Virginia Niculescu

CALCUL PARALEL

Proiectare şi dezvoltare formală a programelor paralele

2005

blanc

Familiei mele

Cuprins

Prefaţă

În timp ce multe probleme de mare interes practic cer tot mai multă putere de calcul, viteza componentelor calculatoarelor se apropie de limitele posibile. Este în general acceptat şi chiar dovedit faptul că aceste probleme ce necesită calcul intensiv nu vor putea fi rezolvate prin creşterea continuă a performanţelor calculatoarelor individuale, ci singura soluţie viabilă este folosirea calculului paralel.

Maşinile paralele, constând din mii de procesoare, oferă o putere de calcul foarte mare pentru aplicaţii complexe. Comunitatea potenţialilor utilizatori creşte foarte puternic şi datorită reţelelor globale, care aduc hard, soft şi expertize din surse dispersate geografic. Totuşi, calculul paralel nu a devenit deocamdată, o cale de rezolvare mai rapidă a problemelor, cu o foarte largă răspândire.

În acest moment programarea paralelă este proiectată, în general, pentru tipuri speciale de arhitecturi şi de aceea mutarea unui program de pe o arhitectură pe alta, necesită rescrierea lui aproape în întregime. Singura cale de a depăşi această problemă şi de a folosi paralelismul pe scară largă, este de a distruge această conexiune strânsă dintre soft şi hard.

Scopul principal al paralelismului este performanţa, dar aceasta este în acelaşi timp şi sursa principală a dificultăţilor legate de paralelism. Pentru a proiecta o soluţie paralelă eficientă, programatorul trebuie să descompună problema într-o colecţie de procese care se pot executa simultan, să mapeze procesele pe procesoarele disponibile, să sincronizeze procesele, să organizeze comunicaţiile, etc. Pentru acestea s-au dezvoltat numeroase familii particulare de algoritmi, limbaje şi tehnici de implementare, pentru diferite tipuri de arhitecturi.

Criza programării secvenţiale a evidenţiat necesitatea abstractizării şi ascunderii detaliilor nivelului de jos de programare. În programarea paralelă, necesitatea abstractizării este şi mai acută, datorită complexităţii programelor paralele. Dacă programatorii de aplicaţii, consideră abstractizarea foarte necesară, totuşi implementatorii programelor paralele sunt de părere că aceasta intră în conflict cu performanţa. Reconcilierea dintre abstractizare şi performanţă, pare a fi calea prin care programarea paralelă s-ar putea impune mai mult în viitor.

În procesul de dezvoltare a unui program se pot identifica trei etape:

Specificarea. Construirea formală a unei descrieri a problemei care trebuie să fie rezolvată. Această descriere exprimă esenţa problemei. Specificaţia trebuie să ajute la demonstrarea formală a corectitudinii programului rezultat în final.

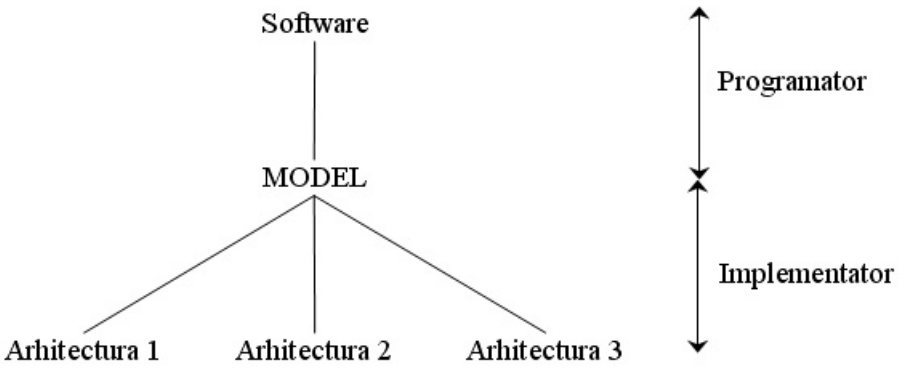

Figura 1: Rolul modelului în calculul paralel

Proiectarea. Sarcina de a crea un "program", care să satisfacă specificaţia problemei şi care să poate fi implementat eficient pe o arhitectură ţintă.

Implementarea. Maparea programului pe resursele de calcul disponibile pentru execuţia sa. Aceasta trebuie să fie făcută folosind instrumente şi tehnici, care au fost verificate formal.

Proiectarea este o etapă fundamentală nu doar pentru aplicaţiile de dimensiuni mari, dar şi în cazul algoritmilor. Paradigme precum programarea structurată, proiectarea "top-down" sau "bottom-up" sunt bine cunoscute. La nivel de algoritm, pentru programarea secvenţială, sunt de asemenea bine cunoscute tehnici de programare precum: greedy, backtraking, branch-and-bound, sau programarea dinamică. Ele direcţionează construcţia programelor, permit stăpânirea complexităţii oferind un cadru ştiinţific şi matematic şi, de asemenea, permit realizarea unor bune documentaţii.

Pentru programarea paralelă, faza de proiectare este poate chiar mai importantă decât în cazul programării secvenţiale, datorită gradului ridicat al complexităţii programelor paralele. Putem analiza, şi în acest caz, diferite paradigme consacrate, precum şi tehnici de programare care ghidează construcţia algoritmilor paraleli.

În programarea secvenţială, faza de implementare este realizată de obicei de un compilator capabil să producă cod optimizat pentru o arhitectură ţintă. Asemenea compilatoare sunt posibile, deoarece majoritatea arhitecturilor secvenţiale sunt similare la nivele înalte de abstractizare şi diferenţele de la nivelele de jos pot fi rezolvate de către compilator. În plus, datorită arhitecturilor similare, limbajele de nivel înalt pot fi proiectate, fără a sacrifica strategiile de compilare eficiente. Aceasta permite programatorilor de programe secvenţiale, luxul de a se concentra asupra corectitudinii şi complexităţii abstracte a programelor, lăsând compilatorului sarcina de a le implementa eficient şi corect.

Arhitecturile paralele sunt foarte diferite unele de altele, chiar şi la nivele înalte de abstractizare. O posibilă abordare este de a lăsa limbajul să reflecte particularităţile arhitecturii – dar aceasta nu este totuşi, o soluţie prea bună. O altă abordare ar fi de a face abstracţie de consideraţiile arhitecturale şi de a construi un limbaj de programare

bazat pe noţiunile abstracte ale paralelismului. Aceasta permite programatorului să se concentreze asupra a *ceea ce trebuie făcut*, mai degrabă decât asupra *cum se poate face pe o anumită arhitectură*.

În concluzie, este importantă rezolvarea conflictului dintre soft şi hard în programarea paralelă prin decuplarea semanticii programelor de potenţialele lor implementări. Aceasta se poate face printr-un *model de calcul paralel*. Un model furnizează programatorului o singură maşină paralelă abstractă, perminţând diferite implementări pe arhitecturi concrete, diferite (Figura 1).

Un program paralel poate implica multe procese cu interacţiuni complexe. Este cunoscut faptul că gradul înalt de complexitate al unui program paralel face extrem de dificilă dezvoltarea lui. Un model de proiectare a programelor paralele trebuie să poată pune la dispoziţie mecanisme prin care să se poată stăpâni complexitatea şi care să permită *verificarea formală* a corectitudinii programelor paralele. Aceasta conduce la necesitatea folosirii metodelor formale, adică folosirea de instrumente şi tehnici bazate pe modele matematice de calcul.

În general, pentru dezvoltarea algoritmilor paraleli s-a folosit mai degrabă intuiţia decât *metode formale de derivare*. Folosirea unor astfel de metode, poate uşura mult procesul de derivare a algoritmilor paraleli, asigurându-se în acelaşi timp şi corectitudinea lor. Derivarea corectă este esenţială, datorită depanării greoaie a programelor paralele. Intuiţia nu poate fi întotdeauna folosită cu succes, iar folosirea unei metode formale poate conduce la obţinerea unor variante eficiente, chiar dacă nu atât de evidente.

Există mai multe modele pentru calcului paralel (care vor fi analizate în Capitolul 9), fiecare cu propria metodă de derivare a algoritmilor, mai riguroasă sau mai puţin riguroasă. Modelele se diferenţiază prin gradul lor de abstractizare, şi prin urmare, şi metodele de derivare a algoritmilor pot fi mai generale sau particularizate pentru anumite variante de programare paralelă.

Cartea îşi propune să prezinte atât tehnici de proiectare cât şi metodologii de dezvoltare formală a programelor paralele şi este dedicată atât studenţilor de la secţiile de informatică şi calculatoare, cât şi tuturor celor interesaţi de aprofundarea problemelor legate de calculul paralel.

Prima parte a cărţii tratează fundamentele calcului paralel şi ale proiectării programelor paralele. Primul capitol introduce noţiunile fundamentale ce intervin în calculul paralel: tipuri de arhitecturi, tipuri de reţele de interconectare, măsuri ale performanţei, caracteristici ale algoritmilor paraleli, etc. În cel de-al doilea capitol se evidenţiază etapele de bază ale construcţiei programelor paralele.

A doua parte formată din capitolele 3 şi 4 tratează paradigme şi tehnici de proiectare ale programelor paralele. Sunt analizate tehnici precum: dublarea recursivă, divide&impera, arbore binar, pipeline, etc. Cititorul poate găsi numeroase exemple care ilustrează aceste tehnici, dar care reprezintă şi soluţii paralele pentru unele probleme foarte des întâlnite.

Cea de-a treia parte a cărţii prezintă diferite modele de dezvoltare formală a programelor paralele.

UNITY "Unbounded Nondeterministic Iterative Transformations" reprezintă o teorie, care este în acelaşi timp şi un model de calcul şi un sistem de demonstrare. Această

teorie are ca scop introducerea unei metodologii de dezvoltare de programe, care să fie independentă de arhitectura secvenţială sau paralelă, pe care se vor implementa.

În Capitolul 6 se prezintă o metodă de dezvoltare corectă a programelor paralele, pornind de la specificaţii. Metoda prezentată foloseşte procese parametrizate şi structurarea pe nivele a programelor. Distribuţia datelor este determinantă în construcţia programelor paralele şi este aici analizată formal.

Proprietăţile programării funcţionale fac ca aceasta să aducă avantaje importante pentru programarea paralelă, făcând posibilă dezvoltarea programelor prin transformări riguroase şi exploatând mecanisme de abstractizare a datelor. Capitolul 7 prezintă o abordare funcţională de dezvoltare a programelor paralele, care combină abstractizarea şi performanţa într-un mod sistematic. Este folosit formalismul BMF cu structura de bază lista, cu operatorul de concatenare ca şi constructor.

Algoritmii recursivi intervin în rezolvarea unei mari varietăţi de probleme, constituindu-se astfel într-o clasă importantă de probleme. *PowerList*, *ParList* şi *PList* sunt teorii bazate pe structuri de date liniare, ce pot fi folosite cu succes în descrierea funcţională simplă a programelor paralele care sunt de natură Divide&Impera. Folosind tehnici formale se pot deriva descrieri succinte pentru programele paralele, plecând de la specificaţii. Capitolul 8 descrie acest formalism.

Ultimul capitol face o analiză şi o clasificare generală a modelelor de calcul paralel existente, plecând de la caracteristicile pe care un asemenea model trebuie să le satisfacă.

Notaţii folosite

Notaţia pentru exprimarea cuantificărilor diferă puţin de cea folosită în mod uzual în matematică. Formatul general este:

$$(\odot k : Q : E),$$

unde $\odot$ este un cuantificator, de exemplu $\sum, \max, \forall$ etc., k este o listă de variabile legate, Q este un predicat care descrie domeniul variabilelor, iar E este o expresie care conţine variabilele k. Tipul de bază al variabilelor este tipul întreg.

Această notaţie are avantajul clarităţii, în cazul în care domeniul variabilelor este exprimat prin mai multe relaţii.

Mulţimile sunt notate în mod similar cu cuantificările. Notaţia

$$V = \{i, j : i^2 + j^2 = a^2 : (i, j)\}$$

specifică mulţimea tuturor perechilor de întregi de pe cercul cu centrul în origine şi de rază a. Cardinalul mulţimii V este notat cu $|V|$.

Aplicaţia funcţiilor este notată prin punct (.). Are prioritate maximă şi este asociativă la stânga. De exemplu, $f.a.b$ este echivalent cu $(f.a).b$. Vom folosi această notaţie doar pentru funcţiile care exprimă algoritmi sau au legătură cu derivarea programelor; pentru funcţiile care exprimă formule matematice vom folosi notaţia clasică.

Pentru funcţiile anonime se foloseşte o abstractizare lambda. De exemplu, $(\lambda n \cdot n^2)$ reprezintă funcţia care returnează pătratul argumentului său.

Derivarea demonstraţiilor este făcută cu următorul stil de notaţie, datorat lui W.F.H. Feijen:

$$\begin{array}{ll} & E_0 \\ = & \quad \{\text{explicaţia egalităţii}\} \\ & E_1 \\ \geq & \quad \{\text{explicaţia inegalităţii}\} \\ & E_2 \end{array}$$

unde $E_i, 0 \leq i < 3$ sunt expresii. În acest fel, derivarea $E_0 \geq E_2$ este făcută prin intermediul expresiei intermediare E_1.

Pentru operaţiile cu numere întregi, notăm operaţia *modulo* cu %.

Mulţimea numerelor $\{0, 1, \ldots, n-1\}, n > 0$ se va nota cu $\overline{n}$.

Listă de figuri

Partea I

Fundamente

Capitolul 1

Noţiuni generale ale calculului paralel

1.1 Clasificări ale sistemelor paralele

Spre deosebire de calculatoarele secvenţiale unde modelul de calcul von Neumann este unic şi foarte simplu conceptual, în domeniul calculului paralel coexistă mai multe modele de arhitecturi care trebuie avute în vedere. Nevoia de ordonare a determinat, ulterior, propunerea a numeroase clasificări, câteva dintre ele fiind consacrate de către comunitatea stiinţifică. Clasificarea propusă de M. Flynn [61] este cea mai populară, probabil şi datorită simplităţii ei.

1.1.1 Clasificarea lui Flynn

Clasificarea lui Flynn [61] se realizează preponderent pe baza definirii şi utilizării noţiunilor de flux de instrucţiuni şi flux de date. Conform acestei clasificări avem următoarele categorii de calculatoare:

1. *Sisteme de calcul cu flux unic de instrucţiuni şi flux unic de date (SISD - "Single Instruction Single Data")* în care un flux unic de instrucţiuni operează asupra unui singur flux de date. Acest tip de arhitectură corespunde sistemelor de calcul monoprocesor, clasice. Totuşi ele nu exclud paralelismul în totalitate (organizări de tip look-ahead, pipeline, blocuri funcţionale ale microprocesoarelor moderne, ...).

   Calculatoarele SISD pot avea mai multe unităţi funcţionale de exemplu: coprocesoare matematice, unităţi vectoriale, procesoare grafice şi de intrare-ieşire. Aceste calculatoare se încadrează în categoria SISD atâta timp cât au doar un procesor. Exemple de calculatoare SISD care folosesc paralelismul sunt: CDC 6600 care are multiple unităţi funcţionale, CDC 7600 care are o unitate aritmetică de tip pipeline, Cray-1 care suportă procesarea vectorială.

2. *Sisteme de calcul cu un flux de instrucţiuni şi fluxuri de date multiple (SIMD - "Single Instruction Multiple Data")* în care un singur flux de instrucţiuni operează

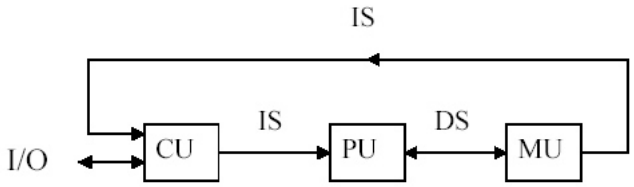

Figura 1.1: Arhitectura uniprocesor SISD (IS – flux de instrucţiuni, DS – flux de date, CU – unitate de control, PU – procesor, MU – memorie).

asupra mai multor fluxuri de date. Această clasă presupune existenţa unor elemente de procesare identice (PE) şi capabile să execute aceeaşi instrucţiune pe seturi diferite de date şi care sunt dirijate de către o unitate de control (CU). Modelele SIMD pot varia în funcţie de modul de transmitere a datelor între elementele de procesare. Elementele de procesare pot comunica între ele printr-o memorie partajată sau prin intermediul unităţii de control.

Execuţia instrucţiunilor este sincronă, în sensul că fiecare PE care execută o instrucţiune în paralel trebuie să termine execuţia respectivei instrucţiuni înainte ca o nouă instrucţiune să fie lansată în execuţie.

Tablourile de procesoare sunt exemple tipice de maşini SIMD. Un tablou de procesoare este format dintr-o unitate de control care decodifică instrucţiunile şi transmite semnale control către mai multe elemente de procesare care lucrează cu datele locale. O unitate centrală, însă, nu poate comanda un număr mare de procesoare, decât cu adoptarea unor soluţii costisitoare. Fiecare PE se poate găsi într-o stare fie activă, fie inactivă în timpul unui ciclu de execuţie. Dacă un PE se găseşte într-o stare inactivă atunci el nu execută instrucţiunea care i-a fost trimisă de către unitatea centrală. Selectarea PE active se face pe baza unor scheme de mascare. Aceste scheme pot depinde de adresele elementelor de procesare sau de datele lor locale.

Procesarea vectorială poate fi executată pe un tablou de procesoare. Prin procesare vectorială se înţelege prelucrarea informaţiei reprezentată sub formă de vectori. O operaţie vectorială presupune operanzi sub forma de vectori, iar rezultatul obţinut este în general tot un vector. Pe un tablou de procesoare, dacă o operaţie este de tip scalar atunci ea va fi executată de unitatea centrală, iar dacă operaţia este de tip vectorial atunci ea va fi executată de către elementele de procesare.

Calculatoarele vectoriale prezintă caracteristice specifice pentru procesarea vectorială şi se încadrează tot în clasa maşinilor SIMD.

Exemple de calculatoare care corespund acestei categorii sunt: ILLIAC-IV, PEPE, BSP, STARAN MPP, DAP şi Connection Machine(CM-1).

3. *Sisteme de calcul cu fluxuri multiple de instrucţiuni şi flux unic de date (MISD - "Multiple Instruction Single Data")* în care mai multe fluxuri de instrucţiuni

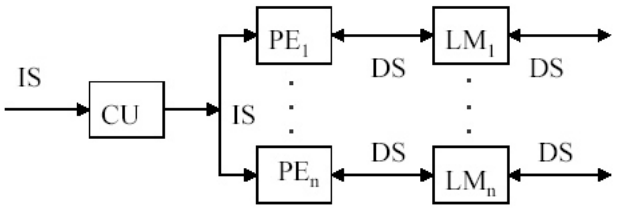

Figura 1.2: Arhitectura SIMD (IS – flux de instrucţiuni, DS – flux de date, CU – unitate de control, PE – element de procesare, LM – memorie locală).

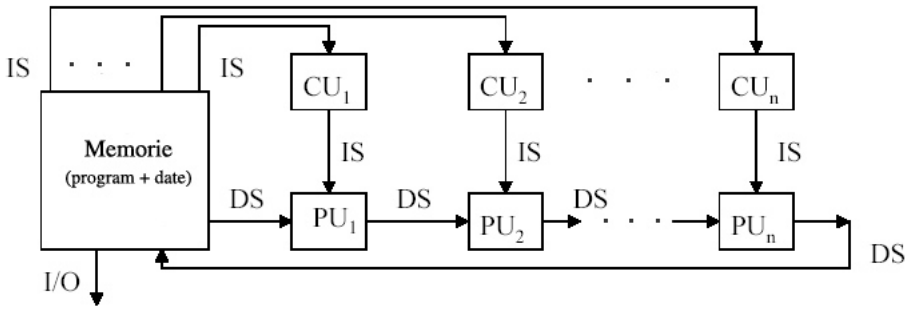

Figura 1.3: Arhitectura MISD – arhitectura sistolică (IS – flux de instrucţiuni, DS – flux de date, CU – unitate de control, PU – procesor).

operează asupra aceluiaşi flux de date. Această clasă pare a fi vidă, cu toate că în ea ar putea fi cuprinse structuri specializate de calcul. Execuţia simultană a mai multor operaţii asupra aceleiaşi date este dificil de definit: poate fi stabilită o ordine de execuţie a operaţiilor şi care este rezultatul final? De aceea, în timp ce unii consideră acest model arhitectural pur teoretic, alţii consideră că arhitecturile sistolice s-ar încadra în această categorie.

O arhitectură sistolică constă dintr-o mulţime de elemente de procesare, sincronizate şi construite pentru un scop precis, care sunt interconcetate într-o reţea cu structură fixă. Fiecare PE "pompează" în mod regulat datele în interior şi în exterior, de fiecare dată executând un anumit calcul simplu. Funcţia unui asemenea PE şi schema de interconectare depinde în general de problema care se rezolvă. Simplitatea elementelor de procesare şi uniformitatea şablonului de interconectare permite maşinilor sistolice cu un număr mare de elemente de procesare să fie implementate pe un singur chip folosind tehnologia VLSI ("Very Large System Integration"). Sarcinile de calcul se divid în două clase mari: calcule propriu-zise şi operaţii de I/O.

În general o maşină sistolică poate fi definită ca o reţea de calcul care are următoarele proprietăţi:

i Simplitate şi regularitate: maşina constă într-un număr mare de elemente de procesare simple cu interconexiuni simple şi omogene.

ii Sincronicitate: Datele sunt calculate ritmic fiind dirijate de un ceas global.

iii "Pipelining": La fiecare pas rezultatul unui calcul executat de către un PE devine intrare pentru un PE vecin.

Reţeaua de interconectare poate fi liniară, caz în care putem considera că avem un calculator pipeline, sau poate fi mai complexă - de exemplu bidimensională.

Aceste tipuri de arhitecturi au fost încadrate în diferite exemple din literatură atât în clasa SISD cât şi în SIMD.

4. *Sisteme cu fluxuri multiple de instrucţiuni şi fluxuri multiple de date (MIMD - "Multiple Instruction Multiple Data")* sunt sistemele la care mai multe fluxuri de instrucţiuni operează simultan asupra unor fluxuri de date diferite. Fluxurile de instrucţiuni pot fi identice (SPMD - "Single Program Multiple Data") sau nu, dar ele nu se execută sincron. Această clasă include toate formele de configuraţii multiprocesor, de la reţelele de calcul de uz general până la masivele de procesoare. Ele pot fi cu memorie comună sau distribuită. Aceste tipuri de sisteme reprezintă cele mai folosite arhitecturi paralele.

   Calculatoarele MIMD, sau calculatoarele cu unităţi de control (CPU) multiple, constau într-un număr de elemente de procesare, care pot fi indexate individual. Fiecare are propriul indice şi memorie locală unde pot fi stocate atât datele cât şi programul. Elementele de procesare sunt complet programabile şi pot executa propriul program.

   Arhitecturile MIMD cu memorie partajată permit fiecărei unităţi de procesare să acceseze o memorie globală prin intermediul unei reţele de interconectare. Procesele care se execută în paralel pot să îşi partajeze date pentru diferite scopuri, inclusiv pentru a-şi sincroniza activitatea. Această partajare a datelor pune programatorilor două probleme importante: sincronizarea accesului la date şi menţinerea consistenţei. Câteva caracteristici care le diferenţiază de calculatoarele MIMD cu memorie distribuită sunt:

   - Comunicarea se face prin intermediul memoriei comune.
   - Întârzierea ("latency") datorată accesului la memorie poate fi mare şi variabilă.
   - Coliziunile sunt posibile la accesarea memoriei.

   Într-o arhitectură MIMD cu memorie distribuită unităţile de procesare sunt conectate printr-o reţea de interconectare. Fiecare unitate de procesare are propria memorie locală; dacă o anumită dată memorată în memoria unei unităţi de calcul este dorită de o alta, atunci data trebuie să fie trimisă explicit de la o unitate de procesare la cealaltă. Calculatoarele MIMD cu memorie distribuită au câteva caracteristici de bază care le diferenţiază:

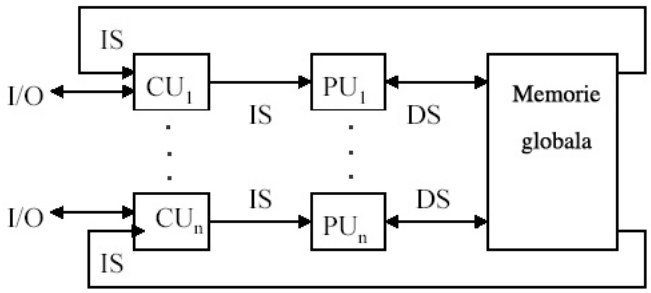

Figura 1.4: Arhitectura MIMD cu memorie partajată (IS – flux de instrucţiuni, DS – flux de date, CU – unitate de control, PU – procesor).

- Comunicarea se face prin soft, folosind instrucţiuni de transmitere de date ("send/receive").
- Mesajele de dimensiuni mari pot ascunde întârzirea("latency").
- Este necesar un management atent al comunicaţiilor pentru a evita interblocările.

