dims` je nenulová, algoritmus rozloženia uzlov ju zachováva, a ak to nie je možné, vedie to k chybe.

Niekoľko príkladov závislosti výstupnej hodnoty `dims` od vstupnej hodnoty `dims` a parametrov volania `MPI_Dims_create` je uvedených v nasledujúcej tabuľke.

vstupná hodnota dims	volanie procedúry	výstupná hodnota dims
{0,0}	`MPI_Dims_create`(6, 2, dims)	{3,2}
{0,0}	`MPI_Dims_create`(7, 2, dims)	{7,1}
{0,3,0}	`MPI_Dims_create`(6, 3, dims)	{2,3,1}
{0,3,0}	`MPI_Dims_create`(7, 3, dims)	chyba

Ak je počet uzlov mriežky určený, na vytvorenie komunikátora s topológiou možno použiť konštruktor mriežkovej topológie, ktorý má nasledujúci interfejs:

```
int MPI_Cart_create(MPI Comm comm old, int ndims,
              int *dims,   int *periods, int reorder,
              MPI Comm *comm_cart)
```

a ktorej parametre majú tento význam:

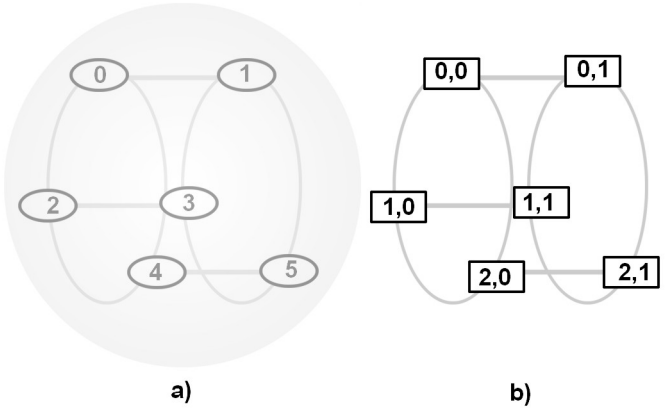
IN	`comm`	vstupný komunikátor
IN	`ndims`	počet rozmerov mriežky (integer)
IN	`dims`	pole veľkosti `ndims` obsahujúce počet uzlov v každom rozmere
IN	`periods`	pole veľkosti `ndims` určujúce či je mriežka cyklicky uzavretá (1) alebo nie (0) pre každý rozmer
IN	`reorder`	usporiadanie procesov nezmenené (0) alebo v opačnom poradí (1)
OUT	`comm_cart`	komunikátor s novou mriežkovou topológiou

Príklad 6.8.4

Nasledujúci fragment programu vytvorí mriežku cyklickú v prvom rozmere (cyklické stĺpce) a acyklickú v druhom rozmere (acyklické riadky) viď obr. 6.3, pričom procesy sú rozložené prioritne do riadkov (druhý rozmer) a budú usporiadané v rovnakom poradí poradových čísel, ako v pôvodnom komunikátore.

```
/* size=6 */
int dims[2]={3,2};
int periods[2] = {1, 0};
MPI_Comm comm_cart;

MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm_cart);
```



Obr. 6.3: Komunikátor s topológiou a) a poloha procesov v mriežke b)

Pre poradové číslo každého procesu, ktorý patrí komunikátoru s mriežkovou topológiou, možno zistiť súradnice uzla, v ktorom sa nachádza proces, a to pomocou procedúry `MPI_Cart_coords`, ktorá má tento interfejs:

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims,
                    int *coords)
```

kde

IN	`comm`	je komunikátor s mriežkovou topológiou
IN	`rank`	poradové číslo procesu v komunikátore comm
IN	`maxdims`	počet rozmerov mriežky (celé číslo)
OUT	`coords`	integer array (of size ndims) containing the cartesian coordinates of specified process (integer)

Príklad 6.8.5

Predpokladajme `size =6; dims[2]={3,2};`. Potom fragment programu

```
int mycoords[2];

MPI_Cart_coords(comm_cart, 4, 2, mycoords);
```

zistí `int mycoords[2] = {2,0};`, čiže súradnice procesu 4, podľa obr.6.3.

Operácia posunu v kartézskych súradniciach umožňuje ľubovoľnému procesu v mriežke zistiť zdrojový a cieľový proces v ľubovoľnom smere s ľubovoľným krokom a následne túto informáciu využiť pre zložitejšie presuny správ. Procedúra posunu v kartézskych súradniciach má nasledujúci interfejs:

```
int MPI_Cart_shift(MPI_Comm comm, int direction,
  int disp, int *rank_source, int *rank_dest)
```

kde

IN	`comm`	komunikátor s mriežkovou topológiou
IN	`direction`	smer posunu (súradnica – celé číslo)
IN	`disp`	posun (> 0 : nahor, < 0 : nadol) (celé číslo)
OUT	`rank_source`	poradové číslo zdrojového procesu
OUT	`rank_dest`	poradové číslo cieľového procesu

Príklad 6.8.6

Definujme napríklad smer posunu ako posun v stĺpci, t.j. `direction=0`, a `disp=1`, t.j. posun nahor o jeden krok.

```
int rank_src, rank_dst;
MPI_Cart_shift(comm_cart,0,1,&rank_src,&rank_dst);
MPI_Sendrecv_replace (A, 1, MPI_REAL,rank_dst, 0, rank_src, 0,
                      comm_cart, &status);
```

Ak aktuálny proces má poradové číslo 3 podľa obr.6.3 a), potom výsledkom volania `MPI_Cart_shift` sú poradové čísla dvoch procesov `rank_src=5` a `rank_dst=1`, ktoré sú využité pri volaní procedúry `MPI_Sendrecv_replace`, ktorá odovzdá najprv reálne číslo z bafra A aktuálneho procesu (3) procesu 1 a bezprostredne nato prijme do bafra A reálne číslo od procesu 5.

V štandarde MPI sú definované mnohé ďalšie užitočné operácie pre prácu s komunikátormi, napr. pre vytvorenie nového komunikátora na základe podštruktúry komunikátora s topológiou, ako aj rozsiahla množina procedúr MPI pre prácu s grafovou topológiou, ktorá je všeobecnejšia ako mriežková. Prístupy sú však podobné ako pre mriežkovú topológiu, a ciele sú totožné – organizácia vzájomnej komunikácie v priestore procesov tak, aby čo najlepšie vystihovala jednak implementované algoritmy, a jednak použitú architektúru.

Záver

Pri zjednodušenom pohľade na riešenie problémov, vedúcich k výkonným paralelným výpočtom by sa zdalo, že znalosť matematických modelov, paralelných algoritmov a metód plánovania a implementácie paralelných výpočtov sú dostatočnými východiskami pre riešenie aplikačných problémov v oblasti modelovania zložitých systémov s cieľom predpovede ich správania a vývoja.

Pravda je však iba to, že najmenší problém spôsobuje implementácia výkonných paralelných výpočtov. Napr. pri použití modelu programovania na báze odovzdávania správ je na adrese `http://www.lam-mpi.org/` k dispozícii posledná verzia implementácie LAM/MPI a úplnú definíciu štandardu MPI možno nájsť zasa na adrese `http://www.mpi-forum.org/`. Okrem systému LAM existuje mnoho ďalších implementácii MPI, napr. Alpha Data MPI, Appleseed, ChaMPIon/Pro, CRI/EPCC, DISI, HP MPI. Tieto implementácie sa líšia viac menej iba druhom paralelného počítača, pre ktorý sú určené, a tiež tým, že nie všetky sú voľne dostupné. V súčasnosti však existuje množstvo ďalších implementačných prostriedkov pre výkonné paralelné výpočty, spomeňme len jazyky Compositional C++, Fortran M, High Performance Fortran, ako aj prostriedky, ako sú Paragraph, Upshot, Pablo, AIMS, atď.