Principala proprietate a sistemelor cu memorie distribuită, care le avantajează faţă de cele cu memorie comună, este *scalabilitatea*. Scalabilitatea referă posibilitatea de a creşte eficienţa prin creşterea numărului de procesoare. Deoarece nu există nici un element care ar putea duce la strangularea comunicaţiilor, cum este accesul la memoria comună, sistemele cu memorie distribuită pot avea, cel puţin în principiu, un număr nelimitat de procesoare.

Clasificări mai detaliate ale acestor calculatoare sunt date de Hockeney [90] şi Bell [15]. Exemple de calculatoare care corespund categoriei MIMD sunt: Cray-2, Cay X-MP, HEp, IBM 370/168 MP, Univac 1100/80, Tndem/16, IBM 3081/3084, iPCS.

Sisteme Cluster. Reţelele de calculatoare au devenit în ultimii ani o alternativă foarte atractivă pentru înlocuirea costisitoarelor supercalculatoare. Ele s-au dovedit a fi utile chiar şi în domenii ale calculului de înaltă performanţă ("High-Performance Computing"), unde performanţa este crucială. Sistemele de tip cluster reprezintă un exemplu de acest tip care a avut un deosebit succes în ultimul timp. Acestea presupun interconectarea de staţii de lucru de performanţă ridicată prin intermediul unor strategii clasice de interconectare (de exemplu: Ethernet sau Linux OS). În anumite cazuri, memoria acestor staţii de lucru poate fi gestionată în aşa fel încât tehnici specifice arhitecturilor cu memorie partajată să poate fi folosite. Sistemele cluster sunt sisteme scalabile, pot fi adaptate în funcţie de buget şi de nevoile de calcul şi permit execuţia eficientă atât a aplicaţiilor secvenţiale, cât şi a celor paralele. Câteva exemple de asemenea sisteme sunt: Berkeley NOW, HPVM, Beowulf, Solaris-MC.

Poate fi considerată o nouă clasă, hibridă, de sisteme paralele: SIMD-MIMD. Aceste sisteme numite şi SAMD ("Synchronous-Asynchronous Multiple Data"), sunt în mod

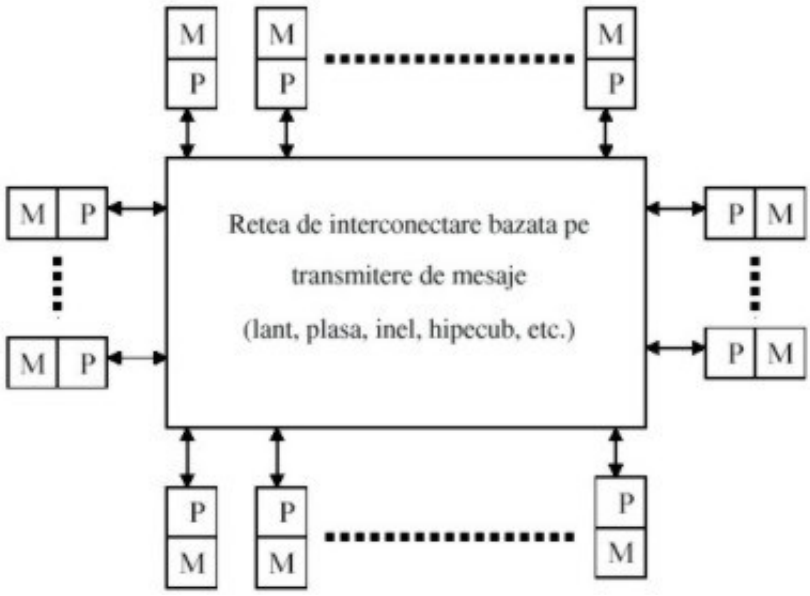

Figura 1.5: Multicalculator bazat pe transmitere de mesaje (P – procesor, M – memorie).

esenţial calculatoare MIMD care au în plus următoarele caracteristici:

- Permit asigurarea sincronizării proceselor.
- Sincronizarea proceselor poate fi menţinută fără un consum mare de timp, prin faptul că există un "ceas" uniform pentru toate procesoarele.

În timp s-a dezvoltat şi o a doua generaţie de abstractizări pentru aceste clase. Câteva dintre acestea sunt:

- Relaxarea sistemelor de calcul SIMD astfel încât sincronizarea să nu mai apară după fiecare pas-instrucţiune, ci doar ocazional. Rezultatul este modul de lucru SPMD("Single Program Multiple Data"), care se poate executa pe maşini SIMD, dar poate fi folosit eficient şi pe alte arhitecturi.
- Modul de tratare a transmiterii mesajelor, al clasei MIMD cu memorie distribuită, poate fi lărgit pentru a se permite o dezvoltare mai simplă a softului prin adaugarea de concepte cum ar fi: spaţii partajate asociative ("tuple spaces") pentru comunicaţie, sau comunicaţie mapată în memorie astfel încât operaţiile "send" şi "receive" se aseamănă cu accesul la memorie("put/get").

1.1.2 Alte clasificari

Clasificarea lui Schwartz

Schwartz a introdus o clasificare a calculatoarelor paralele utilizate în cercetare [151]. El utilizează pentru aceasta noţiunea de *granulaţie* prin care se înţelege numărul de

procesoare din sistem. Conform acestei clasificări, calculatoarele sunt cu granulaţie brută (cele care conţin câteva zeci de procesoare), medie (sute de procesoare) şi cu granulaţie fină (cu mii şi zeci de mii de procesoare).

Clasificarea lui Handler

Handler propune o notaţie elaborată pentru exprimarea paralelismului calculatoarelor [83]. Clasificarea lui Handler compară calculatoarele pe trei nivele distincte: unitatea de control a procesorului (PCU), unitatea aritmetică (ALU) şi nivelul circuitului de bit (BLC). PCU corespunde procesorului sau CPU, ALU corespunde unei unităţi funcţionale sau unui element de procesare într-un tablou de procesoare, iar BCL corespunde logicii necesare realizării operaţiilor pe bit în ALU.

Taxonomia lui Handler foloseşte trei perechi de întregi pentru a descrie un calculator:

$$calculator = (k * k^{'}, d * d^{'}, w * w^{'})$$

unde:
k = numărul de PCU;
$k^{'}$ = numărul de PCU care pot fi aranjate în pipeline;
d = numărul de ALU controlate de fiecare PCU;
$d^{'}$ = numărul de ALU care pot fi aranjate în pipeline;
w = numărul de biţi ai cuvântului din ALU sau ai elementului de procesare (PE);
$w^{'}$ = numărul de segmente pipeline din toate ALU sau dintr-un singur PE.

Următoarele reguli şi operatori sunt folosiţi pentru a arăta relaţiile dintre diferitele elemente ale calculatorului:

- Operatorul $*$ este folosit pentru a indica faptul că unităţile sunt legate în pipeline cu un singur flux de date în toate unităţile.
- Operatorul $+$ poate fi folosit pentru a arata că unităţile nu sunt legate în pipeline, ci lucrează independent pe fluxuri diferite de date.
- Operatorul v poate fi folosit pentru a indica faptul că calculatorul poate lucra în moduri diferite.
- Simbolul $\sim$ indică rangul valorilor pentru un parametru.

De exemplu, calculatorul Cray-1 este un calcultor cu un singur procesor pe 64 de biţi, care are 12 unităţi funcţionale, dintre care 8 pot fi înlănţuite pentru a forma un pipeline. Unităţile funcţionale au între 1 şi 14 segmente care pot fi deasemenea legate în pipeline. Descrierea lui Handler pentru acesta este:

$$\text{Cray-1} = (1, 12 * 8, 64 * (1 \sim 14))$$

Descrierea lui Handler este foarte potrivită descrierii procesoarelor de tip pipeline. Este, de asemenea, bine adaptată descrierii paralelismului pentru un singur procesor, dar nu tocmai potrivită descrierii varietăţii de paralelism care poate apare într-un calculator multiprocesor.

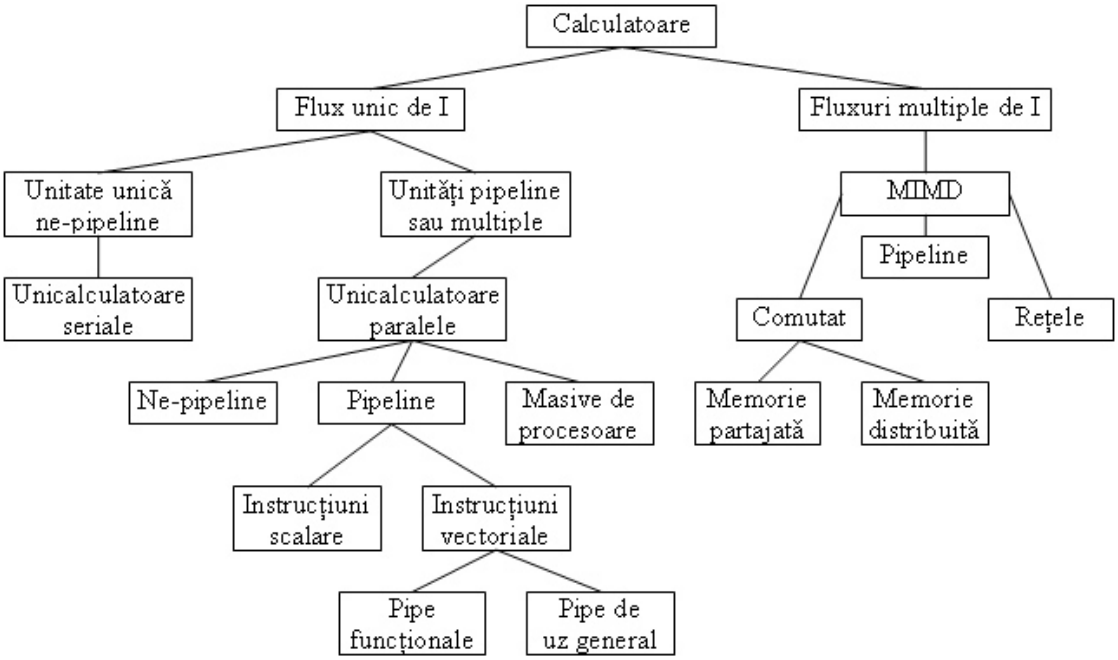

Figura 1.6: Clasificarea lui Hocney.

Clasificarea lui Hockney

Hockney propune o altă clasificare foarte detaliată a calculatoarelor atât seriale cât şi paralele [90]. Clasificarea este dată în formă ierarhică, iar o variantă simplificată este prezentată în Figura 1.6.

La nivelul cel mai înalt se respectă clasificarea funcţională a lui Flynn, în sensul împărţirii calculatoarelor în cele cu un singur flux de instrucţiuni (SI) şi cele cu mai multe fluxuri de instrucţiuni (MI).

Este demn de remarcat faptul că această clasificare evidenţiază procesarea pipeline atât în cazul fluxului unic de instrucţiuni, cât şi în cazul fluxurilor multiple de instrucţiuni.

Clasificarea lui Bell

O altă clasificare, numai a sistemelor MIMD, cunoscută ca şi taxonomia lui Bell, a fost propusă de Gordon Bell în 1992. Acesta a considerat multiprocesoarele cu memorie partajată ca având un spaţiu unic de adresare. Multiprocesoarele scalabile trebuie să utilizeze o memorie distribuită. Mutiprocesoarele nescalabile utilizează memorie partajată.

- Multiprocesoare(spaţiu unic de adrese)
  - cu memorie distribuită (scalabil)
  - cu memorie comună (nescalabil)

- Multicalculatoare(mai multe spaţii de adrese, transfer de mesaje)
  - distribuite (scalabil)
  - centralizate

Clasificarea lui Lewis

Ted Lewis[108] propune ca în clasificarea sistemelor de calcul să se plece de la cele două modele fundamentale de paralelism, cel generat de *fluxul controlului*, iar al doilea generat de *paralelismul de date*("data-parallelism"). Primul tip de paralelism se referă la construirea unui program paralel din mai multe procese ce se pot executa simultan. În acest caz, există gradul cel mai mare de generalitate, dar nu se pot obţine niveluri foarte ridicate de paralelism. Sistemele de calcul sunt de tipul MIMD. Al doilea tip presupune execuţia aceloraşi operaţii simultan, dar cu date diferite. De data aceasta, se poate obţine un nivel foarte ridicat de paralelism, dar numai la algoritmii care folosesc operaţii cu vectori sau matrice. Sistemele de calcul paralel corespunzătoare sunt SIMD sau SPMD.

În ceea ce priveşte analiza proiectării algoritmilor paraleli clasificarea cea mai utilă ar fi aceea în care se iau în considerare două clase mari de arhitecturi paralele:

i Multiprocesoare cu memorie partajată.

ii Multicalculatoare cu memorie distribuită.

Aceasta pentru că un algoritm paralel poate fi văzut ca şi o mulţime de componente de calcul ("task"-uri) independente, care se pot executa concurent şi în mod cooperativ pentru rezolvarea unei probleme. La proiectarea unui asemenea algoritm ne interesează să ştim că avem unităţi de procesare independente şi modul în care acestea pot interacţiona: fie prin intermediul unei memorii comune, fie prin intermediul unei reţele de interconectare.

În general, componentele care constituie programul paralel trebuie să coopereze într-un anumit mod pentru a rezolva o problemă. De asemenea, foarte des poate apare necesitatea ca anumite sarcini să se execute doar după ce s-a ajuns la o anumită stare, sau anumite operaţii au fost executate. Prin urmare, posibilitatea *sincronizării* proceselor este esenţială. În funcţie de clasa de arhitectură paralelă - cu memorie partajată sau distribuită - sincronizarea poate fi obţinută prin diferite metode specifice. Pe lângă încărcarea echilibrată a componentelor, posibilitatea de a implementa mecanisme eficiente de sincronizare poate conduce la obţinerea de programe paralele eficiente.

1.2 Criterii de performanţă ale sistemelor paralele

Posibilităţile hardware ale unui sistem paralel pot fi evaluate pe baza unor criterii bine precizate; câteva dintre acestea sunt prezentate pe scurt în continuare:

- *Viteza de calcul* maximă, măsurată în Mflop/s (Million FLoting point OPeration per Second), sau în MIPS (Million Instructions Per Second). În descrierea vitezelor de calcul, se folosesc în general două valori: una de vârf, care presupune că procesoarele lucrează la încărcare maximă (ignorând conflictele de access la memorie, comunicaţie, sincronizări, timpi morţi) şi cealaltă, evaluată cu programe speciale de test ("benchmark"), care rezolvă probleme des întâlnite în calcule ştiinţifice.
- *Numărul de procesoare.* Se proiectează variante arhitecturale cu număr de procesoare de la câteva zeci, la câteva mii. Este indicat ca aceste sisteme să fie scalabile, adică să poată fi adăugate noi procesoare, uşor, fără necesitatea unor modificări laborioase.
- Dimensiunea memoriei locale a unui procesor sau a memoriei globale, dacă există. Acesta este un parametru de mare importanţă, deoarece dictează dimensiunea problemelor care pot fi rezolvate.
- *Frecvenţa* ceasului unui procesor, care dă informaţii despre puterea unui nod de procesare.
- Viteza de comunicaţie şi caracteristici ale bandei de trecere a unei legături, măsurată în număr de octeţi transferaţi pe secundă. Este esenţială pentru sistemele cu memorie distribuită. Ca şi la viteza de calcul, trebuie să se facă diferenţe între viteza de vârf şi cea atinsă efectiv în aplicaţii reale.
- *Viteza de acces la memorie* a fiecărui procesor, esenţială pentru calculatoarele cu memorie comună.
- Viteza de comunicaţie cu exteriorul, care poate produce un efect de strangulare a vitezei efective de calcul, prin introducerea unor timpi morţi, datorită nealimentării cu programe şi date.
- Suportul sofware oferit odată cu calculatorul. În ultimii ani se constată un efort intens de standardizare, materializat prin apariţia de biblioteci implementate eficient pe diferite clase de arhitecturi. Aceasta conduce la portabilitatea programelor.
- Raportul dintre costul calculatorului şi viteza sa de calcul – preţul unui Mflop/s. Tendinţa actuală este de menţine cât mai scăzut acest raport, chiar dacă nu se obţine o viteză de calcul deosebit de ridicată. Este şi motivul pentru care sistemele cluster s-au impus atât de mult în ultima vreme în detrimentul costisitoarelor supercalculatoare.

1.3 Reţele de interconectare a procesoarelor

Performanţele unei arhitecturi paralele cu memorie distribuită depind mult de numărul de procesoare dar şi de modul în care acestea sunt interconectate.

Reţelele de interconectare se pot clasifica în trei categorii:

- Fiecare procesor este conectat direct cu oricare altul. Această construcţie este practic inexistentă, exceptând maşinile cu număr mic de procesoare.

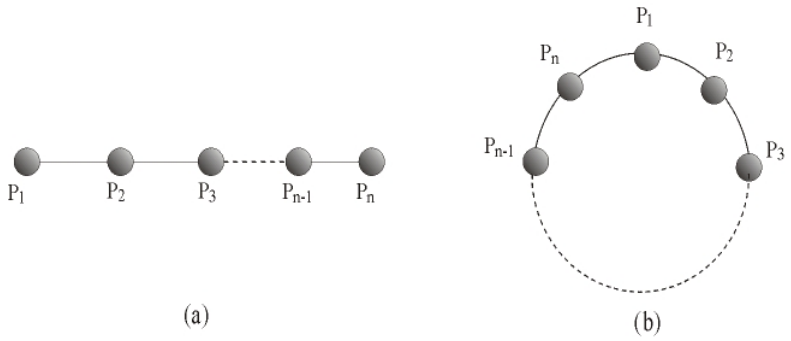

Figura 1.7: (a) – Reţeaua liniară(lanţ); (b) – reţeaua ciclică(inel).

- Fiecare procesor este conectat cu oricare altul prin intermediul unei punţi de legătură numită "switchboard". Această conectare asigură legătura între oricare două procesoare după un număr mic de paşi. Se pot utiliza în plus conexiuni directe între procesoare învecinate.

- Fiecare procesor este conectat direct cu un număr de alte procesoare.

În continuare, vom prezenta câteva reţele de interconectare din cea de-a treia categorie, care sunt mai des folosite în practică.

Înainte de a le prezenta, să vedem care sunt proprietăţile unei astfel de topologii.

În primul rând graful asociat trebuie sa fie *conex* (există cel puţin un drum între oricare două noduri ale sale) şi pe cât posibil *regulat* (gradele tututor nodurilor să fie egale). Este indicat ca un procesor să nu aibă prea multe canale de comunicaţie, pentru a fi uşor de realizat; strâns legată de aceasta este cerinţa ca numărul de arce să fie relativ mic. Prin urmare se urmăreşte ca *gradul grafului* – g (maximul gradelor tuturor nodurilor) să fie cât mai mic. Pe de altă parte, este bine ca *diametrul* – d (maximul tuturor distanţelor între perechi de noduri din graf) să fie cât mai mic, astfel încât distanţele între procesoare să fie limitate şi deci comunicarea cât mai facilă. Evident cele două deziderate sunt antagonice; cu cât gradul este mai mic, cu atât sunt mai puţine muchii şi deci scad şansele să existe drumuri scurte între oricare două procesoare.

1.3.1 Topologii de bază

Reţea liniară şi ciclică

Într-o reţea liniară (*lanţ*), un procesor P_i este conectat cu vecinii săi P_{i-1} şi P_{i+1} pentru toţi $i : 1 < i < n$ (Figura 1.7 (a)). Gradul este 2, iar diametrul este $n - 1$.

Reţeaua ciclică (*inel*) aduce un plus de flexibilitate prin stabilirea unei conexiuni şi între primul şi ultimul procesor (Figura 1.7 (b)). Diametrul se reduce la jumătate – $\lceil n/2 \rceil$, iar gradul rămâne egal cu 2.

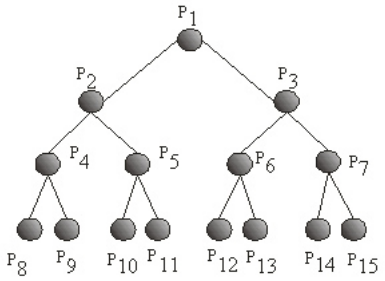

Figura 1.8: Reţeaua arbore binar pentru $n = 15$.

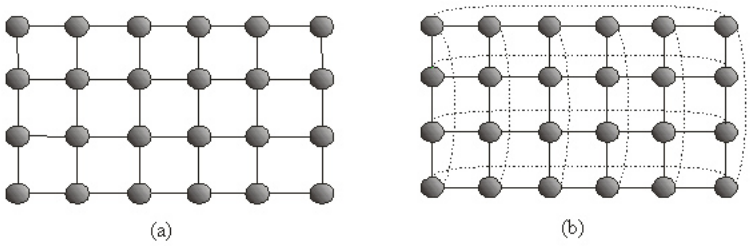

Figura 1.9: (a) Reţeaua grilă cu $M = 4$ şi $N = 6$. (b) Reţeaua tor cu $M = 4$ şi $N = 6$.

Reţea arbore binar

Numeroşi algoritmi pot fi implementaţi convenabil pe calculatoare paralele ale căror procesoare sunt conectate în structură de arbore binar. Acest tip de reţea este foarte convenabil datorită diametrului mic, care este $\lceil \log_2 n \rceil$ (n =numărul de procesoare); gradul este egal cu 3.

Reţea grilă

Se pot obţine noi reţele de interconectare prin considerarea produsului cartezian a altor reţele. De exemplu, o reţea grilă(latice) $M \times N, M, N > 0$ este produsul cartezian a două reţele *lanţ* de lungime M şi N (Figura 1.9 (a)). Diametrul acestei reţele este $M + N - 2$, iar gradul este 4.

O reţea *tor* sau *toroid* este produsul cartezian a două reţele *inel* (Figura 1.9 (b)). Diametrul scade în acest caz şi este $\lceil M/2 \rceil + \lceil N/2 \rceil$.

Reţea fluture

O reţea fluture de ordin k are $(k + 1)2^k$ procesoare dispuse în nodurile unui graf şi are următoarele proprietăţi:

- există 2^k coloane fiecare cu câte $k + 1$ noduri;

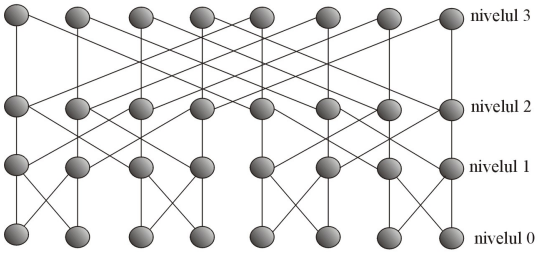

Figura 1.10: Reţeaua fluture pentru $k = 3$.

- procesoarele de pe fiecare coloană sunt numerotate de la 0 la k, iar acest număr reprezintă rangul acelui nod;
- procesorul de rang r de pe coloana j se notează cu $d_{r,j}$;
- procesorul $d_{r,j}$ este conectat cu procesoarele $d_{r-1,j}$ şi $d_{r-1,j'}$, unde j' este numărul al cărei reprezentări binare pe k biţi este aceeaşi ca şi reprezentarea binară a lui j cu excepţia bitului $r - 1$.

În anumite cazuri, procesoarele de rang 0 şi k coincid, caz în care toate procesoarele sunt conectate cu exact alte 4 procesoare.

Diametrul este egal cu k.

Reţeaua fluture de rang k are două proprietăţi remarcabile:

1. Dacă procesoarele de rang k se şterg, împreună cu toate arcele incidente, rezultă două reţele fluture de rang $k - 1$.
2. Dacă procesoarele de rang 0 se şterg, împreună cu toate arcele incidente, rezultă două reţele fluture interclasate de rang $k - 1$.

Reţea hipercub

Un *hipercub binar* H_n de dimensiune n are 2^n noduri şi se obţine din produsul cartezian a n lanţuri de lungime 2. Nodurile sunt etichetate prin numere reprezentate binar prin n cifre binare, iar două noduri sunt conectate dacă şi numai dacă numerele corespunzătoare lor diferă doar într-o singură poziţie binară. Prin urmare, fiecare nod are exact n vecini şi diametrul este n. Dacă se înlocuieşte 2 cu un număr k oarecare se obţin n-cuburi cu aritate k.

O altă modalitate de definire a unui hipercub binar este recursivă. H_n este obţinut din două hipercuburi H_{n-1} prin unirea corespunzătoare a nodurilor: fiecărui număr care reprezintă un nod i se adaugă un nou bit pe poziţia cea mai semnificativă (egal cu 0 pentru primul hipercub şi egal cu 1 pentru cel de-al doilea) şi apoi se unesc nodurile care diferă în cel mult o poziţie binară. Hipercubul H_1 este format din două procesoare unite.

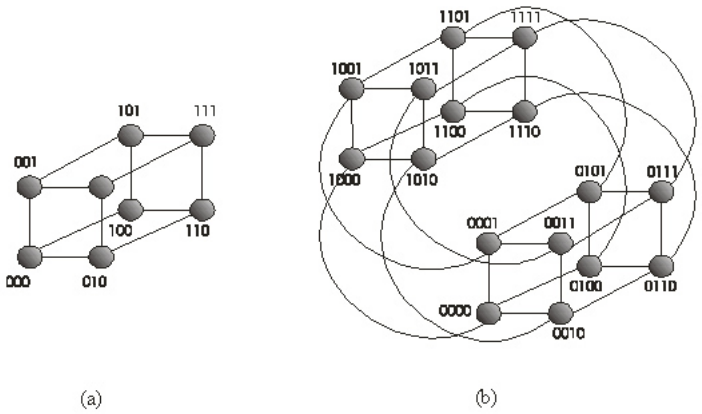

Figura 1.11: Reţeaua hipercub. (a) Reţeaua hipercub pentru $n = 3$.(b) Reţeaua hipercub pentru $n = 4$ (se formează prin combinarea a două hipercuburi de dimensiune 3).

Definiţia 1.1 (Distanţa Hamming) *Fie a şi b două secvenţe de biţi de aceeaşi lungime.* Distanţa Hamming *dintre aceste secvenţe este numărul de poziţii din cele două secvenţe în care există valori diferite.*

Evident distanţa dintre două noduri ale unui hipercub este egală cu distanţa Hamming dintre reprezentările binare ale indecşilor celor două noduri. Diametrul pentru reţeaua H_n este n, egală cu gradul.

Un hipercub de dimensiune n este echivalent cu o reţea fluture ale cărei coloane sunt unite într-un singur vârf şi care va avea jumătate din numărul de muchii ale reţelei fluture.

Reţea amestecare perfectă

Acest tip de reţea este unul special, inspirat de amestecarea perfectă a cărţilor de joc dintr-un pachet. Structura acestei reţele este exploatată de mulţi algoritmi paraleli dintre care cel mai cunoscut este cel de sortare bitonică (Secţiunea 4.14.1).

Se consideră $n = 2^m$ cărţi de joc împărţite în mod egal în două pachete. Se amestecă aceste cărţi astfel încât prima carte din pachetul al doilea se află între prima şi a doua carte din primul pachet, a doua carte din al doilea pachet între a doua şi a treia carte din primul ş.a.m.d. Acest procedeu de amestecare este numit *amestecare perfectă*.

Amestecarea se bazează pe o permutare π:

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4\ldots & n-2 & n-1 \\ 0 & n/2 & 1 & n/2+1 & 2\ldots & n/2-1 & n-1 \end{pmatrix}$$

Permutarea inversă π^{-1} are următoarea proprietate:

$$\pi^{-1}(x) = 2x\%(n-1), x = 0, \ldots, n-2$$
$$\pi^{-1}(n-1) = n-1.$$

Conectarea este definită în următorul mod:

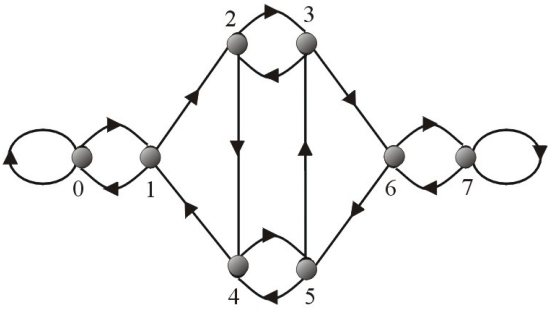

Figura 1.12: Reţeaua amestecare perfectă pentru $n = 8$.

- procesorul x este conectat cu procesorul $\pi^{-1}(x)$, dacă $x = 0, 1, \ldots, n-1$,
- procesorul $2x$ este conectat cu procesorul $2x + 1$,
- procesorul $2x + 1$ este conectat cu procesorul $2x$,

Acestui procedeu de amestecare i se poate asocia o topologie de interconectare a procesoarelor.

O reţea amestecare perfectă pentru $n = 8$ este prezentată în Figura 1.12.

1.3.2 Problema includerii

Mulţi algoritmi folosesc o topologie virtuală în care comunicaţia se desfăşoară doar între vecinii grafului. Performanţa obţinută poate fi îmbunătăţită considerabil dacă topologia virtuală a algoritmului corespunde cu topologia fizică, reală, a sistemului paralel.

Totuşi, în proiectarea algoritmilor este indicată o distanţare faţă de arhitectură (chiar independenţă), pentru a nu fi necesară rescrierea programului pentru fiecare implementare pe un alt sistem. O modalitate de obţinere a portabilităţii se bazează pe găsirea unor modalităţi de includere a unei topologii (cea a algoritmului) în altele, astfel încât, un algoritm să poată fi implementat pe diferite arhitecturi.

Problema este interesantă şi din alte motive, cum este de exemplu, posibilitatea emulării unei arhitecturi prin alta şi determinarea efectului cerut de această emulare.

Această problemă a includerii se bazează pe o problemă matematică, aşa numita *scufundare* a grafurilor [122].

Definiţia 1.2 (Scufundarea grafurilor) *Fie G şi H două grafuri simple neorientate. O* scufundare *a grafului G în graful H este definită printr-o funcţie f definită pe mulţimea nodurilor din G cu valori în mulţimea nodurilor din H, şi o funcţie S_f definită pe mulţimea muchiilor din G cu valori în mulţimea lanţurilor din H; prin S_f se asociază unei muchii (x, y) din G o cale din H care uneşte $f(x)$ cu $f(y)$.*

În general, se consideră doar funcţia f, lăsând funcţia S_f să fie dedusă prin considerarea unui drum minim între $f(x)$ şi $f(y)$ corespunzător muchiei (x, y).

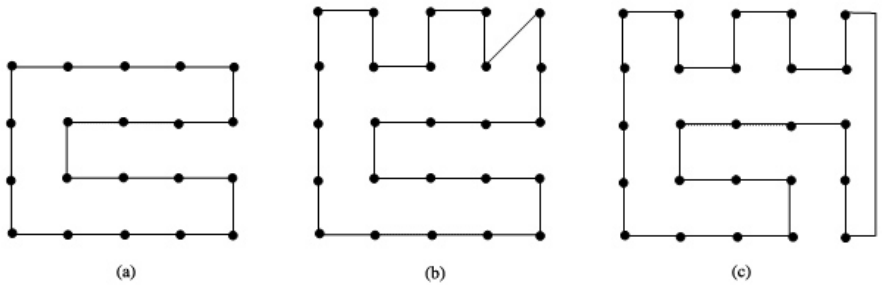

Figura 1.13: (a)Ciclu hamiltonian într-o grilă 4×5; (b) scufundarea unui inel într-o grilă 5×5 cu dilatare 2 (muchia diagonală se mapează printr-un lanţ cu lungimea 2); (c) ciclu hamiltonian într-un tor 5×5.

Dilatarea şi *congestia* sunt două caracteristici importante ale unei scufundări. Dilatarea exprimă lungimea maximă a unei căi $S_f(x, y)$, unde (x, y) este muchie în G; iar congestia este maximul după toate muchiile $e \in E(H)$, a numărului de căi $S_f(x, y)$ care sunt imagini de muchii din G care conţin muchia e din H.

Putem considera H ca fiind arhitectura reală, iar G topologia emulată (fie că ea provine dintr-un algoritm, sau este topologia altei arhitecturi). Dilatarea exprimă distanţa maximă între două noduri emulate; este de dorit să fie cât mai mică, valoarea ideală fiind 1. Congestia dă informaţii despre numărul de canale emulate pe un canal de comunicaţie fizică; cu cât este mai mic cu atât conflictele de utilizare a canalului sunt reduse. Dacă dilatarea este egală cu 1, atunci şi congestia este egală cu 1.

Exemplul 1.1 (Scufundarea unui inel) Dorim să analizăm modalităţile de scufundare cu dilatare 1, a unui inel în topologiile: grilă, tor şi hipercub. Problema se reduce la găsirea unui drum hamiltonian (care trece prin toate nodurile grafului) în grafurile topologiilor respective. Vom prezenta doar rezultatele, fără a intra în detalii ce ţin de demonstrare.

O grilă de dimeniuni $M \times N$ are ciclu hamiltonian dacă ori M ori N este par – Figura 1.13 (a). Dacă ambele dimensiuni sunt impare atunci scufundarea unui inel se poate face doar cu o dilatare egală cu 2. Un exemplu este dat în Figura1.13 (b), unde se observă că linia oblică nu corepunde unei muchii a grilei şi deci este nevoie de un drum de lungime 2 pentru acea legătură.

Un tor are întotdeanuna ciclu hamiltonian; dacă M sau N este par, atunci se aplică rezultatul precedent, iar dacă ambele dimensiuni sunt impare se folosesc muchiile pe care torul le are în plus faţă de de grilă – Figura 1.13 (c).

Şi un hipercub are întotdeauna un ciclu hamiltonian – Figura 1.14. Pentru un hipercub H_{2n} există n cicluri hamiltoniene distincte.

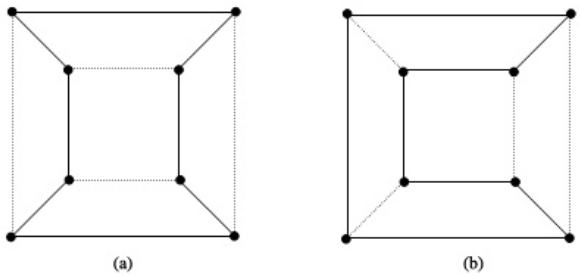

Figura 1.14: Două cicluri hamiltoniene ale hipercubului de ordin 3; liniile punctate indică muchiile din hipercub neutilizate.

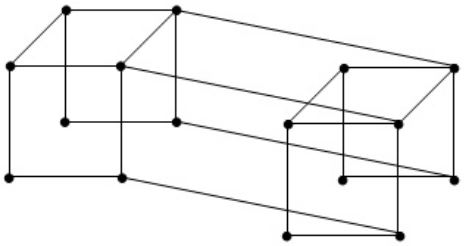

Figura 1.15: Scufundarea unei grile 4×4 într-un hipercub de ordin 4; un tor este chiar hipercubul.

Exemplul 1.2 (Scufundarea unei grile într-un hipercub) O grilă de dimensiuni $2^{n/2} \times 2^{n/2}$ poate fi scufundată cu dilatare 1 într-un hipercub de ordin n (cu n par). Mai general, orice grilă de dimensiuni $2^{n_1} \times 2^{n_2}$, cu $n_1 + n_2 = n$ poate fi scufundată cu dilatare 1 într-un hipercub H_n [167]. Un exemplu este prezentat în Figura 1.15.

1.3.3 Comunicaţia în reţele

Principalele criterii de performanţă de care se ţine cont în proiectarea unei reţele de interconectare sunt asociate cu parametrii următori:

- întârzierea ("latency") - timpul de transmitere pentru un singur mesaj, măsurat de la momentul trimiterii sale până la primirea la destinaţie (este indicat să fie cât mai mic);
- lărgimea de bandă ("bandwidth") - numărul de octeţi transferaţi de reţea în unitatea de timp (este indicat să fie cât mai mare);
- diametrul ("diameter") - distanţa maximă între oricare două procesoare din reţea (este indicat să fie cât mai mic); distanţa dintre două procesoare se defineşte ca fiind lungimea drumului minim dintre ele;

- mărimea bisecţiei ("bisection width") - numărul minim de canale de comunicaţie, ce trebuie eliminate pentru a împărţi reţeaua în două jumătăţi egale (este indicat să fie cât mai mică);
- conectivitatea ("connectivity") - numărul de vecini direcţi ai fiecărui procesor, parametru cunoscut ca şi gradul procesorului;
- costul hardware - proporţia din costul total care reprezintă costul reţelei;
- fiabilitatea - se asigură prin introducerea unor căi redundante, etc.;
- funcţionalitatea - posibilitatea asigurării unor funcţii suplimentare de către reţea, cum ar fi combinarea mesajelor şi arbitrarea.

Este imposibil de satisfăcut toate aceste criterii de performanţă, unele chiar contradictorii. De aceea, trebuie să se decidă care sunt obiectivele de performanţă ce trebuie atinse. De exemplu, nu este posibil întotdeauna ca fiecare procesor să fie conectat direct cu toate celelalte procesoare (cazul ideal), decât în cazul unei configuraţii reduse, cu până la câteva zeci de procesoare. Un procesor este conectat direct doar cu o submulţime de procesoare, care formează vecinătatea sa. În consecinţă, pentru ca un mesaj trimis de oricare dintre procesoare să ajungă la destinaţie, trebuie să existe un mecanism de propagare a mesajului la nivelul fiecărui procesor. Acesta trebuie să decidă care este canalul pe care se transmite în continuare mesajul primit, astfel ca acesta să ajungă în mod sigur la destinaţie şi într-un timp rezonabil. Funcţia de propagare a mesajelor prin reţeaua de comunicaţie poartă numele de dirijare (rutare - "routing"). În legătură cu dirijarea mesajelor prin reţea va trebui să se asigure respectarea următoarelor cerinţe:

- nici un mesaj nu trebuie să fie pierdut;
- strategia de rutare trebuie să evite producerea fenomenului de interblocare;
- toate mesajele trebuie să-şi atingă destinaţia într-un interval de timp finit;
- strategia de dirijare trebuie să fie independentă de topologia şi de mărimea reţelei.

Transmiterea de mesaje între două procesoare poate fi sincronă sau asincronă.

În cazul **transmiterii asincrone de mesaje**, procesul care comunică informaţie continuă execuţia după ce aceasta a fost transmisă spre destinaţie. Procesul receptor va prelua mesaje într-o coadă de mesaje. Mesaje pot fi primite într-o ordine diferită de cea în care au fost trimise, existând şi posibilitatea trimiterii lor cu prioritate.

Transmisia sincronă cere ca atât procesul care trimite cât şi cel care recepţionează mesajul să fie disponibile până în momentul când transmisia s-a terminat, ambele procese putând apoi să-şi continue lucrul. În felul acesta, nu este necesară stocarea mesajelor (lucru care poate fi necesar în cazul transmiterii asincrone).

1.4 Niveluri la care poate apare paralelism

În calculatoarele moderne, paralelismul apare pe diferite niveluri atât în hardware cât şi în software: niveluri semnal şi circuit, componente şi sistem. La cel mai scăzut nivel,

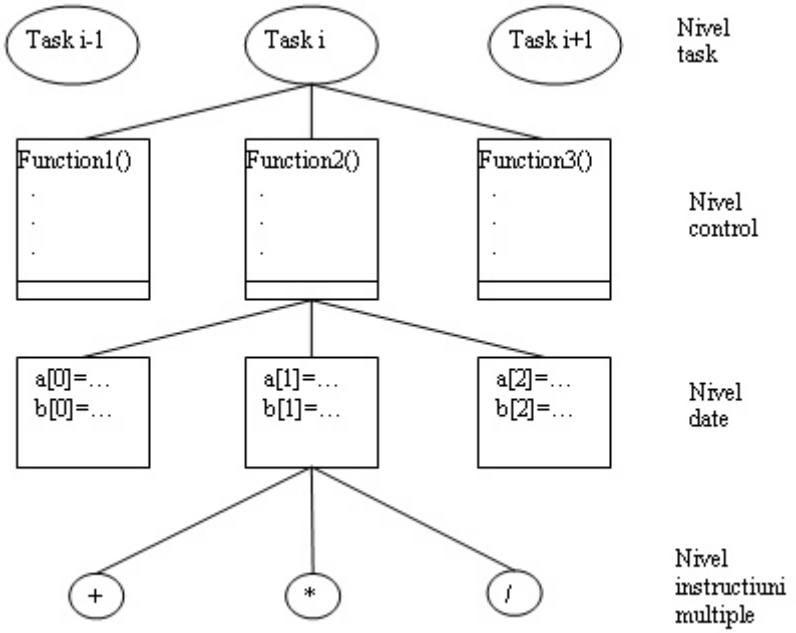

Figura 1.16: Niveluri ale paralelismului.

semnalele sunt transmise în paralel, iar la un nivel un pic mai înalt poate apare aşa numitul paralelism la nivel de instrucţiune. De exemplu, un procesor cum este Pentium Pro are capabilitatea de a procesa trei instrucţiuni simultan. Majoritatea calculatoarelor suprapun activităţile unităţii de control şi cele de intrare-ieşire (I/O). Anumite calculatoare folosesc tehnica de întreţesere a memoriei – câteva blocuri de memorie pot fi accesate în paralel pentru accesul mai rapid la memorie. La un nivel şi mai înalt, sistemele de calcul pot avea mai multe procesoare, iar la un nivel şi mai înalt se pot conecta împreună mai multe calculatoare care pot lucra împreună ca o singură maşină.

La primele două niveluri – semnal şi circuit – paralelismul se realizează prin tehnici hardware şi are loc aşa numitul paralelism hardware. Paralelismul celorlalte două niveluri – componente şi sistem – este în general exprimat implicit sau explicit prin diferite tehnici software care formează paralelismul software.

Nivelurile de paralelism pot fi structurate şi în funcţie de dimensiunea componentelor de cod ("grain size") care se pot executa în paralel. În funcţie de acestea avem următoarele nivele:

- granularitate foarte fină (instrucţiuni multiple) – paralelizarea este realizată de procesor;
- granularitate fină (nivel de date)– paralelizarea este realizată de compilator;
- granularitate medie (nivel al controlului) – paralelizarea este realizată de programator;

- granularitate brută (nivel-task) – paralelizarea este realizată de programator.

Diferitele niveluri de paralelism sunt evidenţiate în Figura 1.16. Primele două niveluri sunt suportate fie de către hardware, fie de către compilatoare de paralelizare. Programatorul se ocupă în general de ultimele două niveluri de paralelism.

1.5 Clasificarea algoritmilor paraleli

Aşa cum am precizat şi anterior, un algoritm paralel poate fi văzut ca şi o mulţime de p componente de calcul/taskuri independente, care se pot executa concurent şi în mod cooperativ la rezolvarea unei probleme. În mod evident, dacă $p = 1$ atunci avem un algoritm secvenţial.

În timpul execuţiei unui algoritm paralel componentele interacţionează prin sincronizare şi/sau schimb de date. Aceste puncte sunt numite *puncte de interacţiune*. O componentă poate fi împărţită în funcţie de aceste puncte de interacţiune într-un anumit număr de etape, astfel încât la sfârşitul unei etape o componentă poate comunica cu alte componente înainte de începerea etapei următoare. În execuţia unui program paralel, timpul asociat unei etape pentru un procesor oarecare este o variabilă aleatoare, datorită următorilor factori:

- procesoarele pot avea viteze şi caracteristici diferite;
- operaţiile efectuate de către un procesor pot fi întrerupte de către sistemul de operare;
- timpul de procesare poate depinde de datele de intrare.

Ca rezultat al necesităţii interacţiuniilor dintre componente, o componentă poate fi blocată pentru o anumită perioadă de timp. Un algoritm paralel în care componentele trebuie să aştepte execuţia unor anumite evenimente care apar în alte procese se numeşte *algoritm sincron*. Într-un algoritm sincronizat componenta ce ajunge la un punct de interacţiune este blocată până când componentele cu care trebuie să interacţioneze ajung şi ele la acel punct de interacţiune. Această formă de sincronizare se numeşte explicită. Deoarece execuţia unei componente este variabilă, toate componentele care trebuie să fie sincronizate la un anumit punct trebuie să aştepte terminarea celui cu execuţia cea mai lungă. Aceasta reprezintă o limitare importantă a algoritmilor paraleli sincroni şi poate duce la o utilizare ineficientă a procesoarelor.

Algoritmii paraleli asincroni elimină problemele implicate de sincronicitate. Componentele algoritmilor asincroni nu trebuie în general să se aştepte unele pe altele. Comunicarea se realizează prin citirea unor variabile globale actualizate în mod dinamic. Pot apare întârzieri şi în acest caz, datorită rezolvării conflictelor ce pot apare în accesarea memoriei comune. Acest tip de sincronizare se numeşte implicit. Pot apare însă probleme legate de consistenţă şi corectitudine şi pentru eliminarea lor sunt programate aşa numitele *secţiuni critice*, iar procesele pot fi blocate înainte de intrarea în aceste secţiuni.

Algoritmii paraleli pot fi clasificaţi şi în funcţie de controlul concurenţei, de granularitate, scalabilitate şi de structura comunicaţiei impuse de ei.

Controlul concurenţei. Controlul concurenţei este esenţial pentru calculul paralel deoarece mai multe componente se execută în acelaşi timp. Interacţiunile dorite între componente sunt gestionate prin controlul concurenţei astfel încât algoritmul să funcţioneze corect. Există două tipuri de control al concurenţei: centralizat şi descentralizat. Algoritmii paraleli cu control al concurenţei centralizat sunt sincronizaţi. De exemplu, algoritmii pentru calculatoarele SIMD sunt caracterizati de un control centralizat al concurenţei. Cei pentru calculatoare MIMD au în general un control al concurenţei descentralizat.

Granularitatea("grain size") este un parametru calitativ care caracterizează atât sistemele paralele cât şi aplicaţiile paralele. Granularitatea aplicaţiei se defineşte ca dimensiunea minimă a unei unităţi secvenţiale dintr-un program, exprimată în număr de instrucţiuni. Prin unitate secvenţială se înţelege o parte program în care nu au loc operaţii de sincronizare sau comunicare cu alte procese. Fiecare flux de instrucţiuni are o anumită granularitate. Granularitatea aplicaţiei se defineşte ca valoarea minimă a granularităţii pentru activităţile paralele ale componentelor(proces, thread, task).

Granularitatea unui algoritm poate fi aproximată ca fiind raportul dintre timpul total calcul şi timpul total de comunicare.

Pentru un sistem dat, există o valoare minimă a granularităţii aplicaţiei, sub care performanţa scade semnificativ. Această valoare de prag este cunoscută ca şi granularitatea sistemului respectiv. Justificarea constă în faptul că timpul de *overhead* (comunicaţii, sincronizări, etc.) devine comparabil cu timpul de calcul paralel. De aceea, este de dorit ca un calculator paralel să aibă o granularitate mică, astfel încât să poată executa eficient o gamă largă de programe. Pe de altă parte, este dezirabil ca programele paralele să fie caracterizate de o granularitate mare, astfel încât să poată fi executate eficient de o diversitate de sisteme.

Există totuşi şi clase de aplicaţii cu o valoare a granularităţii foarte mică, dar care se execută cu succes pe arhitecturi specifice. Este cazul aplicaţiilor sistolice, în care în general o operaţie este urmată de o comunicaţie. Aceste aplicaţii impun însă o structură a comunicaţiilor foarte regulată şi comunicaţii doar între noduri vecine (Secţiunea 4.8).

Gradul de paralelism (DOP) al unui algoritm este dat de numărul de operaţii care pot fi executate simultan. Similar cu granulaţia sistemelor paralele (fină–număr mare de procesoare, medie, brută–număr mic de procesoare) se poate defini şi ***granulaţia*** unui algoritm care poate fi la fel: fină, medie, brută, în funcţie de numărul de componente care sunt construite de acel algoritm.

Scalabilitatea măsoară modul în care se schimbă performanţa unui anumit algoritm în cazul în care sunt disponibile mai multe procesoare. Un indicator important pentru aceasta este numărul maxim de procesoare care pot fi folosite pentru rezolvarea unei probleme. În cazul folosirii unui număr mic de procesoare, un program paralel se execută mult mai încet decât un program secvenţial. Diferenţa poate fi atribuită comunicaţiilor şi sincronizăriilor care nu apar în cazul unui program secvenţial. Pe măsură ce numărul de procesoare creşte, creşte şi performanţa; dar la un moment dat creşterea devine nesemnificativă, chiar nulă, odată cu creşterea numărului de procesoare. Graficul din Figura 1.17 arată cum se modifică performanţa la creşterea numărului de procesoare. Adăugarea de procesoare nu determină nici o creştere a performanţei după depăşirea unui anumit prag,

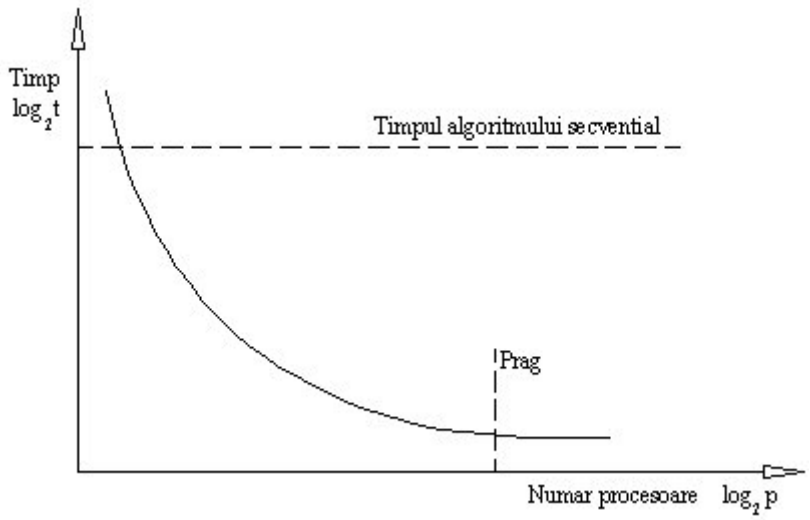

Figura 1.17: Dependenţa performanţei de numărul de procesoare.

datorită faptului că nu există suficient calcul de executat pentru a ţine toate procesoarele ocupate. Numărul minim de componente pentru o anumită partiţionare, poate fi de asemenea un indicator important.