Väčší problém nastáva pri posúdení vhodnosti použitia paralelnej počítačovej architektúry vo vzťahu k triede riešených problémov.

Pre ilustráciu, ak máme konkrétny problém, ktorý spočíva vo výpočte na podmatici veľkosti $n \times n$ každým procesom a na výmene údajov veľkosti n (dĺžka hrany podmatice) medzi dvoma procesmi, situácia je jednoduchá. Ak chceme výpočet a výmenu údajov v čase prekryť, potom vieme presne, že procesor musí byť n krát rýchlejší ako komunikačný kanál medzi procesormi (pre jednoduchosť nech čas spracovania jedného údaja je rovnaký ako čas jeho prenosu), pretože $(n \times n)/n = n$. A naopak, ak je daná rýchlosť výpočtu v na procesore počtom spracovaných údajov za sekundu a rýchlosť prenosu p počtom prenesených údajov za sekundu, potom je zrejmé, že maticu má zmysel rozdeliť na matice $n \times n$, kde $n = v/p$.

Vo všeobecnosti však neexistuje paralelná počítačová architektúra, rovnako vhodná pre riešenie všetkých paralelných problémov. Ak aj prihliadneme na súčasnú modulárnu výstavbu superpočítaov a možnosti ich rekonfigurácie, je potrebné dôkladne zvážiť triedu problémov, pre ktoré bude paralelný počítač, resp. superpočítač použitý. Z tohto pohľadu je iba orientačným pravidlom, že viac ako rýchlosť procesorov je dôležitá rýchlosť prístupu k pamäti a veľkosť pamäti, najmä však vysoká rýchlosť komunikácie.

Podobná situácia vzniká pri voľbe vhodných paralelných algoritmov. V súčasnosti existuje obrovské množstvo špecializovaných paralelných algoritmov na riešenie konkrétnych problémov. Treba si však všimnúť nielen podstatu algoritmu, ale aj parametre prostredia (architektúry), v ktorom bol alebo bude využitý. V opačnom prípade nie je možné dosiahnuť požadovaný výkon výpočtu, vzhľadom na algoritmické náklady.

A napokon, kľúčovým problémom je vzťah systémov a matematických modelov. Modelovať systém výkonným paralelným výpočtom, riešiacim napr. parciálnu diferenciálnu rovnicu niektorou z numerických metód na paralelnom počítači má zmysel vtedy, ak tento model vystihuje podstatné vlastnosti systému. Iba vtedy možno očakávať, že model sa bude chovať rovnako ako systém. V tejto súvislosti sa výkonné paralelné výpočty uplatňujú mimoriadne úspešne najmä pre modely technických systémov, pretože prostredie modelovania – paralelné počítačové architektúry – zvyšovaním svojej celkovej veľkosti pamäti a komunikačnej rýchlosti medzi procesormi v súčasnosti umožňujú modelovať tieto systémy do dostatočnej hĺbky detailov. Menšia úspešnosť je u fyzikálnych systémov, napríklad pri predpovedi počasia, pretože ich modely sú pravdepodobnostné a v čase divergujúce. Modelovanie biologických systémov sa dotýka zásadnej otázky, ktorej odpoveď iba tušíme, že totiž systém, ktorý nie je analogický inému systému, ho nielenže nemôže poznať, ale ani primerane napodobniť. Modelovaním biologických systémov sa dostávame do situácie, keď pomocou neživého programu vykonávanom na neživom počítači chceme vystihnúť niektoré stránky podstaty živého organizmu.

I keď otázka, akými smermi sa bude uberať vývoj v oblasti metodológie paralelného programovania, nie je v súčasnosti jednoznačne zodpovedaná, o praktickej užitočnosti paralelného programovania niet žiadnych pochybností. Neustále rastúce environmentálne problémy nás o tom každodenne presviedčajú, keďže vyvolávajú potrebu skúmania zložitých systémov, ktoré je bez výkonných paralelných výpočtov nemysliteľné.