Structura comunicaţiei. Algoritmii paraleli pot fi clasificaţi de asemenea şi în funcţie de structura comunicaţiei pe care ei o necesită. Evident această structură trebuie să fie mapată pe o topologie a unei reţele de interconectare existentă.

De exemplu, dacă algoritmul presupune comunicaţia pe diagonală între componentele organizate ca o matrice de procese, acestea pot fi mapate direct pe o reţea de tip grilă, iar pentru o comunicaţie se vor folosi doi paşi. Se foloseşte problema scufundării topologiilor.

Pentru foarte mulţi algoritmi reţeaua de calcul corespunzătoare este de tipul unui arbore binar şi prin urmare structura de comunicaţie a acestor algoritmi este de tip arbore binar. Reţeaua de interconectare arbore binar poate fi folosită în acest caz.

Există situaţii când structura de comunicaţie a unui algoritm conduce la realizarea unor reţele de interconectare speciale, astfel încât este dificil ca structura de comunicaţie să se mapeze direct pe o reţea de interconectare clasică. Este cazul algoritmului de sortare bitonică datorat lui Batcher care a condus la crearea reţelei amestecare perfectă.

Există şi situaţia inversă când la construcţia algoritmului paralel pentru o problemă dată se are în vedere adaptarea lui pentru un anumit tip de reţea de interconectare, prin urmare asigurarea unei anumite structuri de comunicaţie.

1.6 Modelul standard PRAM

Rolul esenţial al unui model de calcul este să furnizeze o maşină abstractă care să simplifice dezvoltarea şi specificarea programelor.

Modelul RAM este modelul clasic folosit de către calculatoarele seriale care simulează

calculatorul clasic von Neumann. Din păcate, pentru calculul paralel nu există un singur model universal datorită faptului că arhitecturile paralele prezintă diferenţe esenţiale între ele şi pentru că performanţa algoritmilor paraleli depinde foarte mult de detaliile arhitecturii pe care se implementează. Cel mai cunoscut model de calcul paralel şi probabil cel mai vechi este modelul PRAM ("Parallel Random Access Machine").

PRAM modelează calculul pentru sistemele cu memorie comună, dar s-au făcut extinderi şi pentru modelarea arhitecturilor care se bazează pe transmiterea de mesaje.

Un P-procesor PRAM sau PRAM de dimensiune P este definit de un număr de P programe inalterabile, o memorie comună şi P procesoare, ce pot avea şi memorie locală. Accesul la memoria comună poate fi făcut de oricare dintre procesoare într-un pas-program. Un procesor, memoria sa locală şi programul pe care îl execută formează un RAM.

Un P-procesor PRAM execută P instrucţiuni într-un pas, câte una din fiecare program. Acest model corespunde clasei MIMD, cu diferenţa că PRAM este un model sincron.

Pseudocodul clasic, utilizat de modelul RAM este utilizat cu mici diferenţe şi pentru modelul PRAM, cu interpretarea că fiecare citire şi scriere este în memoria globală. În plus, se foloseşte o instrucţiune nulă – pentru a preciza când un procesor este inactiv, şi instrucţiunea `for... in parallel do...end for` – pentru a preciza execuţia în paralel a instrucţiunilor din blocul `for`.

Complexitatea calculului este dată de numărul de procesoare necesare pentru al executa şi de numărul de paşi globali. Acurateţea evaluării costurilor nu este totuşi perfectă, datorită faptului că unitatea de timp pentru accesul la memorie, depinde de numărul total de accesări şi de şablonul pe care acestea îl urmează.

Se poate considera diferenţierea dintre accesurile la memoria locală şi cea globală prin adăugarea a două operaţii: `global_read` şi `global_write` pentru citirea, respectiv scrierea în/din memoria globală(partajată). Această versiune are dezavantajul distrugerii unităţii algoritmului paralel, dar în schimb aduce avantajul explicitării comunicaţiei interprocesor şi permite o evaluare a performanţei mai rafinată.

Exemplul 1.3 (Adunarea a doi vectori) Fie vectorii A şi B cu n elemente.
Se cere calcularea vectorului suma C.

for $i = 0, n-1$ **in parallel do**
  global_read(A[i], a);
  global_read(B[i], b);
  $c \leftarrow a + b$;
  global_write(c, C[i])
end for

O distincţie importantă între diversele variante PRAM se realizează pe baza existenţei sau nu a posibilităţii de a se efectua accesuri simultane, în scriere sau citire, la aceeaşi locaţie de memorie. În cadrul modelului original s-a adoptat un protocol standard *"readers-writers"*, în care o locaţie poate fi citită de mai multe fluxuri de instrucţiuni

simultan, dar scrierea o execută un singur flux (maşina CREW – Concurrent Read Exclusive Write). Modelul cel mai restrictiv, dar şi singurul care are corespondenţi reali, este EREW P-RAM (EREW - Exclusive Read Exclusive Write), care permite ca o locaţie să fie citită sau scrisă de către un singur flux de instrucţiuni, la un moment dat de timp. Pe de altă parte, CRCW PRAM (CRCW - Concurrent Read Concurrent Write) permite orice tip de acces. Versiunea CRCW adaugă problema rezolvării scrierilor simultane: ce se va obţine de fapt? Printre variantele existente putem menţiona:

- "common CRCW PRAM" pentru care o scriere se realizează doar dacă toate procesoarele încercă să scrie aceeaşi valoare,
- "arbitrary CRCW PRAM" pentru care se alege în mod arbitrat un procesor dintre cele care încearcă să scrie şi se realizează scrierea respectivă, şi
- "priority CRCW PRAM" pentru care se alege procesorul cu cel mai mare (sau mic) indice dintre cele care încearcă să scrie şi se realizează scrierea respectivă.

Există un rezultat important, datorat lui Vishkin [168] prin care se arată că aceste modele pot fi efectiv simulate de modelul EREW.

Teorema 1.1 *Dacă un algoritm pe un model CRCW se execută într-un timp α folosind β procesoare, atunci el poate fi simulat pe un model EREW cu β procesoare într-un timp $O(\alpha \log_2^2 n)$ (n=dimensiunea datelor). Memoria RAM va creşte cu un factor de $O(\beta)$.*

O strategie posibilă de proiectare a algoritmilor paraleli folosind modelul PRAM este acela de a studia comparativ mai multe variante distincte, pentru modelele EREW, CREW şi CRCW. Dacă varianta CRxx este mai bună, citirea concurentă poate fi simulată cu un algoritm de tip *broadcasting* (difuzare) bazată pe un calcul de tip arbore binar, care are complexitatea $O(\log_2 P)$.

Pentru sistemele cu transfer de mesaje s-a definit modelul MPRAM (Message Passing RAM). Acesta este constituit din P programe inalterabile, P memorii, P procesoare şi o reţea de interconectare. Fiecare RAM este plasat într-un nod al grafului care modelează reţeaua de interconectare. RAM-urile care comunică direct se numesc vecini. Gradul unui nod este egal cu numărul de vecini, iar gradul reţelei se determină pe baza evaluării funcţiei $max(grad\_nod)$. În setul de instrucţiuni se adaugă SEND_To_Neighbor şi RECEIVE_From_Neighbor. O pereche de noduri vecine comunică prin intermediul unui canal bidirecţional. Pentru acest model este importantă definirea timpului de comunicaţie între două procesoare. De exemplu, dacă procesoarele sunt vecine şi canalul e liber, se poate folosi ecuaţia următoare:

$$T_{com} = t_{start} + wt_{word}$$

unde t_{start} reprezintă timpul necesar iniţierii operaţiei, iat t_{word} timpul consumat pentru transmiterea unui cuvânt din cele w cuvinte ale mesajului.

Se poate considera un model PRAM-SIMD, în care toate componentele execută aceeaşi instrucţiune în fiecare pas-program şi care ar corespunde masivelor de procesoare.

Modelul PRAM impune precizarea detaliată a descompunerii calculului pe procese, a gestiunii memoriei partajate şi a întregii comunicaţii. Datorită complexităţii foarte mari

a specificării unui astfel de program, în general, algoritmii PRAM dezvoltaţi sunt de tip SIMD, astfel încât fiecare procesor execută la fiecare pas aceeaşi operaţie.

Acest model nu furnizează nici o abstractizare pentru descompunere, comunicaţie şi sincronizare. Nu furnizează, de asemenea, nici o metodologie de dezvoltare a softwareului.

Unul din avantajele majore ale acestui model constă în faptul că permite evaluarea costurilor algoritmilor paraleli şi de aceea teoria complexităţii algoritmilor paraleli se bazează pe acest model.

1.7 Măsurarea performanţei algoritmilor paraleli

Evaluarea unui program paralel trebuie să ţină seama de arhitectura (modelul) sistemului de calcul, ceea ce înseamnă că este posibil ca algoritmul ales să fie cel mai bun pentru o anumită maşină, dar să nu fie cel mai bun algoritm paralel pentru problema dată. De asemenea, este posibil, ca pentru un volum mic de date să existe un algoritm paralel mai bun, iar pentru un volum mare de date, un altul. De aceea, pentru compararea riguroasă a variantei paralele cu cea serială, trebuie să se precizeze modelul de calcul paralel, să se aleagă algoritmul serial cel mai bun şi să se indice dacă există condiţionări ale performanţei algoritmului datorită volumului de date.

Dacă în cazul algoritmilor secvenţiali performanţa este măsurată în termenii complexităţiilor timp şi spaţiu, în cazul algoritmilor paraleli se folosesc şi alte măsuri ale performanţei, care au în vedere toate resursele folosite. Elementele de procesare folosite sunt, în cazul programării paralele, o resursă importantă şi prin urmare şi complexitatea elementelor de procesare trebuie să fie luată în considerare.

În general, într-un studiu al performanţei algoritmilor paraleli se au în vedere următorii factori:

- aritmetica operaţiilor
- transferul de date

Câteva remarci pot fi făcute cu privire la măsurarea performanţei programelor paralele în raport cu cele secvenţiale:

1. În calculul paralel, obţinerea unui timp de execuţie mai bun nu înseamnă neapărat utilizarea unui număr minim de operaţii, aşa cum este în calculul serial.
2. Factorul memorie nu are o importanţă foarte mare în calculul paralel. În schimb, o resursă majoră în obţinerea unei performanţe bune a algoritmului paralel o reprezintă numărul de procesoare folosite.
3. Dacă timpul de execuţie a unei operaţii aritmetice este mult mai mare decât timpul de transfer al datelor între două elemente de procesare, atunci întârzierea datorată reţelei este nesemnificativă, dar, în caz contrar, timpul de transfer joacă un rol important în determinarea performanţei programului.

1.7.1 Complexitatea-timp

Timpul este cea mai importantă măsură a performanţei unui algoritmi paralel, deoarece principala motivaţie a folosirii calculului paralel este de a se obţine o îmbunătăţire a timpului de execuţie.

Timpul de execuţie al unui program paralel măsoară perioada care s-a scurs între momentul iniţierii primului proces şi momentul când toate procesele au fost terminate. În timpul execuţiei, fiecare procesor execută operaţii de calcul, comunicaţie, sau este în aşteptare. Prin urmare, timpul total de execuţie, al unui program paralel executat pe un sistem cu pe p procesoare, se poate obţine prin formula:

$$t_p = \left(max\ i : i \in \overline{0, p-1} : T^i_{calcul} + T^i_{comunicatie} + T^i_{asteptare}\right)$$

sau în cazul echilibrării perfecte a încărcării de calcul pe fiecare procesor din formula:

$$t_p = \frac{1}{p} \sum_{0}^{p-1} \left(T^i_{calcul} + T^i_{comunicatie} + T^i_{asteptare}\right)$$

Ultima variantă este, în general, mai folositoare deoarece este mai simplu de calculat timpul global necesar comunicaţiilor şi timpul global de calcul. Dar, timpul de aşteptare datorat blocărilor care pot apare este în general foarte dificil de evaluat.

Ca şi în cazul programării secvenţiale, pentru a dezvolta algoritmi paraleli eficienţi trebuie să putem face o evaluare a performanţei încă din faza de proiectare a algoritmilor. Prin urmare, măsuri de tipul complexităţii-timp trebuie să poate fi evaluate.

Complexitatea-timp pentru un algoritm paralel care rezolvă o problemă P_n, cu dimensiunea n a datelor de intrare, este o funcţie T care depinde de n, dar şi de numărul de procesoare p folosite.

Pentru un algoritm paralel, un pas elementar de calcul se consideră a fi o mulţime de operaţii elementare care pot fi executate în paralel de către o mulţime de procesoare. Complexitatea-timp a unui pas elementar se consideră a fi $O(1)$. Complexitatea-timp a unui algoritm paralel este dată de numărarea atât a paşilor de calcul necesari dar şi a paşilor de comunicaţie a datelor, iar timpul necesar fiecărui pas de comunicaţie depinde de şablonul de comunicaţie folosit.

În general, toate aprecierile timpului de execuţie se fac începând dintr-un moment în care datele se află într-un loc accesibil direct procesoarelor (se ignoră timpul necesar distribuţiei iniţiale a datelor). Aceast lucru poate fi în general corect pentru compararea a doi algoritmi paraleli, dar poate fi incorect în comparaţia între un algoritm paralel şi unul secvenţial.

1.7.2 Acceleraţia

Acceleraţia("speed-up"), notată cu S_p, este definită ca raportul dintre timpul de execuţie al celui mai bun algoritm serial cunoscut, executat pe un calculator monoprocesor şi timpul de execuţie al programului paralel echivalent, executat pe un sistem de calcul

paralel. Dacă se notează cu t_s timpul de execuţie al programului serial, iar t_p timpul de execuţie corespunzător programului paralel, atunci:

$$S_p(n) = \frac{t_s(n)}{t_p(n)}. \tag{1.1}$$

Numărul n reprezintă dimensiunea datelor de intrare, iar p numărul de procesoare folosite.

Această definiţie este uneori dificil de utilizat în practică, pentru că fie nu se cunoaşte timpul de execuţie al celui mai bun algoritm serial, fie varianta paralelă se deosebeşte foarte mult de cea serială, iar comparaţia nu-şi mai are sensul. De aceea, se acceptă mai multe variante de definiţie a acceleraţiei:

- *relativă*, când t_s este timpul de execuţie al variantei paralele pe un singur procesor al sistemului paralel;
- *reală*, când se compară timpul execuţiei paralele cu timpul de execuţie pentru varianta serială cea mai rapidă, pe un procesor al sistemului paralel;
- *absolută*, când se compară timpul de execuţie al algoritmului paralel cu timpul de execuţie al celui mai rapid algoritm serial, executat de procesorul serial cel mai rapid;
- *asimptotică*, când se compară timpul de execuţie al celui mai bun algoritm serial cu funcţia de complexitate asimptotică a algoritmului paralel, în ipoteza existenţei numărului necesar de procesoare;
- *relativ asimptotică*, când se foloseşte complexitatea asimptotică a algoritmului paralel executat pe un procesor.

Analiza asimptotică (consideră dimensiunea datelor n şi numărul de procesoare p foarte mari) ignoră termenii de ordin mic şi este folositoare în procesul de construcţie al programelor paralele.

Legile acceleraţiei

În condiţii ideale, când calculele sunt distribuite egal celor p procesoare şi nu există operaţii de comunicare şi sincronizare:

$$t_p = \frac{t_s}{p}, \tag{1.2}$$

iar $S_p = p$. Dar şi în aceste condiţii, apelurile funcţiilor sistem fac ca acceleraţia să fie mai mică. Deci,

$$1 \leq S_p \leq p. \tag{1.3}$$

Foarte puţine probleme au o asemenea regularitate şi un asemenea paralelism implicit, care să conducă la o acceleraţie apropiată de cea ideală – liniară.

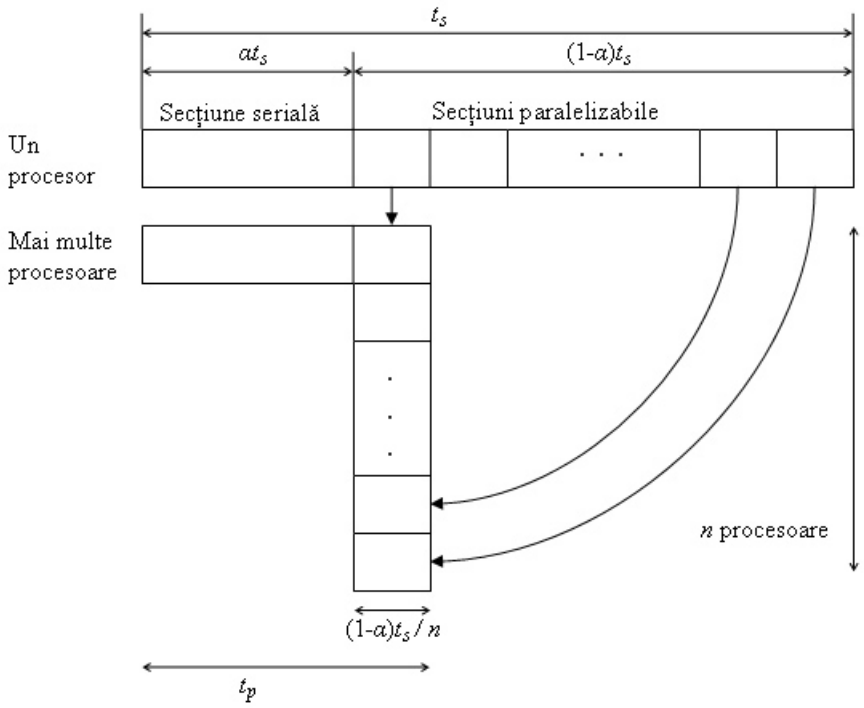

Figura 1.18: Exemplificarea legii lui Amdahl.

O observaţie comună, care se poate face în legătură cu calculul paralel, este că orice algoritm are o componentă secvenţială, care va limita până la urmă acceleraţia care poate fi obţinută pe un calculator paralel. Această observaţie este recunoscută ca **Legea lui Amdahl**[9].

Timpul de calcul al unui program paralel poate fi descompus în două părţi: *o parte pentru calcul paralel* şi *o parte pentru calcul secvenţial*. Indiferent cât de ridicat este gradul de paralelism al părţii paralele a programului, acceleraţia finală va fi limitată de partea serială, care trebuie executată pe un singur procesor.

Teorema 1.2 (Legea lui Amdahl) *Fie $\alpha (0 \leq \alpha \leq 1)$ proporţia operaţiilor din algoritm care se execută secvenţial (fracţia lui Amdahl). Atunci:*

- *Partea serială a algoritmului se execută în timpul αt_s.*
- *Partea paralelă a algoritmului se execută în timpul $\frac{(1-\alpha)t_s}{p}$.*
- *Întregul algoritm se execută în timpul $t_p = \alpha t_s + \frac{(1-\alpha)t_s}{p}$.*
- *Acceleraţia relativă este $RS_p = (\alpha + \frac{1-\alpha}{p})^{-1}$ care nu poate depăşi α^{-1} (Legea lui Amdhal).*

Chiar dacă avem un număr nelimitat de procesoare, acceleraţia maximă este limitată de $1/\alpha$. De exemplu, dacă componenta secvenţială reprezintă 5%, atunci acceleraţia maximă care poate fi atinsă este 20.

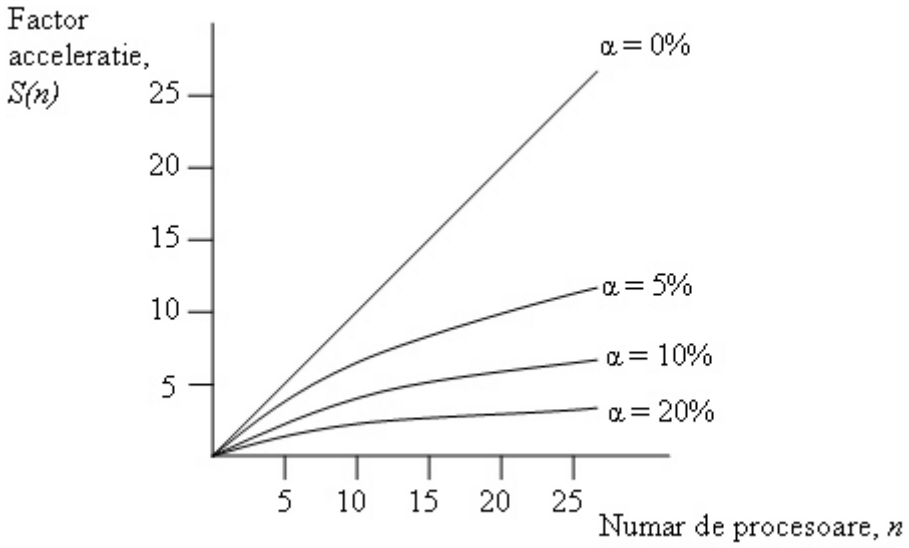

Figura 1.19: Acceleraţia în funcţie de numărul de procesoare.

Legea lui Amdahl este relevantă în special atunci când construirea variantei paralele se face prin paralelizare incrementală. Există probleme pentru care α depinde de n şi poate descreşte odată cu creşterea lui n.

Lee [107] a extins rezultatul anterior, considerând că odată cu variaţia parametrului i=număr de procesoare, care ia valori în domeniul $[1, p]$, se modifică şi proporţia din program, q_i, care poate fi executată de i procesoare – paralelizată. Astfel acceleraţia este:

$$RS_p = \frac{\sum_{i=1}^{p} q_i t_s}{\sum_{i=1}^{p} \frac{q_i}{i} t_s},$$

iar dacă $q_i = 1/p, \forall i \in [1, p]$, se obţine:

$$RS_p \leq \frac{p}{\log_2 p}.$$

Rezultatul evidenţiază faptul că acceleraţia depinde nu numai de natura aplicaţiei ci şi de numărul de procesoare.

Un alt rezultat cu privire la această problemă este **ecuaţia Gustafson-Barsis**[108]. Ei au plecat de la ipoteza conform căreia timpul de execuţie al unui program paralel cu paralelism de date poate fi normalizat, de exemplu, la valoarea 1:

$$t_p = 1.$$

Când se execută acest program pe un procesor, avem:

$$t_1 = \alpha + p(1 - \alpha) = (\alpha + p(1 - \alpha))t_p,$$

Rezultă că acceleraţia relativă este:

$$RS_p = \alpha + p(1 - \alpha).$$

În acest caz $RS_p = O(p)$, ceea ce înseamnă că odată cu folosirea mai multor procesoare, acceleraţia relativă va creşte proporţional.

1.7.3 Eficienţa

Acceleraţia este o măsură a reducerii timpului de execuţie, dar nu indică nimic despre cât de optim sunt folosite cele p procesoare. Eficienţa este un parametru care măsoară gradul de folosire a procesoarelor.

Eficienţa este definită ca fiind acceleraţia împărţită la p:

$$E = \frac{S_p}{p}. \tag{1.4}$$

Din ecuaţia 1.3 se deduce că valoarea eficienţei este întotdeauna subunitară.

Overhead. Denumirea generică a factorilor care micşorează eficienţa este de *overhead*. Este un nume prin care se au în vedere întârzierile datorate încărcării inegale a procesoarelor, a comunicaţiilor, etc. Prezentăm în continuare câteva tipuri generale de overhead [51]:

- Overhead *algoritmic*, datorat părţilor intrinsec secvenţiale ale algoritmului – părţi care nu se pot paraleliza, sau datorat unor operaţii suplimentare efectuate în algoritmul paralel faţă de cel mai rapid algoritm secvenţial;
- Overhead datorat *încărcării inegale* a procesoarelor – cazul ideal este cel în care calculele sunt repartizate uniform procesoarelor, astfel încât fiecare să aibă acelaşi număr de operaţii de executat – dacă unele procesoare vor avea mai mult de lucru decât altele este evident că timpul total de execuţie va creşte;
- Overhead datorat *comunicaţiei* – orice comunicaţie contribuie la ineficienţa algoritmului;
- Overhead *software* – acesta apare datorită repetării unor operaţii ce se execută o singură dată în cazul secvenţial, pe fiecare procesor. (Precizăm însă că repetarea unor calcule poate reduce timpul de execuţie, dacă prin aceasta se reduce numărul de comunicaţii.) Acest tip de overhead poate apare şi datorită unor operaţii suplimentare datorate implementării algoritmului – decizii în funcţie de adresa procesorului, aritmetică întreagă repetată pe mai multe procesoare, etc.

1.7.4 Costul şi volumul de lucru

O altă măsură a eficienţei unui program paralel presupune măsurarea muncii totale executate de către maşina paralelă.

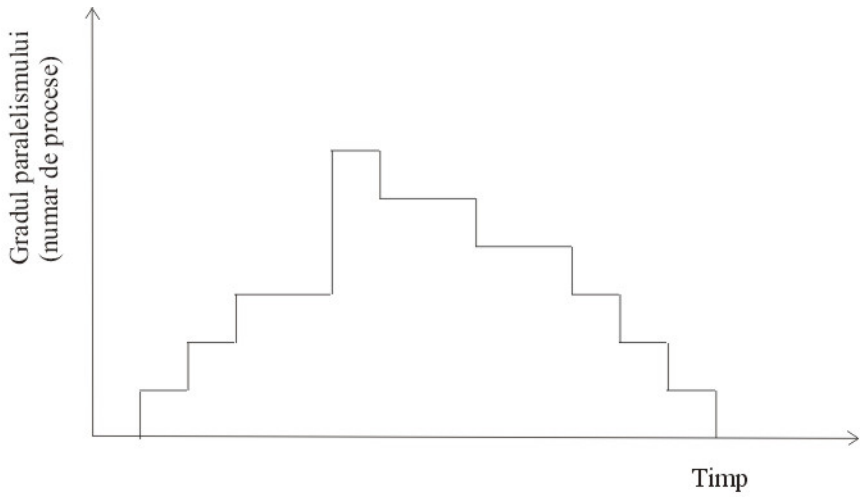

Figura 1.20: Volumul de lucru.

Costul se defineşte ca fiind produsul dintre timpul de execuţie şi numărul maxim de procesoare care se folosesc:

$$C_p(n) = t_p(n) \cdot p. \tag{1.5}$$

Această definiţie este justificată de faptul că orice aplicaţie paralelă poate fi simulată pe un sistem secvenţial, situaţie în care unicul procesor va executa programul într-un timp egal cu $O(C_p(n))$.

O aplicaţie paralelă este *optimă* din punct de vedere al costului, dacă valoarea acestuia este egală, sau este de acelaşi ordin de mărime cu timpul celei mai bune variante secvenţiale; aplicaţia este *eficientă* din punct de vedere al costului dacă $C_p = O(t_s \log p)$.

O măsură mai rafinată este *volumul de lucru ("work")* – W, definit de numărul total de operaţii executate de către procesoarele active. Putem vedea acest volum de lucru ca şi integrala profilului de paralelism al programului – adică numărul de procese active ca o funcţie de timp – Figura 1.20.

Volumul de lucru poate fi evaluat şi ca fiind produsul dintre dimensiunea problemei n şi numărul mediu de operaţii, care se fac asupra unei date de intrare c: $W = n \cdot c$. Această evaluare nu este totuşi precisă, deoarece în afară faptului că se ia în considerare un număr mediu de calcule pe o anumită dată, face şi presupunerea că nu se fac operaţii decât asupra datelor de intrare şi nu şi asupra unor valori obţinute ulterior prin calcul.

1.7.5 Paralelism limitat şi nelimitat

Ca şi pentru algoritmii seriali, pentru algoritmii paraleli, complexitatea-timp se exprimă în funcţie de dimensiunea datelor problemei.

Complexitatea-timp a unui algoritm paralel depinde de tipul sistemului de calcul folosit, dar şi de numărul de procesoare disponibile. De aceea, atunci când este dată complexitatea-timp a unui algoritm paralel este indicat să fie dat numărul maxim de

procesoare folosite, ca o funcţie de dimensiunea datelor problemei. Aceasta este referită ca fiind *complexitatea-procesor*. Este posibil, de asemenea, de a exprima complexitatea-timp ca funcţie şi de numărul de procesoare.

Paralelismul limitat consideră că există un număr fix $p(p > 1)$ de procesoare. În contrast, paralelismul nelimitat exprimă situaţia în care se presupune că există un număr nelimitat de procesoare disponibile.

Din punct de vedere practic, algoritmii bazaţi pe paralelismul limitat sunt de preferat. Este mai realist de presupus că numărul de procesoare disponibile este limitat. Un algoritm paralel implementat pe un model de calcul cu P procesoare este numit P-algoritm. Dacă un P-algoritm paralel necesită $T(n)$ paşi paraleli pentru o problemă de dimensiune n, atunci el este P-calculabil în timpul $T(n)$.

Algoritmii paraleli bazaţi pe paralelism nelimitat, au în vedere în general o margine polinomială, pentru numărul de procesoare, exprimată în funcţie de dimensiunea datelor problemei (de exemplu $O(n)$, $O(n^2)$). Dar această limită poate deveni, pentru dimensiuni mari ale problemelor, un număr impracticabil de mare. Totuşi, algoritmii bazaţi pe paralelism nelimitat au interes teoretic mare, deoarece ei exprimă limitele pentru calculul paralel şi furnizează o înţelegere bună a complexităţii intrinseci a problemei. Complexitatea-timp a unui algoritm bazat pe paralelism nelimitat reflectă caracteristicile problemei.

În practică, algoritmii paraleli bazaţi pe paralelism nelimitat pot deveni utili, dacă pot fi transformaţi în P-algoritmi paraleli. Există două metode pentru o asemenea transformare. Una se bazează pe descompunerea problemei şi cealaltă pe descompunerea algoritmului. Considerăm un algoritm paralel A pentru o problemă P_n de dimensiune n care o rezolvă în timpul $T_1(n)$ folosind $p_1(n)$ procesoare. Trebuie construit un nou algoritm B care rezolvă problema P_n în timpul $T_2(n)$ folosind $p_2(n)$ procesoare unde $p_2(n) < p_1(n)$. Descompunerea problemei presupune împărţirea problemei P_n în subprobleme de dimensiune $m < n$ şi aplicarea în mod repetat a algoritmului A pentru fiecare subproblemă. Descompunerea algoritmului presupune descompunerea fiecărui pas paralel în subpaşi care pot fi executaţi de un număr mai mic de procesoare. Principiul lui Brent enunţat în secţiunea următoare reflectă impactul pe care îl poate avea micşorarea numărului de procesoare asupra timpului de execuţie.

1.7.6 Teorema lui Brent

Această teoremă precizează formal anumite elemente euristice legate de timpul necesar executării în paralel a anumitor calcule pe reţele de calcul. Este legată de reducerea numărului de procesoare necesare execuţiei unui program paralel astfel încât să se păstreze ordinul de mărime al complexităţii-timp.

Mai întâi este necesară o definiţie riguroasă a reţelei de calcul.

Definiţia 1.3 *O reţea de calcul este un graf orientat aciclic ale cărui noduri sunt împărţite în trei mulţimi:*

Noduri de intrare: *noduri cu gradul interior nul.*

Noduri de ieşire: *noduri cu gradul exterior nul.*

Noduri interioare: *noduri cu grad interior şi exterior nenul.*

Fiecare nod interior este etichetat cu o operaţie elementară.

Numărul de muchii incidente unui nod interior este numit *"fan-in"*, iar numărul de muchii care ies *"fan-out"*. Maximul acestor numere este numit numărul *"fan-in"*, respectiv numărul *"fan-out"* al grafului.

Lungimea celui mai lung drum de la nodurile de intrare la cele de ieşire este numită *adâncimea* reţelei de calcul.

Calculul efectuat de o reţea de calcul, pentru o mulţime de date de intrare, este definit de datele care apar în nodurile de ieşire, ca rezultat al următoarei proceduri:

1. Se aplică datele de intrare nodurilor de intrare.
2. Se transmit datele de-a lungul muchiilor. De fiecare dată când se trece printr-un nod interior, se aşteaptă până când datele de pe toate muchiile lui de intrare sosesc şi apoi se realizează operaţia elementară indicată. Rezultatul se transmite pe toate muchiile de ieşire din nod.
3. Procedura se încheie când nu mai există date pe nodurile interioare.

Teorema 1.3 *Fie N o reţea de calcul cu n noduri interioare, cu adâncimea d şi cu numărul fan-in mărginit. Atunci calculele efectuate de N pot fi executate pe un calculator CREW-PRAM cu p procesoare într-un timp $O(\frac{n}{p} + d)$.*

Timpul total depinde de numărul *fan-in* al reţelei – care este inclus în constanta de proporţionalitate.

Demonstraţie: Se poate simula calculul într-un mod direct. Presupunem că avem o structură de date în memoria calculatorului PRAM pentru codificarea unui nod al reţelei şi că aceasta are un câmp care conţine un pointer spre nodul care primeşte rezultatul produs de calculul său, dacă nu este un nod exterior. Dacă este nod exterior atunci acel câmp va fi nul. Definim adâncimea unui nod, v, ca fiind maximul lungimilor drumurilor de la un nod de intrare la v. Simularea se realizează inductiv: simulăm toate calculele care au loc la nodurile de adâncime mai mici decât $k-1$ înaintea celor de adâncime k. Presupunem că sunt n_i noduri interioare ale căror adâncime este exact i. Avem atunci $\sum_{i=1}^{d} n_i = n$. După simularea calculelor de pe nodurile cu adâncimea $k-1$, putem simula calculele de la nodurile de adâncime k, deoarece intrările sunt toate disponibile.

Ordinea calculelor de la nivelul k este irelevantă. Aceasta se datorează definiţiei adâncimii care asigură că ieşirea oricărui nod de adâncime k este intrare a unui nod cu adâncime strict mai mare.

Simularea calculului la nivelul k este după cum urmează:

1. Procesoarele citesc datele din zonele de ieşire ale structurilor de date corespunzătoare nodurilor de adâncime $k-1$.

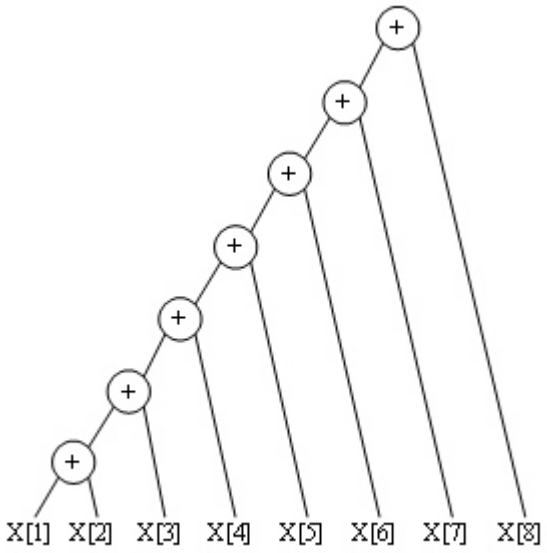

Figura 1.21: Reţea de calcul, cu adâncimea 7, pentru adunarea a 8 numere.

2. Procesoarele execută calculele cerute.

Deoarece sunt n_k noduri cu adâncimea k şi calculele se pot realiza în orice ordine, timpul de execuţie al acestei faze este:

$$\left\lceil \frac{n_k}{p} \right\rceil \leq \frac{n_k}{p} + 1$$

Timpul total de execuţie este astfel:

$$\sum_{i=1}^{d} \left\lceil \frac{n_k}{p} \right\rceil \leq \sum_{i=1}^{d} \left(\frac{n_k}{p} + 1 \right) = \frac{n}{p} + d$$

Teorema se aplică, în general, pentru a modifica algoritmii paraleli astfel încât să folosească mai puţine procesoare.

Principiul de organizare al lui Brent

Presupunem că un algoritm A are proprietatea că toate calculele efectuate de el se exprimă în termenii unei reţele de calcul cu x noduri şi adâncimea d şi are numărul fan-in mărginit. Atunci algoritmul poate fi executat pe un calculator CREW-PRAM cu p procesoare într-un timp $O(\frac{x}{p} + d)$.

Aceste rezultate au consecinţe interesante în ceea ce priveşte relaţia dintre reprezentarea datelor şi timpul de execuţie al unui algoritm.

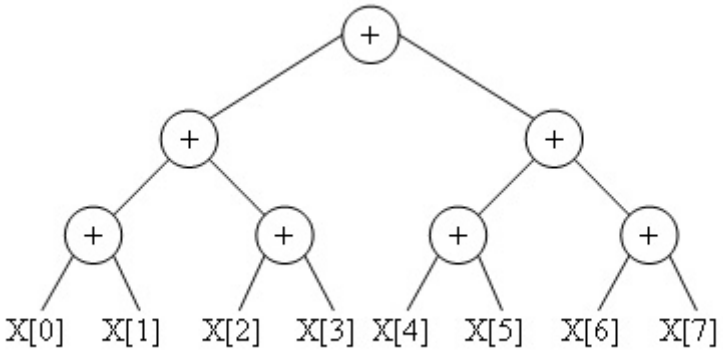

Figura 1.22: Reţea de calcul, cu adâncimea 3, pentru adunarea a 8 numere.

Sarcina de a găsi un algoritm paralel bun într-un asemenea caz poate fi privită ca o problemă de remodelare a reţelei de calcul, astfel încât distanţa de la rădăcină la frunze să fie minimă. Acest proces de remodelare poate să conducă la creşterea numărului total de noduri ale reţelei, astfel încât un asemenea algoritm paralel ar fi ineficient dacă s-ar executa secvenţial.

De exemplu, dacă dorim să adunăm 8 numere $a_1, a_2, \ldots, a_8$, algoritmul secvenţial de bază are reţeaua de calcul prezentată în Figura 1.21.

Această reţea de calcul poate fi remodelată şi se ajunge la reţeaua din Figura 1.22.

Generalizând, dacă avem $n = 2^k$ numere, acestea pot fi adunate în k paşi, prin punerea numerelor în frunzele unui arbore binar total echilibrat - care reprezintă reţeaua de calcul. Dacă considerăm $p = n/2$, deci numărul de procesoare egal cu jumătate din numărul datelor de intrare, complexitatea-timp obţinută este egală cu $k = \log_2 n$. Costul algoritmului în acest caz este $n \log_2 n$ şi prin urmare nu este un algoritm optim din punct de vedere al costului; este doar eficient.

Ca o consecinţă a principiului lui Brent, algoritmul poate fi executat cu $\lceil n/\log_2 n \rceil$ procesoare fără nici o degradare asimptotică a timpului de execuţie. La fiecare nivel sunt de executat $n/2$, $n/4$,..., $n/2^{\log_2 n}$ operaţii. Demonstraţia constă în înlocuirea directă a valorilor concrete pentru x şi p. Dacă $p = \lceil n/\log_2 n \rceil$ atunci complexitatea-timp este tot $O(\log_2 n)$, iar costul este $(n/\log_2 n) * \log_2 n = n$ şi prin urmare s-a obţinut un algoritm optim din punct de vedere al costului.

1.7.7 Clasa problemelor NC

Din analiza algoritmilor paraleli se poate deduce că foarte mulţi algoritmi paraleli (pentru care considerăm paralelism nelimitat) au un timp de execuţie de tip $O(\log^k n)$, (n=dimensiunea datelor). Acesta este, de exemplu, cazul algoritmilor construiţi cu ajutorul tehnicii dublării recursive.

Deoarece acest fenomen este atât de răspândit, N. Pippenger a definit o clasă de probleme numită *NC* [157]. Acestea sunt problemele care pot fi rezolvate pe un calculator

paralel, într-un timp $O(\log^k n)$, folosind un număr polinomial de procesoare. În general, o problemă este numită *paralelizabilă* dacă este din clasa NC. Deoarece orice problemă NC poate fi rezolvată secvenţial într-un timp polinomial (prin simularea unui calculator PRAM de către unul secvenţial), rezultă că $NC \subset P$ (P este clasa problemelor rezolvabile într-un timp polinomial [39]). Întrebarea este dacă $NC = P$, sau altfel spus dacă există probleme *esenţial secvenţiale* ("inherently sequential") – pentru care nu există un algoritm paralel, care este substanţial mai rapid decât cel mai rapid algoritm secvenţial. Această problemă rămâne deschisă, dar presupunerea este că $NC \neq P$, chiar dacă nu s-au găsit exemple care să dovedească aceasta [67].

Pentru problemele NP-complete – probleme pentru care nu se cunoaşte un algoritm secvenţial cu complexitate-timp polinomială, este de aşteptat să nu poată fi găsiţi algoritmi paraleli din clasa NC.

Sumar

În acest prim capitol am prezentat principalele noţiuni legate de calculul paralel, plecând de la clasele de arhitecturi paralele până la parametrii de performanţă ai programelor paralele.

Spre deosebire de calculul serial, unde există un model unic pentru toate sistemele de calcul secvenţial, în cazul calculului paralel avem arhitecturi extrem de diferite care influenţează modelul de calcul. Au fost prezentate diferite clasificări ale acestor arhitecturi, precum şi criteriile lor de performanţă, diferite tipuri de reţele de interconectare şi aspecte legate de comunicaţia în reţele.

Trecând de la sisteme la programe paralele, a fost prezentată o clasificare a algoritmilor paraleli şi apoi modelul standard PRAM. Obţinerea de performanţă este scopul principal al paralelismului şi prin urmare posibilitatea evaluării acesteia este esenţială. Au fost prezentaţi parametrii precum complexitate, acceleraţie, eficienţă şi cost, împreună cu legile corespunzătoare lor.

Capitolul 2

Construcţia programelor paralele

Există două abordări principale pentru obţinerea de programe paralele:

Prima este bazată pe *paralelism implicit*. Această abordare este folosită de către limbajele paralele şi de către compilatoarele de paralelizare. Programatorul nu specifică şi astfel nici nu controlează, planificarea calculelor şi plasamentul datelor.

Cea de-a doua se bazează pe *paralelism explicit*. În acest caz programatorul este responsabil în mare parte de paralelizare prin descompunerea în componente de calcul ("task"-uri), asignarea lor pe procesoare şi stabilirea structurii de comunicaţie. Această abordare este bazată pe presupunerea că programatorul poate analiza cel mai bine cum poate fi exploatat paralelismul pentru o anumită aplicaţie.

Prin experienţă s-a observat că paralelismul explicit conduce la obţinerea unei eficienţe mai ridicate şi această abordare va fi luată în considerare în ceea ce urmează.

După cum am precizat anterior, construcţia unui program paralel presupune descompunerea problemei în componente de calcul individuale, stabilirea necesităţilor de comunicare dintre ele, adaptarea lor pentru implementare pe o arhitectură concretă, etc. Toate acestea pot ajunge la o complexitate deosebit de ridicată şi prin urmare este necesară folosirea unor metodologii care să permită stăpânirea acestei complexităţi. Construcţia algoritmilor paraleli nu poate fi simplu redusă la câteva reţete. Creativitatea şi ingenuozitatea sunt de multe ori necesare. Totuşi putem beneficia de abordări metodice care să restrângă posibilităţile de explorare, să furnizeze mecanisme de evaluare a alternativelor şi să reducă astfel costurile de dezvoltare.

De asemenea, descompunerea calculului nu este întotdeauna evidentă şi de aceea este indicată folosirea unor tehnici şi paradigme care pot direcţiona proiectarea cu scopul de a obţine soluţii eficiente. Majoritatea problemelor au mai multe soluţii paralele. Cea mai bună dintre aceste soluţii poate să difere în mod esenţial de cel mai bun algoritm secvenţial pentru acea problemă.

2.1 Etape în dezvoltarea programelor paralele

Nu există o schemă simplă care să poate fi aplicată pentru dezvoltarea programelor paralele. Totuşi, se poate considera o abordare metodologică care să maximizeze numărul de opţiuni, să furnizeze mecanisme de evaluare a alternativelor şi să reducă costurile necesare revenirii, în cazul luării unor decizii neperformante. O asemenea metodologie de dezvoltare permite programatorului să se concentreze asupra aspectelor independente de arhitectură, cum este concurenţa, în prima fază de dezvoltare, iar analiza aspectelor dependente de maşină să fie amânată pentru finalul procesului de dezvoltare. Vom prezenta în această secţiune o metodologie de dezvoltare a aplicaţiilor paralele propusă de către Ian Foster [62], prin care procesul de dezvoltare este organizat în patru etape distincte: partiţionarea, comunicaţia, aglomerarea şi asignarea (maparea) (PCAM).

Această metodologie defineşte aceste etape ca şi linii directoare în procesul de realizare a programelor paralele şi nu neapărat ca nişte etape stricte de dezvoltare (Figura 2.1).

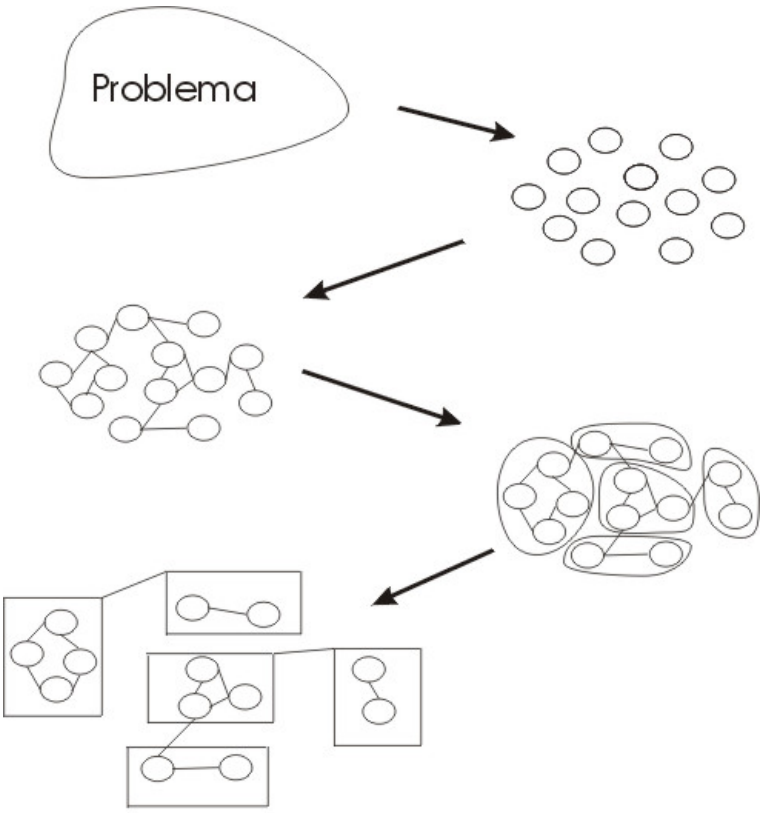

Figura 2.1: Etape în dezvoltarea programelor paralele.

2.1.1 Partiţionarea

Problema partiţionării are în vedere împărţirea problemei de programare în componente de calcul care se pot executa concurent. Aceasta nu implică o divizare directă a programului într-un număr de componente egal cu numărul de procesoare disponibile. Cele mai importante scopuri ale partiţionării sunt legate de *scalabilitate*, abilitatea de a *ascunde întârzierea(latency)* datorată reţelei şi realizarea unei *granularităţi* cât mai mari.

Sunt de preferat partiţionările care furnizează mai multe componente decât procesoare, astfel încât să se permită *ascunderea întârzierii*. Întârzierea este definită de timpul necesar unui mesaj pentru a traversa o arhitectură şi a ajunge la destinaţie. O componentă va fi blocată, sau va aştepta, până când mesajul care conţine informaţia dorită va ajunge. Dacă există şi alte componente de calcul disponibile, procesorul poate continua calculul; acest concept se numeşte *multiprogramare* şi corespunde execuţiei concurente pe acelaşi calculator.

Prin partiţionare se urmăreşte descompunerea problemei în componente de calcul cât mai fine şi concurente. Rezultatul este un *graf de dependenţă (execuţie)* (similar reţelei de calcul definită în Secţiunea 1.7.6) ale cărui noduri sunt componentele de calcul, iar arcele pot exprima relaţii de precedenţă între componentele de calcul sau canale de comunicaţie. Relaţia de ordine între două componente de calcul a_1 si a_2 este indicată de dependenţa de date [8]:

- *dependenţa de curs("flow dependency")*– data de intrare pentru a_2 este produsă ca dată de ieşire de către a_1;
- *antidependenţa("antidependency")* – a_2 urmează după a_1, iar ieşirea lui a_2 se suprapune cu intrarea lui a_1;
- *dependenţa de ieşire("output dependency")* – a_1 şi a_2 produc (scriu) aceeaşi dată de ieşire;
- *dependenţa de I/E (" I/O dependency")* – a_1 şi a_2 accesează simultan aceeaşi resursă (de exemplu fişier).

O relaţie de condiţionare suplimentară este dictată de ordinea de execuţie care stabileşte dacă o componentă de calcul nu poate începe înaintea terminării altei componente de calcul.

Considerând un set de componente S_i, notăm cu $I(S_i)$ mulţimea variabilelor citite (date de intrare) de către S_i, iar cu $O(S_i)$ mulţimea variabilelor scrise (date de ieşire) de către S_i. Două componente S_i şi S_j cu $(j > i)$ sunt independente şi respectă condiţiile lui Bernstein [17] dacă:

$$\begin{array}{ll} O(S_i) \cap O(S_j) = \emptyset & \text{independenţă de ieşire;} \\ I(S_j) \cap O(S_i) = \emptyset & \text{independenţă de flux;} \\ I(S_i) \cap O(S_j) = \emptyset & \text{anti-independenţă.} \end{array}$$

În acest caz execuţia celor două componente produce aceleaşi rezultate, indiferent dacă sunt executate secvenţial, concurent în orice ordine, sau în paralel.

Există două strategii principale de partiţionare: descompunerea domeniului de date şi descompunerea funcţională. În funcţie de acestea putem considera aplicaţii paralele bazate pe descompunerea domeniului de date – *paralelism de date* şi aplicaţii paralele bazate pe descompunerea funcţională. Cele două tehnici pot fi folosite însă şi împreună; de exemplu se începe cu o descompunere funcţională şi după identificarea principalelor funcţii se poate folosi descompunerea domeniului de date pentru fiecare în parte.

Descompunerea domeniului de date

Această tehnică s-a dovedit a fi eficientă pentru o mare varietate de probleme ştiinţifice. Este aplicabilă atunci când domeniul datelor este mare şi regulat. Ideea centrală este de a divide domeniul de date, reprezentat de principalele structuri de date, în părţi care pot fi manipulate independent. Apoi se partiţionează operaţiile, de regulă prin asocierea calculelor cu datele asupra cărora se efectuează. Astfel, se obţine un număr de componente de calcul, definite de un număr de date şi de operaţii. Este posibil ca o operaţie să solicite date de la mai multe componente de calcul. În acest caz sunt necesare comunicaţii.