Literatúra

[1] ABRAMSON, D., EGAN, G.: The RMIT Data Flow Computer: A Hybrid Architecture. The Computer Journal, Vol.33, No.3, 1990, 230–240

[2] ACKERMAN, W.B.: Data Flow Languages. Computer, Vol.15, No.2., Feb. 1982, 15–25

[3] ADVE, V., JIN, G., CRUMMEY, J. M., YI, Q.: High performance Fortran compilation techniques for parallelizing scientific codes, Proceedings of the 1998 ACM/IEEE conference on Supercomputing, San Jose, CA, 1998, 1–23

[4] AGERWALA, T., ARVIND: Data Flow Systems. Computer, Vol.15, No.2, Feb. 1982, 10–13

[5] AHO, A.V., HOPCROFT, J.E., ULLMAN, J.D.: Data Structures and Algorithms. Addison–Wesley, 1985

[6] Al-TAWIL, K., MORITZ C. A.: Performance Modeling and Evaluation of MPI, Journal of Parallel and Distributed Computing, Volume 61, Issue 2, February 2001, 202–223

[7] AMAMIYA, M., HAGESAWA, R., ONO, S: VALID:A High–Level Functional Language for Dataflow Machine. Review of Electrical Communication Laboratories, Vol.32, No.5, 1984, 793–802,

[8] ARVIND, GOSTELOW, K.P.: The U–Interpreter. Computer, Vol.15, No.2., Feb. 1982, 42–49

[9] ARVIND, KATHAIL, V.: A Multiple Processor Data Flow Machine that Supports Generalized Procedures. In: The 8–th Ann. Symp. on Computer Architecture, IEEE, May 1981, 291–302

[10] AXFORD, T.: Concurrent Programming, J.Willey and sons, 1988

[11] BACKUS, J.: Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. Communications of the ACM, Vol.21, No.8, Aug. 1978, 613–641

[12] BERGSTRA, J.A., HEERING, .J., KLINT. P.: Algebraic Specification. Addison Wesley, 1989

[13] BOHM, A.P.W., SARGEANT, J.: Code Optimization for Tagged – Token Dataflow Machine. IEEE Transaction on Computers, Vol.38, No.1, Jan. 1989, 4–14

[14] BOSILCA, G. et al.: MPICH-V: toward a scalable fault tolerant MPI for volatile nodes, November 2002, Proceedings of the 2002 ACM/IEEE conference on Supercomputing, 15–30

[15] BRIGHTWELL, R., PLIMPTON, S.: Scalability and Performance of Two Large Linux Clusters, Journal of Parallel and Distributed Computing, Volume 61, Issue 11, November 2001, 1546–1569

[16] BROBERG, M., LUNDBERG, L., GRAHN, H.: Performance Optimization Using Extended Critical Path Analysis in Multithreaded Programs on Multiprocessors, Journal of Parallel and Distributed Computing, Volume 61, Issue 1, January 2001, 115–136

[17] COMTE, D., HIFDI, N., SYRE, J.C.: The Data Driven LAU Multiprocessor System: Results and perspectives, Proc. IFIP congress,1980

[18] DAVIS, A.L., KELLER, R.M.: Data Flow Program Graphs. Computer, Vol.15, No.2., Feb.1982, 26–41

[19] DENNIS, J.B.: Models of Data Flow Computation. In: Control Flow and Data Flow: Concepts of Distributed Programming, Munich, Germany, 1984

[20] DENNIS, J.B.: Functional Programming for Data Flow Computation. In: Control Flow and Data Flow: Concepts of Distributed Programming, Munich, Germany, 1984

[21] DENNIS, J.B.: VIM: An Experimental Computer System to Support General Functional Programming. In: Control Flow and Data Flow: Concepts of Distributed Programming, Munich, Germany, 1984

[22] EVANS, D.: Parallel Processing Systems. (translation) Moskva, Mir, 1985

[23] GAJSKI, D.D., PADUA, D.A., KUCK, D.J., KUHN, R.A.: A Second Opinion on Data Flow Machines and Languages. Computer, Vol.15, No.2, Feb. 1982, 58–73