Datele care se partiţionează sunt în general datele de intrare, dar pot fi considerate de asemenea şi datele de ieşire sau intermediare. Sunt posibile diferite variante, în funcţie de structurile de date avute în vedere. Trebuie considerate în special structurile de date de dimensiuni mari sau cele care sunt accesate în mod frecvent. De asemenea, faze diferite ale calculului pot impune partiţionări diferite ale aceleiaşi structuri de date, caz în care sunt necesare redistribuiri ale datelor. În acest caz trebuie avute în vedere şi costurile necesare redistribuirii datelor.

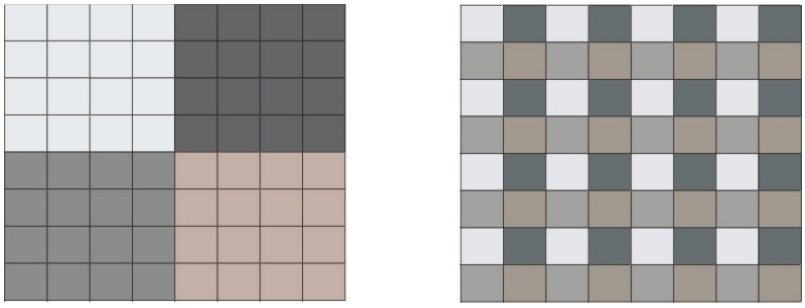

Figura 2.2: Tehnicile de partiţionare a datelor prin – (a)"tăiere"(distribuţie bloc) şi prin – (b) "încreţire" (distribuţie grid). Matricea de dimensiune 8×8 se partiţionează în 4 submatrice - fiecare submatrice este evidenţiată cu o culoare specifică.

Partiţionarea datelor conduce la anumite distribuţii ale datelor per componente de calcul şi de aici implicit şi per procesoare. Există mai multe tehnici de partiţionare a datelor, care pot fi exprimate şi formal prin funcţii definite pe mulţimea indicilor datelor

de intrare cu valori în mulţimea indicilor de procese (vezi Secţiunea 6.2). Cele mai folosite tehnici de partiţionare sunt prin "tăiere" şi prin "încreţire", care corespund distribuţiilor liniară şi ciclică, definite în secţiunea amintită anterior. De exemplu, pentru o matrice A de dimeniune 8×8, partiţionarea sa în $p = 4$ părţi prin tehnica tăierii conduce la partiţionarea arătată de Figura 2.2 (a), iar prin tehnica încreţirii conduce la partiţionarea arătată de Figura 2.2 (b).

Tehnica este folosită în special în cazul în care datele sunt vectori sau matrici, aşa cum se întâmplă în cazul algebrei liniare, de exemplu.

Descompunerea domeniului de date este tehnica de partiţionare cea mai des folosită şi poate conduce la aplicaţii paralele cu granulaţie foarte fină (număr mare de componente de calcul).

Descompunerea funcţională

Descopunerea funcţională este o tehnică de partiţionare folosită atunci când aspectul dominant al problemei este funcţia, sau algoritmul, mai degrabă decât operaţiile asupra datelor. Obiectivul este descompunerea calculelor în componente de calcul cât mai fine. După crearea acestora se examinează cerinţele asupra datelor.

Focalizarea asupra calculelor poate releva uneori o anumită structură a problemei, de unde oportunităţi de optimizare, care nu sunt evidente numai din studiul datelor. În plus, ea are un rol important ca şi tehnică de structurare a programelor.

Această variantă de descompunere nu conduce în general la o granulaţie fină a componentelor de calcul, care se execută în paralel.

Proprietăţi

Putem considera câteva cerinţe care caracterizează o partiţionare bună, care poate deci conduce la realizarea unui program paralel eficient:

- Componentele de calcul obţinute sunt de dimensiuni comparabile, deci se poate obţine o încărcare de calcul echilibrată pe fiecare procesor.
- Scalabilitatea poate fi obţinută. Aceasta înseamnă că numărul de componente de calcul sunt definite în funcţie de dimensiunea problemei; deci creşterea dimensiunii datelor implică creşterea numărului de componente de calcul.
- Întâzierile datorate reţelei de comunicaţii pot fi reduse prin multiprogramare.
- Granularitatea aplicaţiei este suficient de mare astfel încât aplicaţia să poată fi implementată cu succes pe diferite arhitecturi.

2.1.2 Analiza comunicaţiei

Componentele de calcul rezultate în urma partiţionării nu pot fi executate în general independent; ele necesită anumite comunicaţii. Specificarea şi analiza acestor operaţii se face în această fază. Comunicaţia este strâns legată de partiţionare, mai exact modul de partiţionare implică structura comunicaţiei.

Dacă considerăm componentele obţinute la faza de partiţionare ca fiind nodurile unui graf, o comunicaţie între două componente poate fi reprezentată ca fiind o muchie în acel graf. Numărul de comunicaţii între două componente poate fi reflectat în acest graf ca fiind costul muchiei dintre cele două.

În general, pentru problemele care se bazează pe paralelismul de date, cerinţele de comunicaţie sunt mai dificil de evaluat. În acest caz, organizarea comunicaţiei într-un mod eficient este mai dificilă.

În cazul descompunerii funcţionale, cerinţele de comunicaţie se stabilesc mai uşor. Interfeţele dintre funcţii sunt mai bine precizate, deci şi fluxul de date între funcţii se determină într-o manieră directă.

Există diferite tipuri de comunicaţii: locale şi globale, structurate şi nestructurate, statice şi dinamice, sincrone şi asincrone.

Comunicaţiile sunt *locale* dacă fiecare proces comunică cu un număr mic de alte procese, numite *vecini*, şi avem comunicaţii *globale* dacă toate procesele sunt implicate în operaţia de comunicaţie.

Într-o comunicaţie structurată, un proces şi vecinii săi formează o structură regulată, spre deosebire de cazul comunicaţiilor nestructurate când nu se poate stabili o structură între vecini.

O comunicaţie statică implică faptul că vecinii unui proces nu se modifică în timp, iar în cazul unei comunicaţii dinamice, vecinii unui proces se cunosc doar la execuţie şi se pot modifica în timpul execuţiei.

În cazul *comunicaţiei asincrone*, procesul care comunică informaţie continuă execuţia după ce aceasta a fost transmisă la destinaţie, în timp ce procesul care recepţionează mesajul poate suferi o întârziere datorată aşteptării.

Comunicaţia sincronă cere ca atât procesul care trimite, cât şi cel care recepţionează mesajul să fie disponibile până în momentul când transmisia s-a terminat, ambele procese putând apoi să-şi continue lucrul. În felul acesta, nu este necesară stocarea mesajelor (lucru care poate fi necesar în cazul transmiterii asincrone).

2.1.3 Aglomerarea

Algoritmul, rezultat în urma celor două etape precedente, este abstract, în sensul că nu este specializat pentru o execuţie eficientă pe un anumit sistem paralel.

Prin cea de-a treia etapă – aglomerarea – se trece de la abstract la concret prin luarea unor decizii care să conducă la un algoritm eficient pe un sistem dat. Operaţia presupune combinarea componentelor determinate în faza de partiţionare cu scopul de reducere a numărului de comunicaţii şi de asemenea de obţinere a unei granularităţi optime. Deciziile de combinare vor fi luate în funcţie de reţeaua de comunicaţie a sistemului ales şi de granularitatea sistemului, astfel încât să se ajungă la o implementare eficientă. Se analizează, de asemenea, dacă este cazul, replicarea anumitor calcule cu scopul reducerii comunicaţiilor.

Trebuie însă să se urmărească păstrarea echilibrului de calcul şi între componentele nou rezultate în urma acestei faze şi de asemenea păstrarea scalabilităţii aplicaţiei.

Aglomerarea nu presupune obligatoriu obţinerea unui număr de componente egal cu numărul de procesoare. Este indicat, de altfel, ca mai multe componente să fie asignate pe acelaşi procesor, pentru a ascunde întârzierile datorate comunicaţiilor. Dacă un proces este blocat din diferite motive (aşteaptă o informaţie, sau ajungerea la o anumită stare), procesorul poate să execute un alt proces, micşorându-se astfel timpii morţi ai procesoarelor. Totuşi, există cazuri în care este indicată obţinerea unui număr egal de componente cu numărul de procesoare; de exemplu dacă mediul de dezvoltare necesită o aplicaţie SPMD.

Costurile de comunicaţie pot influenţa în mod critic performanţa unui program paralel. Datorită acestui fapt se poate îmbunătăţi performanţa prin reducerea timpului necesar comunicaţiei. Această reducere poate fi obţinută în mod evident prin reducerea volumului de date ce trebuie transmis, dar şi prin transmiterea unui număr mai mic de mesaje, chiar pentru acelaşi volum de date. Aceasta datorită faptului că orice comunicaţie presupune un cost proporţional cu cantitatea de date transmise, dar şi un cost fix de start. Pe lângă costurile de comunicaţie trebuie luate în calcul şi costurile necesare creării componentelor de calcul.

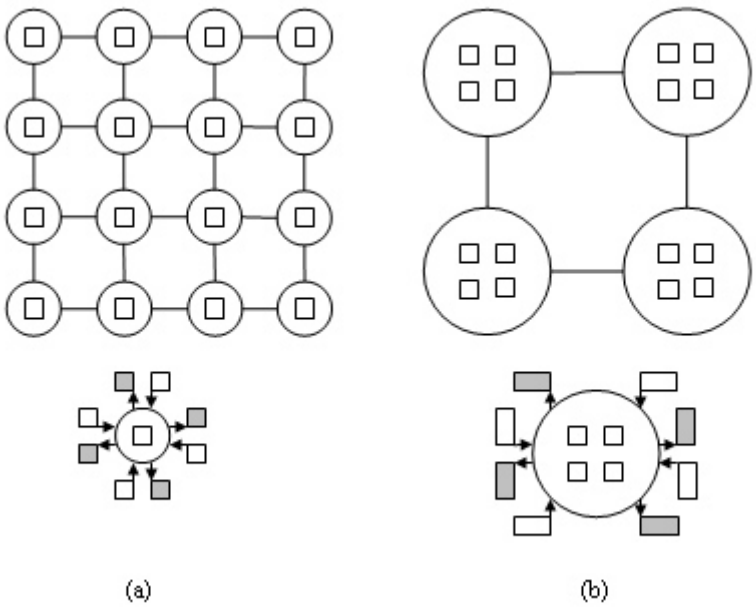

Figura 2.3: Efectul suprafaţă-volum. (a) Grila de componente înainte de aglomerare. (b) Grila de componente după aglomerare.

Dacă numărul de parteneri de comunicaţie ai unei componente de calcul este mic, se poate reduce numărul de operaţii de comunicaţie prin creşterea granularităţii partiţionării, prin combinarea/aglomerarea mai multor componente de calcul într-una singură. Acest efect poartă numele de efect *suprafaţă-volum*. Necesarul de comunicaţii al unei componente de calcul este proporţional cu suprafaţa subdomeniului în care operează, iar

cantitatea calculelor este proporţională cu volumul subdomeniului. Pentru o problemă bidimensională, "suprafaţa" variază proporţional cu dimensiunea problemei, în timp ce "volumul" variază proporţional cu pătratul dimensiunii problemei. Deci cantitatea de comunicaţii realizată de o unitate de calcul (raportul *comunicatie/calcul*) descreşte pe măsură ce numărul de componente de calcul creşte. Efectul "suprafaţă-volum" este vizibil în special în cazul în care partiţionarea s-a făcut pe baza descompunerii domeniului de date.

Ca exemplu, putem lua cazul calculului diferenţei finite bidimensionale Jacobi. Considerăm o grilă bidimensională de $n \times n$ puncte. Prin acest calcul valoarea fiecărui punct este actualizată prin următorul calcul:

$$X_{i,j}^{t+1} = \frac{4X_{i,j}^{t} + X_{i-1,j}^{t} + X_{i+1,j}^{t} + X_{i,j-1}^{t} + X_{i,j+1}^{t}}{8}$$

În Figura 2.3 este evidenţiată o grilă cu $4 \times 4 = 16$ puncte. Înainte de etapa de aglomerare – Figura 2.3(a), avem 16 componente de calcul, fiecare dintre acestea comunicând cu vecinii săi o valoare de intrare şi una de ieşire. După aglomerare – Figura 2.3(b), realizată prin gruparea a câte 4 componente, uniform pe cele două dimensiuni, se observă că numărul comunicaţiilor fiecărei componente se dublează, în timp ce calculul efectuat de fiecare componentă creşte de 4 ori.

O consecinţă a efectului suprafaţă-volum este că descompunerile pe mai multe dimensiuni sunt în general mai eficiente, pentru că reduc suprafaţa (aria de comunicaţie) corespunzătoare unui volum dat (calcul). Deci, din punct de vedere al eficienţei, este bine ca creşterea granularităţii să se facă prin aglomerarea componentelor pe toate dimensiunile decât prin reducerea numărului de dimensiuni.

Pentru problemele fără o structurare a comunicaţiei este foarte dificilă găsirea unei strategii de aglomerare eficientă. Mărirea granularităţii nu este în general o problemă simplă mai ales dacă graful de execuţie al aplicaţiei este complex. O abordare ar consta în analiza comunicaţiilor. Se ordonează descrescător canalele, după cantitatea de date transmise. Apoi, se grupează două câte două componentele care comunică cel mai mult, fără a se ajunge la mai puţine componente decât procesoare. Când nu mai există posibilitatea grupării, se compară granulele între ele. Dacă vor fi diferenţe prea mari, sistemul va fi dezechilibrat şi soluţia va trebui abandonată, caz în care se vor investiga alte posibilităţi de grupare.

Replicarea calculelor poate de asemenea reduce, în anumite cazuri, costurile de comunicaţie. De exemplu, pentru cazul adunării a n numere folosind n procese se poate cere ca în final fiecare proces să conţină suma finală. O abordare posibilă poate folosi un calcul de tip arbore (Sectiunea 4.2) urmat de o comunicaţie de tip difuzare ("broadcast") realizată tot pe baza arborelui de calcul. Pentru a elimina etapa de difuzare se pot replica anumite calcule prin care se ajunge la o reţea de calcul de tip fluture (Secţiunea 1.3.1), prin care timpul de calcul se reduce la jumătate.

Dacă din analiza comunicaţiei sunt depistate componente care nu pot fi executate în paralel, acestea se vor grupa, formând o singură componentă.

Această fază nu mai este legată de proiectarea abstractă a aplicaţiei, ci are în vedere o anumită arhitectură pe care se doreşte implementarea.

2.1.4 Maparea

După ce o problemă este partiţionată, componentele trebuie să fie asignate, sau altfel spus *mapate* pe procesoare, pentru execuţie.

Un important aspect al acestei activităţi este menţinerea *localităţii*: plasarea componentelor problemei care schimbă informaţie, cât mai aproape posibil, pentru a reduce costul comunicaţiilor. Va avea loc, o operaţie de scufundare a grafului de dependenţă în graful reţelei de interconectare a arhitecturii repective.

Deoarece maparea are în vedere folosirea unei arhitecturi particulare, trebuie să fie cunoscute anumite detalii ale acesteia. Pentru simplificare, presupunem existenţa a n calculatoare, numărate secvenţial începând cu 0 până la $n-1$. Problema mapării constă din a găsi o funcţie s definită pe mulţimea componentelor de calcul, cu valori în mulţimea procesoarelor, semnificaţia relaţiei $s(S_j) = i$ fiind: componenta S_j se va executa pe procesorul al i-lea.

Problema mapării este cunoscută ca fiind NP-completă, adică nu există un algoritm cu complexitate polinomială, care să asigure plasarea optimă a componentelor pe procesoare.

Mulţi algoritmi construiţi folosind tehnicile de descompunere a domeniului de date, sunt formaţi dintr-un număr de componente de calcul de mărime egală între care comunicaţia locală şi globală este structurată. În asemenea cazuri, maparea este directă. Mapăm procesele astfel încât să minimizăm comunicaţia interprocesor; este posibil de asemenea, să mapăm mai multe componente pe acelaşi procesor.

Pentru algoritmii bazaţi pe descompunere funcţională cu încărcări de calcul diferite pe procese şi/sau cu comunicaţii nestructurate, strategiile de mapare nu sunt atât de directe. Totuşi se pot aplica algoritmi de echilibrare a încărcării ("load balancing") – în special algoritmi euristici – pentru a identifica strategii eficiente. Timpul consumat execuţiei acestor algoritmi trebuie să ducă la beneficii considerabile ale timpului de execuţie global. Metodele probabilistice de echilibrare a încărcării tind să conducă la programe cu timpi de întârziere mai mici decât cei bazaţi pe structura aplicaţiei.

Cele mai complexe probleme sunt acelea pentru care numărul de componente şi cantitatea de calcul şi comunicare per componentă se schimbă dinamic în timpul execuţiei. În cazul problemelor dezvoltate folosind tehnici de descompunere a domeniului de date, se pot folosi strategii dinamice de echilibrare a încărcării ("dynamic load-balancing"), în care un algoritm de echilibrare a încărcării se execută periodic.

În cazul folosirii descompunerii funcţionale, poate fi folosit un algoritm de distribuire a componentelor de calcul ("task-scheduling"). O componentă manager centralizat sau distribuit are sarcina de a reţine şi a distribui componentele. Cel mai complicat aspect al algoritmului de distribuire a proceselor este strategia folosită pentru alocarea componentelor procesoarelor ("workers"). În general strategia reprezintă un compromis între cerinţele pentru operaţiile independente – de a îmbunătăţii încărcarea de calcul şi pentru informaţiile globale ale stării calculului – de a reduce costurile de comunicaţie.

Sumar

Paralelizarea aplicaţiilor este o problemă complexă; ea poate fi facută fie prin paralelizarea programelor seriale cu ajutorul unor compilatoare specializare (metodă care s-a dovedit a nu fi foarte eficientă), fie prin considerarea paralelismului încă din faza de proiectare a programului. În acest capitol au fost prezentate principalele etape ale construcţiei unui program paralel, considerând un model clasic. Acestea sunt: partiţionarea, comunicaţia, aglomerarea şi maparea. Partiţionarea poate fi funcţională, sau se poate baza pe descompunerea domeniului de date. Abilitatea de a crea un număr variabil de componente de calcul este critică pentru a obţine un program portabil şi scalabil.

Partea II

Proiectare

Capitolul 3

Paradigme ale calculului paralel

Aplicaţiile paralele pot fi clasificate în funcţie de anumite paradigme de programare bine definite. Câteva paradigme de programare sunt folosite în mod repetat pentru dezvoltarea multor programe paralele. O paradigmă poate fi descrisă ca o clasă de programe care au aceeaşi structură de control.

De exemplu, majoritatea aplicaţiilor de calcul distribuit se bazează pe foarte populara paradigmă *client/server*. Este considerată pentru a suporta serviciile distribuite, dar nu reflectă un paralelism al acestor tipuri de aplicaţii. De aceea, această paradigmă nu este considerată o paradigmă de calcul paralel.

Experienţa de până acum arată că există un număr relativ restrâns de paradigme care pot descrie majoritatea programele paralele. Alegerea paradigmei este determinată de resursele de calcul paralel disponibile şi de tipul de paralelism inerent al problemei. Resursele de calcul pot defini nivelul de granularitate care poate fi suportat de sistem, iar tipul de paralelism reflectă structura aplicaţiei sau a datelor. Aşa cum am descris anterior, paralelismul datorat structurii de calcul este numit paralelism funcţional, iar cel datorat structurii datelor paralelism al domeniului de date.

Prezentăm în continuare cinci dintre cele mai cunoscute paradigme ale programării paralele.

3.1 Master/Slave

Această paradigmă care se regăseste şi sub numele de "task-farming", se bazează pe două tipuri de componente: *master* componenta care coordonează calculul şi componentele *slave* care se subordonează componentei *master*. Componenta *master* este responsabilă de descompunerea problemei în subprobleme (şi distribuţia lor către componentele *slave*) cât şi de colectarea rezultatelor parţiale necesare pentru a produce rezultatul final. Componentele *slave* se execută pe baza unui ciclu foarte simplu: preiau un mesaj cu o sarcină de calcul, execută sarcina şi trimit rezultatul componentei *master* – Figura 3.1. În general, comunicaţia apare doar între componenta *master* şi componentele *slave*.

Se poate folosi o încărcare de lucru statică sau dinamică. În cazul încărcării statice, distribuţia componentelor este realizată în totalitate la începutul calculului şi aceasta

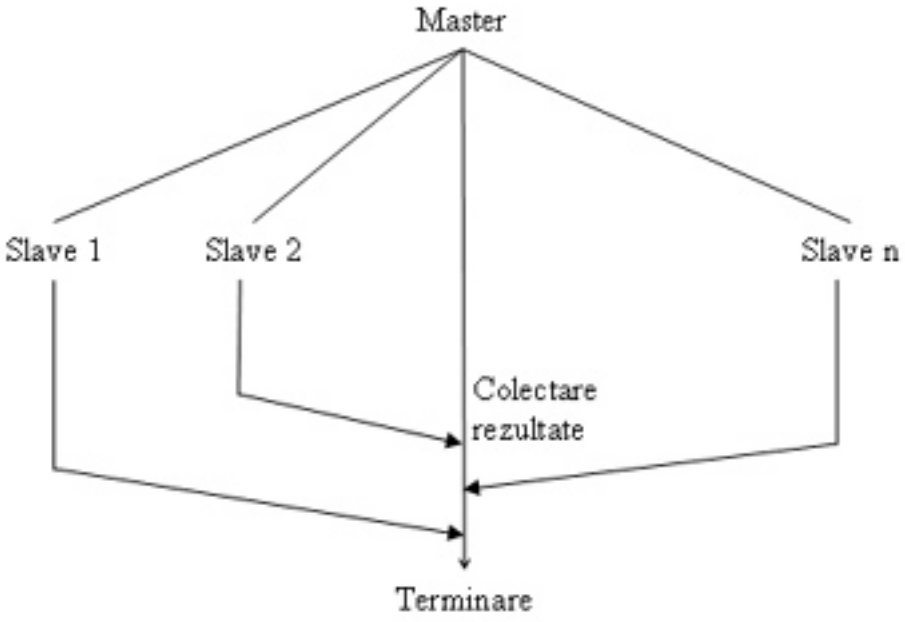

Figura 3.1: Paradigma "Master/Slave".

permite componentei master să participe la calcul după ce a atribuit fiecărei componente *slave* fracţiunea corepunzătoare de calcul. Alocarea componentelor se poate face o singură dată la început sau în mod ciclic.

Încărcarea de calcul dinamică poate fi folosită cu succes în această paradigmă, mai ales atunci când numărul de componente depăşeşte numărul de procesoare disponibile, sau când numărul de componente nu este cunoscut la începutul aplicaţiei. O trăsătură importantă a încărcării dinamice este abilitatea de a se adapta la condiţiile în schimbare ale sistemului, nu doar pentru încărcarea procesoarelor cu componente de calcul dar şi la o posibilă reconfigurare a resurselor sistemului. Datorită acestei proprietăţi, această paradigmă poate să răspundă în mod eficient la blocarea unor procesoare, lucru care simplifică crearea aplicaţiilor robuste, capabile să facă faţă cu bine pierderii unor procesoare.

In extenso, această paradigmă poate include şi aplicaţii care se bazează pe o descompunere trivială a calculului: algoritmul secvenţial este executat simultan pe diferite procesoare pentru diferite date de intrare. În asemenea aplicaţii nu sunt dependenţe între diferitele componente de calcul şi deci nu este nevoie de comunicaţie sau coordonare între ele.

Aplicaţiile ce se încadrează în această paradigmă pot avea acceleraţii de calcul foarte bune şi o scalabilitate rezonabilă. Totuşi, pentru un număr mare de procesoare controlul centralizat poate conduce la blocarea aplicaţiei. Scalabilitatea poate fi îmbunătăţită prin extinderea de la o singură componentă *master* la o mulţime de componente *master*, fiecare dintre ele controlând un grup separat de componente *slave*.

3.2 Work Pool

În această paradigmă componentele de calcul sunt stocate într-un container – aşa zisul bazin ("pool") – aşteptând să fie rezolvate (Figura 3.2). Fiecare procesor liber preia o componentă din container şi returnează un număr (nul sau nenul) de noi componente de calcul, care trebuie rezolvate şi care se stochează la rândul lor în container. Containerul de componente conţine în general şi câteva date globale accesibile tuturor procesoarelor. Aplicaţia se termină când nu mai rămâne nici o componentă în container nerezolvată.

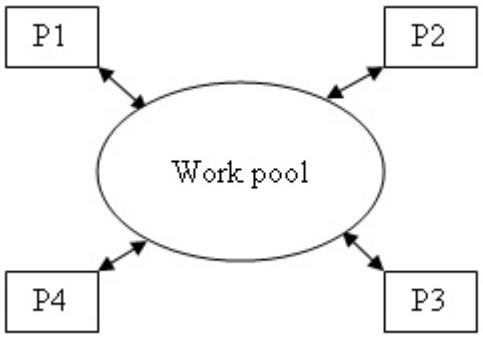

Figura 3.2: Paradigma "Work Pool".

3.3 Paralelism al datelor

Fiecare componentă execută în mod esenţial aceeaşi secvenţă de cod, dar pe părţi diferite de date. Aceasta presupune împărţirea datelor aplicaţiilor şi apoi asignarea lor componentelor disponibile. Această paradigmă se mai regăseste şi sub numele de paralelism geometric, descompunerea domeniului de date, sau SPMD("Single Program Multiple Data").

Multe aplicaţii fizice se bazează pe o structură geometrică regulată cu interacţiuni limitate spaţial. Omogenitatea asigură posibilitatea distribuţiei uniforme a datelor pe procesoare, fiecare corespunzând unei zone spaţiale bine definite. Procesoarele comunică cu procesoarele vecine şi încărcarea de comunicaţie este proporţională cu dimensiunea frontierei elementului, iar încărcarea de calcul este proporţională cu volumul elementului (suprafaţă-volum). Este posibil de asemenea să se ceară anumite sincronizări periodice între componente. În general, şablonul de comunicaţie este foarte structurat şi previzibil. Datele iniţiale pot fi iniţial generate de fiecare proces, sau citite într-o fază de iniţializare.

Aceste aplicaţii pot fi foarte eficiente dacă datele sunt bine distribuite şi sistemul este omogen. Dacă procesele au încărcări de calcul diferite, sau procesoarele au capabilităţi diferite, atunci este necesară o schemă de echilibrare a încărcării în timpul execuţiei.

Paradigma este extrem de sensibilă la pierderea unui proces. În general, pierderea unui singur proces este suficientă pentru determinarea unui blocaj în care nici un proces

nu mai poate avansa dincolo de punctul de sincronizare global.

3.4 Pipeline

Paradigma pipeline se bazează pe un paralelism cu granularitate fină, bazat pe descompunere funcţională. Este una dintre cele mai simple şi mai populare paradigme ale descompunerii funcţionale. Mai este numit şi paralelism al fluxului de date.

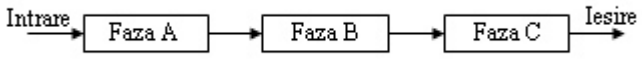

Figura 3.3: Structura pipeline.

Componentele sunt organizate în pipeline - fiecare dintre componente fiind responsabilă pentru o anumită sarcină de lucru. Şablonul de comunicaţie este foarte simplu: fluxul datelor este liniar între fazele adiacente ale calculului, ieşirea unei faze fiind intrare pentru faza următoare (Figura 3.3). Comunicaţia poate fi complet asincronă.

Eficienţa acestei paradigme depinde direct de echilibrarea etapelor de calcul. Robusteţea acestei paradigme în cazul unei reconfigurari a sistemului poate fi obţinută prin furnizarea mai multor căi independente între etape.

Mai multe despre proiectarea aplicaţiilor de acest tip sunt descrise în Secţiunea 4.8.

3.5 Divide&impera

Abordarea divide&impera este bine cunoscută din dezvoltarea algoritmilor secvenţiali. O problemă este împărţită în două sau mai multe subprobleme. Fiecare dintre aceste subprobleme este rezolvată independent şi rezultatele lor sunt combinate pentru a se obţine rezultatul final. În general, subproblemele sunt doar instanţe de dimensiuni mai mici ale problemei iniţiale, ajungându-se la o soluţie recursivă. Subproblemele pot fi rezolvate în paralel, ele nefiind interdependente; nu este necesară nici o comunicaţie între componentele care lucrează la subprobleme diferite. Se pot identifica trei operaţii generice principale: *divide, calculează* şi *compune*. Calculului i se poate asocia o reţea de calcul de tip arbore, componentele care execută operaţiile de calcul fiind asociate nodurilor terminale acestui arbore. Nodurile intermediare realizează doar operaţii de divizare şi combinare.

Paradigma “master/slave” poate fi considerată a fi o formă degenerată a paradigmei divide&impera, în care descompunerea problemei este realizată înaintea asignării componentelor de calcul, iar operaţiile de divizare şi de compunere sunt executate doar de *master*.

Componentele pot fi generate la execuţie şi adăugate unei cozi de execuţie gestionate de un procesor manager sau distribuite mai multor cozi de execuţie aflate în sistem.

3.6 Alte clasificări

În literatura de specialitate existentă sunt mai multe clasificări ale programelor paralele pe baza paradigmei. De exemplu, în [84] se prezintă o clasificare a programelor paralele bazată pe patru clase principale:

1. pipeline şi aplicaţii bazate pe structura inel;
2. divide&impera;
3. "master/slave";
4. aplicaţii bazate pe paralelismul datelor.

O clasificare teoretică este prezentată în [147], prin care se identifică următoarele paradigme:

1. "task-farming", care este echivalentă paradigmei "master/slave";
2. descompunere geometrică, bazată pe paralelizarea structurilor de date;
3. paralelism algoritmic, care rezultă din fluxul datelor.

Următoarea clasificare se bazează pe tehnici de descompunere [174]:

1. Descompunere geometrică: domeniul problemei este împărţit în subdomenii şi fiecare componentă execută algoritmul pe câte un subdomeniu.
2. Descompunere iterativă: se bazează pe paralelizarea ciclurilor, pentru cazurile în care nu există dependenţe între iteraţii diferite. În general, această descompunere se implementează folosind o coadă de componente de calcul şi astfel poate fi considerată făcând parte din paradigma "task-farming".
3. Descompunere recursivă: se bazează pe descompunerea problemei originale în subprobleme pentru care se foloseşte aceeaşi abordare. Corespunde în mod evident paradigmei divide&impera.
4. Descompunere speculativă: se încearcă N soluţii simultan şi se renunţă la $N-1$ dintre ele de îndată ce una a returnat un răspuns plauzibil.
5. Descompunere funcţională: presupune descompunerea aplicaţiei în diferite faze, fiecare fază executând un subalgoritm al problemei date. Poate conduce la paradigma pipeline, dacă fazele se succed secvenţial şi ieşirile unei faze sunt intrări pentru faza următoare.

În [63], este prezentată o clasificare care se bazează pe structura temporală a problemei:

1. probleme sincrone, care corespund unor calcule regulate pe domenii de date regulate;

2. probleme slab sincrone, care consideră calcule iterative pe domenii de date geometric neregulate;
3. probleme asincrone, care sunt caracterizate de paralelism funcţional;
4. probleme stânjenitor de paralele, care corespund execuţiei independente a unor componente independente ale aceluiaşi program.

Problemele sincrone şi slab sincrone se bazează amândouă pe tehnici de descompunere ale domeniului de date. În urma unor analize pe un număr mare de aplicaţii s-a estimat că aceste două clase de probleme domină aplicaţiile paralele într-un procent de 74%. Probleme asincrone, care pot fi de exemplu reprezentate de simulările dirijate de evenimente, reprezintă un procent de 10%. Aplicaţiile stânjenitor de paralele care corespund modelului "master/slave" se regăsesc într-un procent de 14%.

Sumar

Chiar dacă nu s-a ajuns la un consens, încercarea de clasificare a algoritmilor paraleli în funcţie de paradigma folosită este o problemă importantă. Ea poate conduce la simplificarea construcţiei programelor paralele prin încadrarea lor de la început într-o clasă/paradigmă sau alta. După cum am văzut în acest capitol există mai multe clasificări. Paradigmele: *master-slave*, *work-pool*, *pipeline*, *divide&impera* şi *paralelismul datelor* par a fi cele consacrate.

Capitolul 4

Tehnici folosite în construcţia algoritmilor paraleli

Există câteva tehnici foarte cunoscute, care sunt folosite pentru dezvoltarea algoritmilor în general. Acestea includ metoda divide&impera, metoda greedy, programare dinamică, backtracking. Aceste metode reprezintă strategii de rezolvare care sunt frecvent folosite în dezvoltarea eficientă a algoritmilor. Scopul identificării acestor tehnici are un dublu rol. Pe de-o parte de a furniza un cadru ştiinţific şi matematic, care ne poate ajuta în a rezolva probleme diferite prin mijloace similare. Pe de altă parte, aceste tehnici reprezintă pentru programator "unelte" necesare şi eficiente pentru rezolvarea problemelor.

Construcţia unui program paralel presupune partiţionarea calculului şi precizarea comunicaţiilor necesare, care pot fi extrem de complexe. În general, tehnicile de programare pentru programarea secvenţială includ informaţii despre şabloanele necesare organizării datelor. În cazul calculului paralel tehnicile încapsulează, în general, informaţii şi despre şabloanele de comunicaţie necesare. Deoarece în cazul unei arhitecturi paralele o problemă importantă o constituie asigurarea unei comunicaţii eficiente şi efective între procesoare, tehnicile au un rol şi în specificarea şabloanelor de comunicaţie cele mai folosite.

În cele ce urmează vom prezenta cele mai cunoscute tehnici de proiectare pentru algoritmi paraleli şi acestea vor fi evidenţiate prin exemple. Tehnicile pe care le prezentăm au în vedere în special partiţionarea şi de aceea nu vom evidenţia diferenţele care apar pentru cazul memoriei distribuite faţă de cazul memoriei partajate, sau adaptările pentru diferite reţele de interconectare pentru fiecare algoritm in parte. Acestea probleme vor fi analizate la modul general, pe cateva exemple, în ultimele secţiuni ale capitolului.

Limbajul de descriere a algoritmilor este un pseudocod care conţine instrucţiunile de bază ale unui limbaj de nivel înalt precum şi instrucţiuni pentru descrierea paralelismului.

- Atribuirea se va simboliza prin $\leftarrow$.
  Simbolul $\leftrightarrow$ se va folosi pentru interschimbarea valorilor a două variabile.

- Instrucţiunea alternativă va avea forma:

  if *expresie logica* **then**
  instructiuni
  else
  instructiuni
  end if

  unde secţiunea **else** poate lipsi.

- Pentru instrucţinea de ciclare cu număr determinat de paşi vom folosi instrucţiunea:

  for $contor = li, lf, pas$ **do**
  instructiuni
  end for

  unde contorul ia valori de la valoarea iniţială li, la valoarea finală lf, la fiecare pas adăugându-se valoarea pas. Pasul poate lipsi fiind implicit egal cu 1.

- Instrucţiunea iterativă

  while *expresie logica* ***do***
  instructiuni
  end while

  reprezintă ciclul cu număr necunoscut de paşi şi test iniţial.

Pentru descrierea paralelismului vom folosi instrucţiunea de compunere în paralel a două sau mai multe instrucţiuni cu sintaxa:

in parallel
instructiuni
end in parallel

unde instrucţiunile sunt separate prin **,**.

Compunerea paralelă a mai multor instrucţiuni se termină atunci când toate aceste instrucţiuni sunt terminate.

Compunerea secvenţială a instrucţiunilor se specifică prin simbolul **;**.

instructiune-1
;
instructiuni-2

semnifică faptul că execuţia celei de a doua instrucţiuni începe doar după terminarea primeia.

Instrucţiunea de compunere în paralel poate fi combinată cu instrucţiunea **for**, caz în care se specifică un număr de instrucţiuni care se execută în paralel. De exemplu, instrucţiunea

for $i = 0, p-1$ **in parallel do**
  $c_i = a_i + b_i$
end for

descrie adunarea în paralel a doi vectori a şi b cu câte p elemente, rezultatul fiind depus în vectorul c.

Presupunem existenţa unei maşini paralele virtuale cu memorie partajată şi că accesul la variabile se va face direct fără precizarea diferenţelor dintre variabilele globale şi cele locale. Prin urmare, putem considera că lucrăm în contextul unui model PRAM relaxat. O analiză ulterioară a algoritmilor se poate face pentru diferitele adaptări posibile.

Pentru cazul în care vom dori descrierea unui algoritm bazat pe transmitere de mesaje vom considera că fiecare proces este identificat în mod unic şi vom folosi următoarele două instrucţiuni:

- **send**$(date, destinatie)$ – pentru trimiterea unui mesaj format din *date* către procesul specificat de *destinatie*;
- **recv**$(date, sursa)$ – pentru recepţionarea unui mesaj şi stocarea lui în *date* de la procesul specificat de *sursa*.

Vom considera că instrucţiunea *recv* se execută prin blocare, adică procesul care o conţine rămâne blocat până când data este primită de la procesul transmiţător. Algoritmul va fi parametrizat prin identificatorul de proces şi va descrie acţiunea fiecărui proces, încadrându-se în tipul SPMD.

4.1 Tehnica paralelizării directe

Anumite probleme pot fi considerate *direct sau stânjenitor paralelizabile* ("embarrassingly parallel computations"), dacă calculul asociat lor poate fi împărţit direct într-un număr de componente independente, care pot fi executate de câte un procesor. Exemple de asemenea calcule sunt: calculele de tip Monte Carlo folosite pentru selecţii aleatoare, mulţimile Mandelbrot folosite pentru fractali, procesarea imaginilor la nivel scăzut.

Exemplul 4.1 (Aproximarea numărului π) Un calcul de tip Monte Carlo se realizează pentru aproximarea numărului π prin următoarea metodă:
Se consideră un cerc de rază egală cu unitatea înscris într-un pătrat. Are loc următoarea egalitate:

$$\frac{\text{Aria cercului}}{\text{Aria pătratului}} = \frac{\pi(1)^2}{2 \cdot 2} = \frac{\pi}{4}$$

Se aleg în mod aleator puncte în interiorul pătratului (un număr suficient de mare) şi se aproximează numărul $\frac{\pi}{4}$ cu raportul dintre numărul de puncte care sunt în interiorul

```
ALGORITM NUMAR-PI< n, p, pi>
  nrint ← 0;
  for  i = 0, p − 1 in parallel do
    for  j = 0, n − 1  do
      x ← RANDOM(−1, 1);
      y ← RANDOM(−1, 1);
      if (x^2 + y^2 ≤ 1) then
        nint ← nint + 1;
      end if
    end for
  end for
  pi ← 4 ∗ nrint/(n ∗ p)
```

cercului şi numărul total de puncte. Algoritmul paralel va permite fiecărui procesor să aleagă aleator n puncte şi să verifice dacă sunt în interiorul cercului sau nu.

S-a folosit o funcţie RANDOM(a, b) care returnează un număr aleator în intervalul $[a, b]$. Alegerea aleatoare a numerelor şi verificarea lor se paralelizează direct, fără probleme. Complexitatea acestui algoritm este $O(n)$, unde n este numărul de puncte generate aleator de fiecare componentă. Varianta secvenţială a acestui calcul are complexitatea $O(np)$ şi prin urmare acest algoritm este un algoritm foarte eficient: $E_p(n) \approx 1$.

În această variantă, fiecare componentă de calcul citeşte şi actualizează variabila globală *nrint* – numărul total de puncte din interiorul cercului. Prin urmare, la un moment dat o componentă poate dori să scrie(actualizeze) variabila *nrint* în timp ce alta cere citirea ei. Trebuie, deci, impuse reguli care să asigure consistenţa. Pentru a elimina aceste probleme se poate considera o variantă în care fiecare componentă calculează un număr local de puncte şi apoi acestea se însumează printr-un algoritm paralel de calculare a sumei. În acest caz nu mai avem doar paralelizare directă, datorită interacţiunii dintre componentele de calcul, necesare calculării sumei.

Acest tip de calcul poate fi folosit şi pentru aproximarea integralei unei funcţii date $f(x)$ pe un interval (a, b), atunci când integrala nu poate fi calculată numeric. Se foloseşte formula:

$$Aria = \int_a^b f(x)dx = \lim_{n\to\infty} S_n = \lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} f(x_i)(b-a)$$

Punctele x_i se aleg aleator în intervalul (a, b) şi se calculează suma S_n.

Exemplul 4.2 (Rezolvarea unui sistem liniar prin metoda iterativă Jacobi)

Metoda Jacobi de rezolvare a unui sistem liniar $A \times X = B$ cu n ecuaţii şi n necunoscute presupune aplicarea iterativă a unui pas de aproximare Jacobi, pornind de la o

```
ALGORITM JACOBI< n, t, A[0..n-1][0..n-1], B[0..n-1], X[0..n-1]>
  for k = 0, t - 1 do
    for i = 0, n - 1 in parallel do
      suma[i] ← 0;
      for j ← 0, n - 1 do
        if (i ≠ j) then
          suma[i] ← suma[i] + A[i][j] * X[j];
        end if
      end for
      suma[i] ← (B[i] - suma[i])/A[i][i];
    end for
    for i = 0, n - 1 in parallel do
      X[i] ← suma[i];
    end for
  end for
```

aproximare iniţială $X^0 = (x_0^0, x_1^0, \ldots, x_{n-1}^0)$. Formula de calcul a unei noi aproximări este

$$x_i^k = \left(b_i - \left(\sum j : 0 \leq j < n \wedge j \neq i : a_{i,j} * x_j^{k-1}\right)\right) / a_{i,i}$$

Se obţin astfel aproximări succesive ale soluţiei: $X^0, X^1, \ldots, X^k, \ldots$. Procesul de calcul este convergent dacă după un număr de k iteraţii, vectorul $(B - AX^k)$ devine acceptabil de mic. Terminarea calculelor se face prin compararea a două valori succesive ale vectorului X. Este posibil ca procesul de calcul să nu se termine sau să avanseze lent, caz în care se poate stabili un număr maxim de iteraţii; algoritmul prezentat foloseşte această variantă.

Paralelizarea se poate face simplu datorită faptului că fiecare variabilă X_i poate fi calculată independent, în funcţie de aproximarea precendentă. Dacă dorim aplicarea pasului de aproximare Jacobi de t ori se obţine algoritmul paralel ALGORITM JACOBI descris. Este de remarcat faptul că toate cele n componente care lucrează în paralel trebuie să citească cele n valori ale vectorului X; deci este necesară, pentru o execuţie eficientă, posibilitatea citirii concurente din memoria comună. Complexitatea acestui algoritm este $O(tn)$ şi din nou eficienţa este maximă. În general, algoritmii care permit o astfel de paralelizare directă sunt foarte eficienţi, nefiind nevoie de nici o interacţiune între componentele concurente de calcul.

Nu întotdeauna algoritmii sunt atât de uşor de paralelizat şi în general anumite tehnici pot fi utilizate pentru partiţionare. În continuare vom prezenta câteva din cele mai folosite asemenea tehnici.

4.2 Tehnica arbore binar

Problemele pentru care se poate folosi această metodă aplică în general o singură operaţie unui număr mare de date, iar tehnica constă în împărţirea iterativă a domeniului de date asupra căreia se aplică operaţia, în părţi egale.

Dacă un asemenea calcul este realizat secvenţial, timpul de execuţie este aproximativ proporţional cu numărul de noduri ale arborelui binar, care reprezintă reţeaua de calcul. Dacă este făcută în paralel, calcularea diferitelor ramuri ale arborelui este independentă. Prin urmare, timpul de execuţie paralelă este aproximativ proporţional cu distanţa de la rădăcină la frunze, în arborele reţelei de calcul.

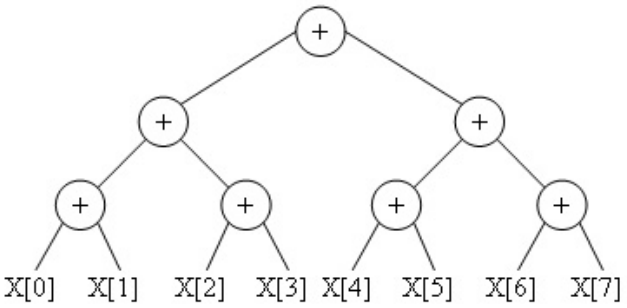

Figura 4.1: Reţea de calcul de tip arbore binar cu adâncimea 3, pentru adunarea a 8 numere.

Exemplul clasic pentru această tehnica este suma a n numere, iar reţeaua de calcul echivalentă pentru $n = 8$ este cea din Figura 4.1.

Operaţia de adunare poate fi înlocuită cu orice operaţie asociativă (min, max, produs, etc.), iar numerele pot fi înlocuite cu orice tipuri de date.

Exemplul 4.3 (Suma a n numere) Considerăm un şir de numere $A[0], A[1], \ldots, A[n-1]$, cu $n = 2^k$. Algoritmul SUMA-1 calculează suma celor n numere şi o depune în locaţia $A[0]$.

ALGORITM SUMA-1$< n = 2^k, A[0..n-1] >$

for $t = 1, \log_2 n$ **do**
  for $i = 0, n-1$ **in parallel do**
    if $(i \,\%2^t = 0 \wedge i + 2^{t-1} < n)$ **then**
      $A[i] \leftarrow A[i] + A[i + 2^{t-1}]$
    end if
  end for
end for

ALGORITHM SUMA-2$< n = 2^k, A[0..n-1] >$
for $t = 1, \log_2 n$ **do**
for $i = 0, n-1$ **in parallel do**
if $(i < n/2^t)$ **then**
$A[i] \leftarrow A[2i] + A[2i+1]$
end if
end for
end for

Un algoritm echivalent este şi algoritmul SUMA-2.

Ce două variante diferă prin locaţia în care se depune rezultatul unei sume. În prima variantă, suma se depune în prima din cele două variabile însumate; astfel după prima iteraţie sumele parţiale sunt depuse în locaţiile pare, după a doua iteraţie în locaţiile divizibile cu 4, etc. În cea de-a doua variantă, sumele se depun în primele locaţii ale vectorului A: prima jumătate după prima iteraţie, primul sfert după cea de-a doua, etc.

Complexitatea-timp a acestui algoritm este egală cu $O(\log_2 n)$, iar $\log_2 n$ reprezintă numărul de paşi paraleli efectuaţi.

Dacă considerăm un program PRAM şi includem în calculul complexităţii şi numărul de citiri şi scrieri atunci avem $n/2 + n/2^2 + \ldots + 1 = n - 1$ citiri şi scrieri în memoria globală. Nu există citiri şi scrieri concurente şi prin urmare citirile, respectiv scrierile executate la un superpas pot fi executate simultan. Prin urmare, complexitatea-timp poate fi evaluată ca fiind egală cu $\log_2 n * (T(+) + T(read) + T(write))$.

Exemplul 4.4 (Evaluarea relaţiilor de recurenţă) Tehnica se poate aplica şi pentru probleme pentru care reţeaua de calcul arbore binar nu este chiar atât de evidentă. Considerăm relaţia de recurenţă liniară cu doi termeni, de forma:

$$x_i = a_i x_{i-1} + b_i x_{i-2}, (i = 2, \ldots, n),$$

care trebuie rezolvată pentru a determina $x_2, \ldots x_n$, atunci când se cunosc x_0 şi x_1, a_i şi b_i. Folosim notaţiile:

$$Y_1 = \begin{bmatrix} x_1 \\ x_0 \end{bmatrix}, Y_i = \begin{bmatrix} x_i \\ x_{i-1} \end{bmatrix}, (i = 2, \ldots, n).$$

şi prin reformularea ecuaţiei în scriere matrice-vector obţinem:

$$Y_i = M_i Y_{i-1},\ M_i = \begin{bmatrix} a_i & b_i \\ 0 & 0 \end{bmatrix}, (i = 2, \ldots, n);$$

$$\begin{bmatrix} x_n \\ x_{n-1} \end{bmatrix} = Y_n = M_n M_{n-1} \ldots M_2 \begin{bmatrix} x_1 \\ x_0 \end{bmatrix}$$

Prin urmare este suficient să se calculeze produsul a $n-1$ matrice 2×2 care poate fi efectuat printr-un calcul de tip arbore binar, cu o complexitate $O(\log_2 n)$.

"Compute-Aggregate-Broadcast" O tehnică de combinaţie este aşa numita *calculează-combină-difuzează*("compute-aggregate-broadcast") [126], care este aplicabilă programelor paralele pe arhitecturi cu memorie distribuită.

Un algoritm paralel construit pe baza acestei tehnici constă din trei faze de bază: o fază în care se realizează calcule cu datele locale, o fază în care se combină rezultatele parţiale obţinute la prima fază şi o fază de difuzare, care transmite rezultatul final fiecărui element de procesare. Faza de calcul variază de la o problemă la alta. Poate fi compusă doar din calcule ale unor operaţii primitive, sau poate fi de complexitatea unui aplicaţii complete. Faza de agregare sau combinare este în general un calcul de arbore binar şi rezultatul final este obţinut într-un singur nod – nodul rădăcină. În final, ultima fază transmite rezultatul fazei de combinare, tuturor celorlalte elemente de procesare printr-o comunicaţie de tip difuzare ("broadcast"). Un exemplu pentru aceasta tehnică este algoritmul prezentat în Exemplul 4.25 de adunare a n numere folosind p procese $p < n$, unde şirul iniţial se împarte în n/p subşiruri pentru care se calculează sume locale în paralel şi apoi acestea sunt însumate printr-un calcul de tip arbore binar. În final, suma totală poate fi transmisă tuturor elementelor de procesare.

4.3 Tehnica dublării recursive

Aceasta tehnică impune ca la fiecare pas, fiecare componentă de calcul să îşi dubleze numărul de elemente pentru care a realizat calculul cerut. Astfel, "progresul" este obţinut prin dublarea numărului de elemente la fiecare pas.