[24] GAUDIOT, J.–L.: Structure Handling in Data–Flow Systems. IEEE Transactions on Computers, Vol.c–35, No.6, Jun 1986, 489–502

[25] GEISLER, J., TAYLOR, V.: Performance Coupling: Case Studies for Improving the Performance of Scientific Applications, Journal of Parallel and Distributed Computing, Volume 62, Issue 8, August 2002, 1227–1247

[26] GENNER, R.E., GUSTAFSON, J.L., MONTRY, R.E.: Development and analysis of scientific applications programs on a 1024–processor hypercube, Sandia National Laboratories, Albuquerque, Feb. 1988, 88–0317

[27] GIBBONS, A., RYTTER, W.: Efficient Parallel Algorithms. Cambridge University Press, 1990

[28] GOLDBERG, B.F.: Multiprocessor execution of functional programs. YALEU/DCS/RR–618, Dept. of Comp. Science, Yale University, Apr. 1988

[29] GROPP, W., LUSK, E., SKJELLUM, A.: Using MPI - 2nd Edition, Portable Parallel Programming with the Message Passing Interface, November 1999 ISBN 0-262-57132-3, 350 pp.

[30] GURD, J.R., KIRKHAM, C.C, WATSON, I.: The Manchester Prototype Dataflow Computer. Communications of the ACM, Vol. 28, No. 1, Jan 1985, 34–52

[31] HARRISON, P.G., REEVE, M.: The parallel graph reduction machine ALICE. In: Graph Reduction: Proc. of a Workshop, Santa Fe, (ed. Fasel, J.H. and Keller, R.M.) , LNCS 279, Springer Verlag, 1987, 181–202

[32] HARTMANN, A.C.: A Concurrent Pascal Compiler for Minicomputers. Springer–Verlag Berlin, 1976

[33] HASSEN, S. B., BAL, H. E., JACOBS, C. J. H.: A task– and data–parallel programming language based on shared objects, November 1998, ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 20 Issue 6, 78–91

[34] HAYNES, L.S., LAU, R.L., SIEWIOREK, D.P., MIZELL, D.W.: A Survey of Highly Parallel Computing. Computer, Vol.15, No.1, Jan. 1982, 9–24

[35] HE, Y., DING, Ch. H. Q.: MPI and OpenMP paradigms on cluster of SMP architectures: the vacancy tracking algorithm for multi-dimensional array transposition, November 2002, Proceedings of the 2002 ACM/IEEE conference on Supercomputing, 79–99

[36] HIRAKI, K., SHIMADA, T., NISHIDA, K.: A Hardware Design of the SIGMA–1, a Data Flow Computer for Scientific Computations. In: Proc. Int. Conf. on Parallel Processing, IEEE, 1984, 594–531

[37] HUDAK, P.: Distributed Graph Marking. Research report, Comp. Science Dept. Yale, Jan.1983

[38] HUDAK, P.: Distributed task and memory management. In Proceedings of the ACM Symposium on Principles of Distributed Computing, Aug.1983, 277–289

[39] HWANG, G.-H., LEE, J.K., JU, R. Dz-Ch.: Array Operation Synthesis to Optimize HPF Programs on Distributed Memory Machines, Journal of Parallel and Distributed Computing, Volume 61, Issue 4, April 2001, 467–500

[40] IANNUCCI, R.A.: A Dataflow/von Neumann hybrid architecture. TR 418, Lab for Computer Science, MIT, Nov. 1988

[41] JELŠINA, M.: Architektúry počítačových systémov: princípy, štruktúrna organizácia, funkcia. monografia, elfa Košice, 2002, 470pp.

[42] JELŠINA M., KOLLÁR J.: The Dataflow Implementation Environment for Functional Languages. PROC. of the Japan – Central Europe Joint Workshop on Advanced Computing in Engineering, September 26–29, 1994, Pultusk, Poland, 419–424,

[43] JOHNSSON, T.: The eta–G–machine – an abstract machine for parallel graph reduction. Dept. of Computer Science, Chalmers University, Goteborg, Sweden, Sep. 1988

[44] JONSSON, B.: Compositional specification and verification of distributed systems, March 1994, ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 16 Issue 2, 56–67

[45] KARWANDE, A., YUAN, X., LOWENTHAL, D. K.: Checkpointing and communication: CC–MPI: a compiled communication capable MPI prototype for ethernet switched clusters, June 2003, Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming, 76–92

[46] KISHI, M., YASUHARA, H., KAWAMURA, Y.: DDDP: A Distributed Data Driven Processor. In: Proc. of the 10–th Int. Symp. on Computer Architecture, Jun. 1983, 236–242

[47] KNUTH, D.: The Art of Computer Programming. Vol.I, Addison Wesley, 1976

[48] KODAMA, Y, SAKAI, S., YAMAGUCHI, Y.: Load Balancing by Function Distribution on EM–4 Prototype. Supercomputing, 1991

[49] KOLLÁR J.: The Structured Data Handling at High Performance Specialized Computer System, Computers and Artificial Intelligence, Vol.13, 1994, No.4, 321–338

[50] KOLLÁR J.: Implementation of Functional Language at the Dataflow Computer System. Proc. on FEI'25 Conference on Electronic Computers and Informatics, September 22–23, 1994, Košice, Slovakia, 139–145

[51] KOLLÁR J.: Dataflow Implementation of Abstract Types. Proc. on FEI'25 Conference on Electronic Computers and Informatics, September 22–23, 1994, Košice, Slovakia, 184–189

[52] KOLLÁR J.: Paralelné programovanie. KPI FEI TU Košice, ISBN 80-88786-14-2, tlač. elfa s.r.o, 1999

[53] KOLLÁR J.: Funkcionálny popis paralelného algoritmu analýzy anonymných signálov, Zborník konferencie "Nové smery v spracovaní signálov III, Lipt. Mikuláš, 29-31.5.1996, 231-234

[54] KUNG, H.T., WONG, S.W.: An Efficient Parallel Garbage Collection Systems, and its Correctness Proof. Dept. of Comp. Science, Carnegie–Mellon Univ., Sep.1977

[55] LANG, J., STEWART, D. B.: A study of the applicability of existing exception-handling techniques to component-based real-time software technology, March 1998, ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 20 Issue 2, 34–45

[56] LI K.: Scalable Parallel Matrix Multiplication on Distributed Memory Parallel Computers, Journal of Parallel and Distributed Computing, Volume 61, Issue 12, December 2001, 1709–1731

[57] MAHESWARAN, M. et al.: Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems, Journal of Parallel and Distributed Computing, Volume 59, Issue 2, November 1999, 107–131

[58] McBURNEY, D., SLEEP, M.R.: Transputer – based experiments with the Zapp architecture. In: Proc. PARLE Conference I, LNCS 258, Springer Verlag, 1987, 247–259

[59] NIKHIL, R.S., ARVIND: Can dataflow subsome von Neumann computing. CSG Memo 292, Lab for Computer Science, MIT, Nov. 1988

[60] O'LEARY, D.P., STEWART, G.W.: Data–Flow Algorithms for Parallel Matrix Computation. Communication of the ACM, Vol.28, No.8, Aug. 1985, 840–853

[61] PAEK, Y., HOEFLINGER, J., PADUA, D.: Efficient and precise array access analysis, January 2002, ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 24,Issue 1, 98–132

[62] PATNAIK, L.M., GOVINDARAJAN, R., RAMADOSS, N.S.: Design and Performance Evaluation of EXMAN: EXtended MANchester Data Flow Computer. IEEE Transactions on Computers, Vol.c–35, No.3, Mar. 1986, 229–244

[63] PETERSON, J.L.: Petri Net Theory and Modelling the Systems. Prentice–Hall, Inc. Englewood Cliffs, 1981

[64] PEYTON JONES, S.L.: Parallel Implementations of Functional Programming Languages. The Computer Journal, Vol.32, No.2, 1989, 175–186

[65] RUGIERO, C.A., SARGEANT, J.: Control of parallelism in the Manchester Dataflow Machine. In: Proc. IFIP Conference on Functional Programming Languages and Computer Architecture, Portland (ed. G. Khan), LNCS 274, Springer Verlag, 1987, 1–15

[66] SCHLICHTING, R.D., SCHNEIDER, F. B.: Using message passing for distributed programming: proof rules and disciplines, July 1984, ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 6 Issue 3, 10–22

[67] SHIMADA, T., HIRAKI, K., NISHIDA, K., SEKIGUCHI, S.: Evaluation of a Prototype Dataflow Processor of the SIGMA–1 for Scientific Computations. In: Proc. 13–th Ann. Int. Symp. on Computer Architecture, 1986, 226–234

[68] SNIR, M., et al.: MPI: The Complete Reference (Vol. 1) - 2nd Edition Volume 1 - The MPI Core September 1998, ISBN 0-262-69215-5, 450 pp.

[69] SRINI, V.P.: An Architectural Comparision of Dataflow Systems. Computer, Vol.19, No.3, Mar. 1986, 68–88

[70] STEELE, G.L.: Multiprocessing compactifying garbage collection. Communications of the ACM, Vol.18, No.9, 1975, 495–508

[71] TANG, H., SHEN, K., YANG, T.: Program transformation and runtime support for threaded MPI execution on shared-memory machines, July 2000, ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 22 Issue 4, 23–34

[72] THAKUR, R., GROPP, W., LUSK,E.: A case for using MPI's derived datatypes to improve I/O performance, November 1998, Proceedings of the 1998 ACM/IEEE conference on Supercomputing, 67–75

[73] TRÄFF, J. L.: Implementing the MPI process topology mechanism, November 2002, Proceedings of the 2002 ACM/IEEE conference on Supercomputing, 35–55

[74] WATSON, I., GURD, J.R.: A Practical Data Flow Computer. Computer, Vol.15, No.2, Feb. 1982, 51–57

[75] WATSON, I., SARGEANT, J., WATSON, P., WOODS, V.: Flagship computational models and machine architecture. ICL Technical Journal 5, 1987, 555–574

[76] WEISSMAN, J.B.: Predicting the Cost and Benefit of Adapting Data Parallel Applications in Clusters, Journal of Parallel and Distributed Computing, Volume 62, Issue 8, August 2002, 1248–1271

[77] WIRTH, N.: MODULA–2. Institut für Informatik, ETH, Zürich, Mar. 1980

[78] WISE, D.: Design for a multiprocessing heap with on–board reference–counting. In: Functional Programming and Computer Architecture, Nancy, Jounnaud (editor)., LNCS201, Springer Verlag, 1985, 289–304

[79] YAMAGUCHI, Y., SAKAI, S., HIRAKI, K., KODAMA, YUBA, T.: An Architectural Design of a Highly Parallel Dataflow Machine. Information Processing 89, by edit. Ritter, G.X., North Holland, 1989, 1155–1160

[80] YAMAGUCHI, Y., SAKAI, S., HIRAKI, K., KODAMA, YUBA, T.: An Architecture of a Dataflow Single Chip Processor. In: Proc. 16–th Ann. Symp. on Computer Architecture, 1989

[81] YAMAGUCHI, Y., TODA, K., YUBA, T.: A Performance Evaluation of a LISP based Data–driven Machine (EM–3). In: Proc. 10–th Ann. Symp. on Computer Architecture, 1983, 363–366

[82] YANG, T., FU C.: Space/time-efficient scheduling and execution of parallel irregular computations, November 1998, ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 20, Issue 6, 23–43