Tehnica arborelui binar poate fi încadrată în această categorie, căci după cum s-a văzut în exemplul calculării sumei, la fiecare nivel numărul elementelor care au fost însumate se dublează. Există însă şi alte situaţii în care această tehnică poate fi folosită, când structura arborescentă nu este aşa de evidentă. Vom prezenta în continuare câteva astfel de exemple, care ilustrează acest lucru.

Exemplul 4.5 (Rang-în-listă) Se dă o listă înlănţuită cu n elemente. Se cere să se calculeze pentru fiecare element rangul lui în listă. Rangul unui element în lista se defineşte ca fiind numărul de elemente de la el la sfârşitul listei. Astfel, primul element din listă are rangul n, cel de-al doilea are rangul $n-1$, iar ultimul are rangul 1.
Pentru a rezolva această problemă, considerăm că fiecare element este atribuit unui procesor. La început fiecare procesor p_i cunoaşte doar elementul imediat următor elementului său, notat cu $urmator[i]$ (cu excepţia ultimului procesor pentru care $urmator(n-1) = -1$). La fiecare pas procesorul p_i îşi actualizează $urmator[i]$ cu valoarea $urmator[urmator[i]]$. Dacă la pasul k fiecare procesor cunoaşte elementul aflat la distanţa x, la următorul pas va cunoaşte elementul aflat la distanţa $2x$. Astfel, se garantează că după cel mult $\lceil \log_2 n \rceil$ paşi fiecare procesor va atinge sfârşitul listei şi astfel va cunoaşte rangul elementului său.

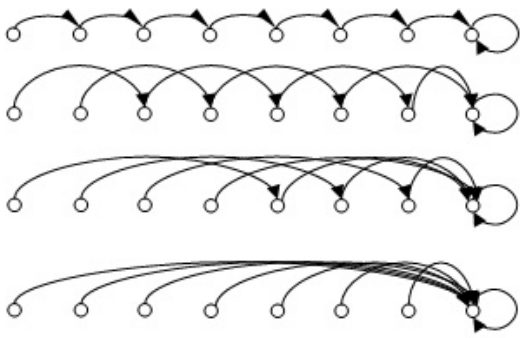

Figura 4.2: Tehnica "pointer-jumping"

ALGORITHM RANG-IN-LISTA$< n, urmator[0..n-1], rang[0..n-1] >$

for $i = 0, n-1$ **in parallel do**
  $rang[i] \leftarrow 1$;
  while $(urmator[i] \neq -1)$ **do**
    $rang[i] \leftarrow rang[i] + rang[urmator[i]]$;
    $urmator[i] \leftarrow urmator[urmator[i]]$;
  end while
end for

Acest exemplu foloseşte o tehnică care se regăseşte în literatura sub numele de **"pointer jumping"**(saltul de pointer) – Figura 4.2. Poate fi aplicată pentru diferite probleme, mai ales din teoria grafurilor.

Exemplul 4.6 (Lungimea unui drum în arbore) Considerăm un arbore cu n noduri şi dorim să găsim distanţa de la rădăcină la fiecare nod, dacă se cunoaşte pentru fiecare nod i nodul său părinte: $parinte[i]$. Rădăcina are părintele egal cu -1. Algoritmul foloseşte aceeaşi tehnică folosită şi de algoritmul pentru aflarea rangului unui element în listă.

ALGORITHM LUNGIME-DRUM$< n, parinte[0..n-1], distanta[0..n-1] >$
  for $i = 0, n-1$ **in parallel do**
    $distanta[i] \leftarrow 1$;
    while $(parinte[i] \neq -1)$ **do**
      $distanta[i] \leftarrow distanta[i] + distanta[parinte[i]]$;
      $parinte[i] \leftarrow parinte[parinte[i]]$;
    end while
  end for

Exemplul 4.7 (Arbore de acoperire de înălţime minimă) În cazul grafurilor, poate fi folosită aşa numita tehnică de dublare a arborelui de acoperire. Fie graful orientat $G = (X, E)$ cu n noduri. Se începe prin a considera toţi arborii a căror rădăcină este un nod al grafului iniţial şi au înălţimea 1. Adică pentru fiecare nod x al grafului, arborele cu rădăcina x conţine toate nodurile y pentru care (x, y) este arc în graful G. Apoi, pentru fiecare arbore de acest fel se realizează următoarea operaţie: nodurile frunză se înlocuiesc cu arborii corespunzători acelor noduri (care au rădăcina un nod frunză) şi după aceasta se elimină nodurile duplicate, dacă acestea există. Dacă această operaţie se repetă, după cel mult $\lceil \log_2 n \rceil$ paşi se ajunge la soluţie: arbore de acoperire minim.

În Figura 4.3 se ilustrează această tehnică pentru graful cu 6 noduri prezentat în Figura 4.3 (a). Se începe cu arborii care au rădăcini nodurile din G şi au înălţimea 1 – Figura 4.3 (b). După primul pas se obţin arbori cu înălţime cel mult 2; se elimină duplicatele (notate cu X pentru fiecare arbore rezultat) – Figura 4.3 (c). Se verifică apoi dacă nu s-a ajuns la un arbore de acoperire (toate nodurile sunt conţinute). Pentru exemplul considerat, primul arbore reprezintă un arbore de acoperire cu rădăcina primul nod al grafului.

Operaţia efectuată la un pas, se poate executa în paralel pentru fiecare arbore, iar adâncimea arborilor obţinuţi de dublează la fiecare pas. Pentru reprezentarea arborilor se pot folosi liste de adiacenţă. La fiecare pas, lista de adiacenţă a fiecărui arbore se modifică prin combinarea ei cu listele de adicenţă ale arborilor ale căror rădăcini sunt frunze în lista respectivă.

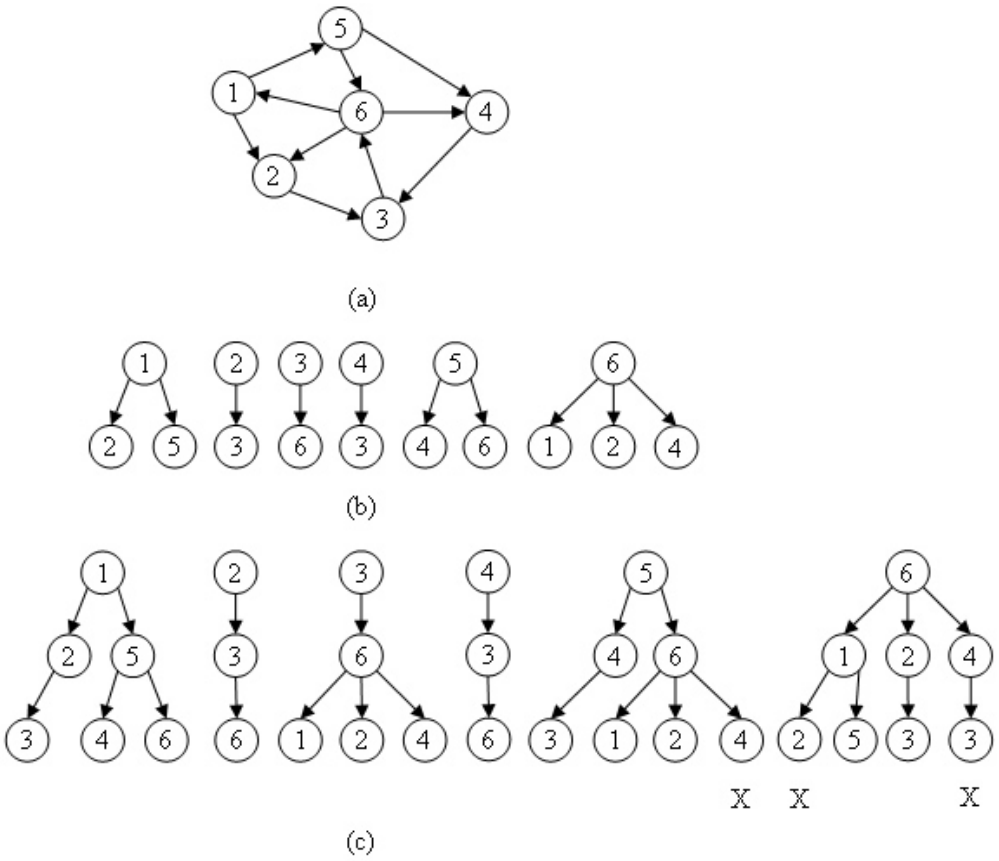

Figura 4.3: Construirea arborelui de acoperire minim. (a) Graful orientat pentru care se construieşte arborele de acoperire minim; (b) primul pas – arborii de înălţime 1 iniţiali; (c) pasul doi – arbori de înălţime cel mult 2, dintre care primul este un arbore de acoperire (cu X sunt notate nodurile care se repetă şi trebuie eliminate).

Această tehnică poate fi folosită si pentru alte probleme din teoria grafurilor, cum este determinarea celui mai scurt drum între oricare două noduri.

4.4 Contracţia arborescentă

O tehnică aplicabilă pentru efectuarea în paralel a anumitor operaţii pe arbori, pentru care fiecare nod interior are cel puţin doi fii, este contracţia arborescentă [120]. În general, se au în vedere arbori cu rădăcină, pentru care se calculează o valoare asociată cu nodurile arborelui. Aceste valori trebuie să aibă proprietatea că valoarea în nodul v depinde de întregul subarbore al cărei rădăcină este v. În general, prin contracţia paralelă arborescentă se calculează aceste cantităţi pentru rădăcina arborelui. Exemple de asemenea valori sunt:

- numărul de fii ai lui v;

- distanţa de la v la cea mai apropriată, sau la cea mai depărtată frunză.

Metoda generală a contracţiei arborescente paralele constă în aplicarea repetată a următoarei operaţii de simplificare: *Se selectează toate nodurile arborilor ale căror fii direcţi sunt frunze. Se execută în paralel calculul pentru nodurile selectate şi apoi se şterg nodurile frunză.*

Cerinţa ca fiecare nod interior să aibă cel puţin doi fii asigură faptul că cel puţin jumătate din numărul total de noduri sunt noduri frunză, la orice pas. Aceasta înseamnă că la fiecare pas cel puţin jumătate din nodurile arborelui se şterg şi prin urmare, cel mult $O(\log_2 n)$ operaţii de simplificare sunt necesare pentru a reduce întregul arbore la rădăcină.

Exemplul 4.8 (Distanţa la cea mai apropiată frunză) Fie graful T cu n noduri, în care fiecare nod are cel puţin doi şi nu mai mulţi de c fii. Algoritmul DISTANTA-FRUNZA calculează distanţa de la rădăcină la cel mai apropiat nod-frunză.

ALGORITHM DISTANTA-FRUNZA< graful $T, distanta[v : v$ nod in $T]>$

for <*toate nodurile v*> **in parallel** ***do***
  $distanta[v] \leftarrow 0$;
end for
while (< *rădăcina are fii* >) **do**
  <*se marchează toate nodurile frunză*>;
  <*se marchează toate nodurile ale căror fii sunt noduri-frunză*>;
  for <*toate nodurile v ale căror fii* $z_1, z_2, \ldots z_c$ *sunt frunze*> **in parallel do**
    $distanta[i] \leftarrow 1 + min(distanta[z_1], \ldots, distanta[z_c])$;
  end for
  <*se şterg toate nodurile frunză*>;
end while

Pentru arbori cu proprietatea că fiecare nod intern are exact doi fii, s-a dezvoltat o varianta mai concretă a contracţiei arborescentă bazată pe o operaţie numită "**rake (shunt)**" [101, 2].

Presupunem că avem un arbore binar T cu rădăcina r, părintele fiecărui nod $v, v \neq r$, fiind notat cu $p(v)$. Toate nodurile interne ale arborelui T au exact doi fii. Operaţia de bază *rake* se defineşte astfel:

Dacă u este frunză cu $p(u) \neq r$, atunci se elimină din T nodurile u şi $p(u)$ şi se conectează fratele lui u la $p(p(u))$.

Operaţia se va aplica repetat până când se ajunge la un arbore cu trei noduri. Operaţiile *rake* care nu folosesc noduri comune pot fi executate în paralel.

În algoritmul CONTRACTIE-ARBORESCENTA, părinţii nodurilor afectaţi de operaţiile *rake* nu sunt adiacenţi şi la fiecare iteraţie numărul frunzelor scade la jumătate. Deci, după $\lceil \log_2 n + 1 \rceil$ operaţii, toate frunzele cu excepţia primei şi ultimei, dispar.

ALGORITHM CONTRACTIE-ARBORESCENTA$< T, p, b >$

$<$ *se etichetează toate frunzele în ordine de la stânga la dreapta,*
excluzând frunza cea mai din stânga şi cea mai din dreapta;
se memorează aceste etichete în tabloul A *de dimensiune* $n >$
for $i = 1, \lceil \log_2 n + 1 \rceil$ **do**
$<$*aplică operaţia rake în paralel elementelor* A_{odd} *(cu indice impar)*
care sunt descendenţi stângi$>$;
$<$*aplică operaţia rake în paralel elementelor rămase în* $A_{odd} >$;
$A \leftarrow A_{even}$ $\{A_{even}$ conţine elementele cu indice par$\}$
end for

Exemplul 4.9 (Evaluarea expresiilor aritmetice)

Algoritmii secvenţiali uzuali pentru evaluarea expresiilor aritmetice asociază (implicit sau explicit) unei expresii aritmetice un arbore binar de evaluare.

Considerăm că avem asociat pentru expresia E un arbore binar T_E cu rădăcina r, astfel încât fiecare nod intern v reprezintă o operaţie binară $\odot \in \{+, -, *, /\}$, iar fiecărei frunze u îi este asociată o constantă (sau un operator unar) $val(u)$. Dacă pentru fiecare nod v notăm cu T_v subarborele cu rădăcina v, atunci algoritmul secvenţial de evaluare a expresiei E se bazează pe următoarele relaţii:

1. $val(E) = val(T_E)$,
2. $val(T_E) = val(T_u) \odot val(T_v)$, unde u şi v sunt descendenţii rădăcinii,
3. $val(T_u) = val(u), \forall u$ frunză.

Pentru algoritmul paralel, considerăm mai întâi cazul simplificat în care $\odot \in \{+, *\}$.

Ideea algoritmului paralel de evaluare este următoarea: se asociază fiecărui nod v, diferit de rădăcină o etichetă (a_v, b_v) cu semnificaţia $val(T_v) = a_v * val(v) + b_v$, adică operandul pe care îl va trimite acest nod părintelui său $p(v)$ se va obţine prin calcul în funcţie de etichetă. Valoarea $val(v)$ se cunoaşte doar pentru frunze. Operaţia *rake* va modifica etichetele în mod corespunzător, astfel încât în final să se ajungă la valoarea expresiei.

Din faptul că pentru frunze $val(T_v) = val(v)$, rezultă că putem considera iniţial etichetele egale cu $(1, 0)$.

Pentru a determina modificările necesare ale etichetelor pentru operaţia *rake*, considerăm un nod intern w cu descendenţii u şi v şi cu operaţia asociată $\odot$:

$$val(T_w) = a_w * val(w) + b_w = a_w \left[(a_u * val(u) + b_u) \odot (a_v * val(v) + b_v) \right] + b_w$$

În cazul aplicării operaţiei *rake*, unul din cele două noduri u şi v, este frunză; considerăm că u, descendentul stâng este frunză, deci $val(u) = c_u$ este cunoscut. Nodul w se va înlocui cu nodul v, iar nodul u va fi eliminat.

Obţinem:

$$a_w\left[(a_u * c_u) + b_u\right) \odot (a_v * val(v) + b_v)] + b_w = a_v^{'} * val(v) + b_v^{'}$$

Prin urmare, putem deduce valoarea noii etichete $(a_v^{'}, b_v^{'})$, în funcţie de operatorul $\odot$. Pentru $\odot = +$ avem:

$$a_w\left[(a_u * c_u) + b_u\right) + (a_v * val(v) + b_v)] + b_w = a_v^{'} * val(v) + b_v^{'}$$

şi deci

$$\begin{cases} a_v^{'} = a_w * a_v \\ b_v^{'} = a_w\left(a_u * c_u + b_u + b_v\right) + b_w \end{cases}$$

Pentru $\odot = *$ avem:

$$a_w\left[(a_u * c_u) + b_u\right) * (a_v * val(v) + b_v)] + b_w = a_v^{'} * val(v) + b_v^{'}$$

şi deci

$$\begin{cases} a_v^{'} = a_v * a_w\left(a_u * c_u + b_u\right) \\ b_v^{'} = b_v * a_w\left(a_u * c_u + b_u\right) + b_w \end{cases}$$

În final, se va ajunge la un arbore cu trei noduri, pentru care evaluarea se face direct:

$$val(T_E) = val(r) = (a_u c_u + b_u) \odot (a_v c_v + b_v)$$

În Figura 4.4 se evidenţiază paşii pentru calcularea expresiei $E = ((2 * 3) + (4 + (5 * 6))) * 2 = 80$.

Dacă considerăm expresii care conţin şi operatorii $-, /$ atunci etichetele vor fi formate din matrici (M) de ordin 2 iniţializate cu matricea unitate. Valoarea care se va trimite părintelui unui nod v va fi:

$$\frac{m_{11} val(v) + m_{12}}{m_{21} val(v) + m_{22}},$$

unde m_{ij} sunt elementele matricei etichetă.

Pentru operaţia *rake* cu frunza-stângă u cu valoarea c, nodul intern w şi nodul frate v al lui u, transformarea etichetei pentru v se va face pe baza formulei $M^{'}(v) = A \times M(v)$, unde:

$$A = \begin{cases} \begin{pmatrix} 1 & c \\ 0 & 1 \end{pmatrix}, & \text{dacă } \odot = + \\ \begin{pmatrix} c & 0 \\ 0 & 1 \end{pmatrix}, & \text{dacă } \odot = * \\ \begin{pmatrix} -1 & c \\ 0 & 1 \end{pmatrix}, & \text{dacă } \odot = - \\ \begin{pmatrix} 0 & c \\ 1 & 0 \end{pmatrix}, & \text{dacă } \odot = / \end{cases}$$

Dacă se foloseşte frunza-dreapta la operaţia *rake*, atunci se procedează în mod similar, modificându-se matricea A pentru ultimele două operaţii care sunt necomutative.

Complexitatea-timp pentru evaluarea unei expresii aritmetice, în paralel, folosind metoda contracţiei, este de $O(\log_2 n)$, unde n este numărul de noduri ale arborelui binar corespunzător expresiei.

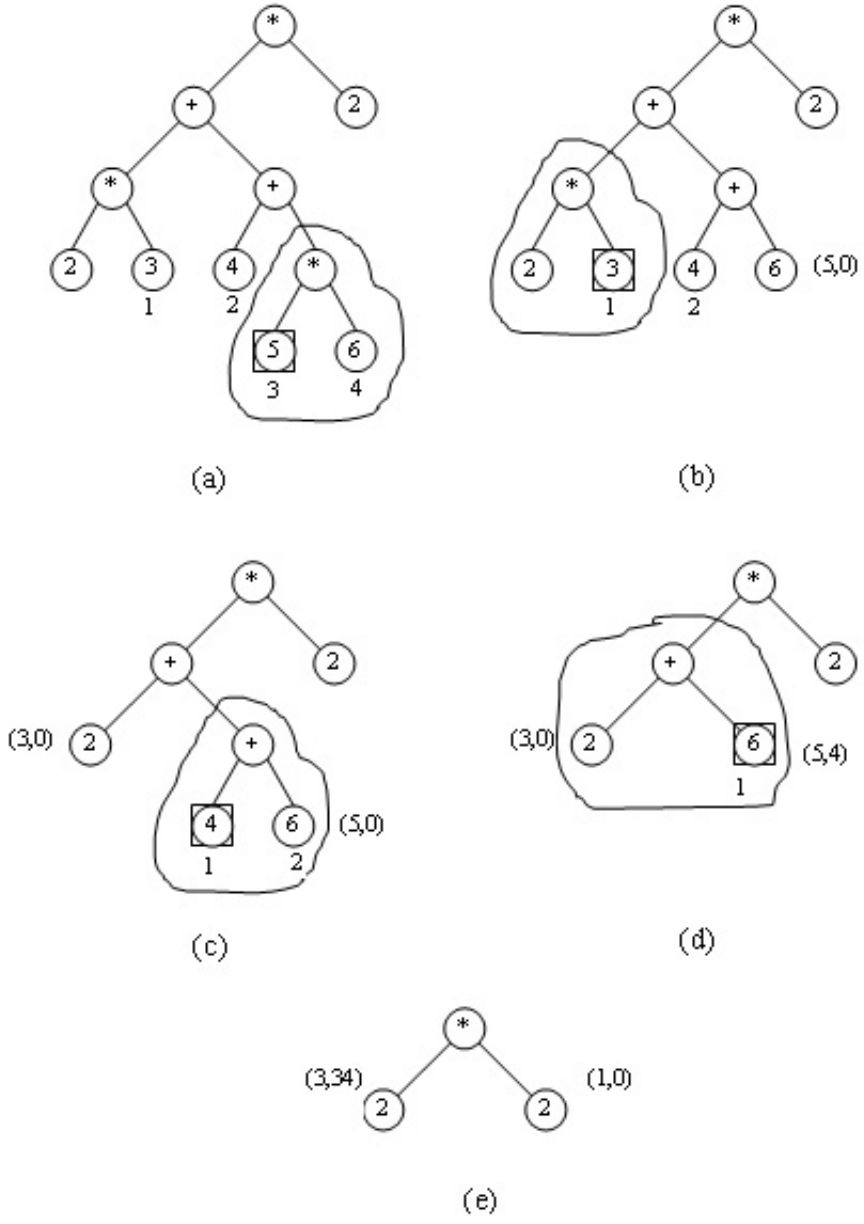

Figura 4.4: Operaţiile *rake* pentru calcularea expresiei $E = ((2*3)+(4+5*6))*2 = 80$.

4.5 Tehnica reducerii ciclice par-impar

Tehnica reducerii ciclice par-impar a fost introdusă de Hockney şi poate fi utilizată pentru rezolvarea unei game largi de relaţii de recurenţă. Se aplică aceleiaşi clase de probleme ca şi dublarea recursivă şi facilitează creşterea vitezei de calcul. Şi aici un rol esenţial îl are ideea reformulării calculului (modificarea reţelei de calcul) cu scopul evidenţierii operaţiilor care pot fi executate independent - deci în paralel.

Pentru a exemplifica această tehnică vom considera o relaţie de recurenţă liniară simplă:

$$x_i = a_i + b_i x_{i-1}$$

unde $a_i, b_i, i = 1, \ldots, n$ sunt numere reale cunoscute şi $x_0 = \alpha$ este de asemenea cunoscută. Se cere calcularea numerelor $x_1, x_2, \ldots, x_n$.

Un caz particular al acestei recurenţe este $x_i = a_i + b_i x_{i-1}$ cu $b_i = 1$, care permite calcularea sumelor $a_1 + a_2 + \ldots + a_i$. Deci şi pentru problema sumei se poate folosi această tehnică.

Considerăm pentru uniformitate şi următoarele notaţii:

$$a_0 = \alpha, a_i = 0, i < 0$$
$$b_i = 0, i \leq 0.$$

Aplicând relaţia pentru x_{i-1}, obţinem:

$$x_i = a_i + b_i(a_{i-1} + b_{i-1}x_{i-2}).$$

Notăm $a_i^{(1)} = a_i + b_i a_{i-1}$ şi $b_i^{(1)} = b_i b_{i-1}$ şi astfel relaţia de mai sus devine:

$$x_i = a_i^{(1)} + b_i^{(1)} x_{i-2}.$$

Se aplică această nouă relaţie pentru x_{i-2} şi avem

$$x_i = a_i^{(1)} + b_i^{(1)}(a_{i-1}^{(1)} + b_{i-1}^{(1)} x_{i-4}) = a_i^{(2)} + b_i^{(2)} x_{i-4}.$$

Continuând în acelaşi mod obţinem exprimări succesive a lui x_i în funcţie x_{i-1}, x_{i-2}, x_{i-4}, ... Notând $a_i^{(0)} = a_i$ si $b_i^{(0)} = b_i$ avem relaţiile:

$$a_i^{(p)} = a_i^{(p-1)} + b_i^{(p-1)} a_{i-2^{p-1}}^{(p-1)}$$
$$b_i^{(p)} = b_i^{(p-1)} b_{i-2^{p-1}}^{(p-1)}$$

pentru orice $p > 0$. După cel mult k paşi – $(k = \lfloor \log_2 i \rfloor + 1)$ – indicele $i - 2^{k-1}$ va deveni 0 sau negativ şi $a_{i-2^{k-1}}^{(k)}$ va deveni egal cu α sau 0. Prin urmare, rezultatul x_i se obţine în $k = \lfloor \log_2 i \rfloor + 1$ paşi:

$$x_i = a_i^{(k)}$$

În Figura 4.5 se poate vedea modul de calcul pentru $n = 4$ al termenilor $b_i^{(p)}$ şi al termenilor $a_i^{(p)}$, (b) în cazul calculului direct şi (a) în cazul folosirii tehnicii de reducere ciclică par-impar. În final $x_4 = a_4^{(3)}$.

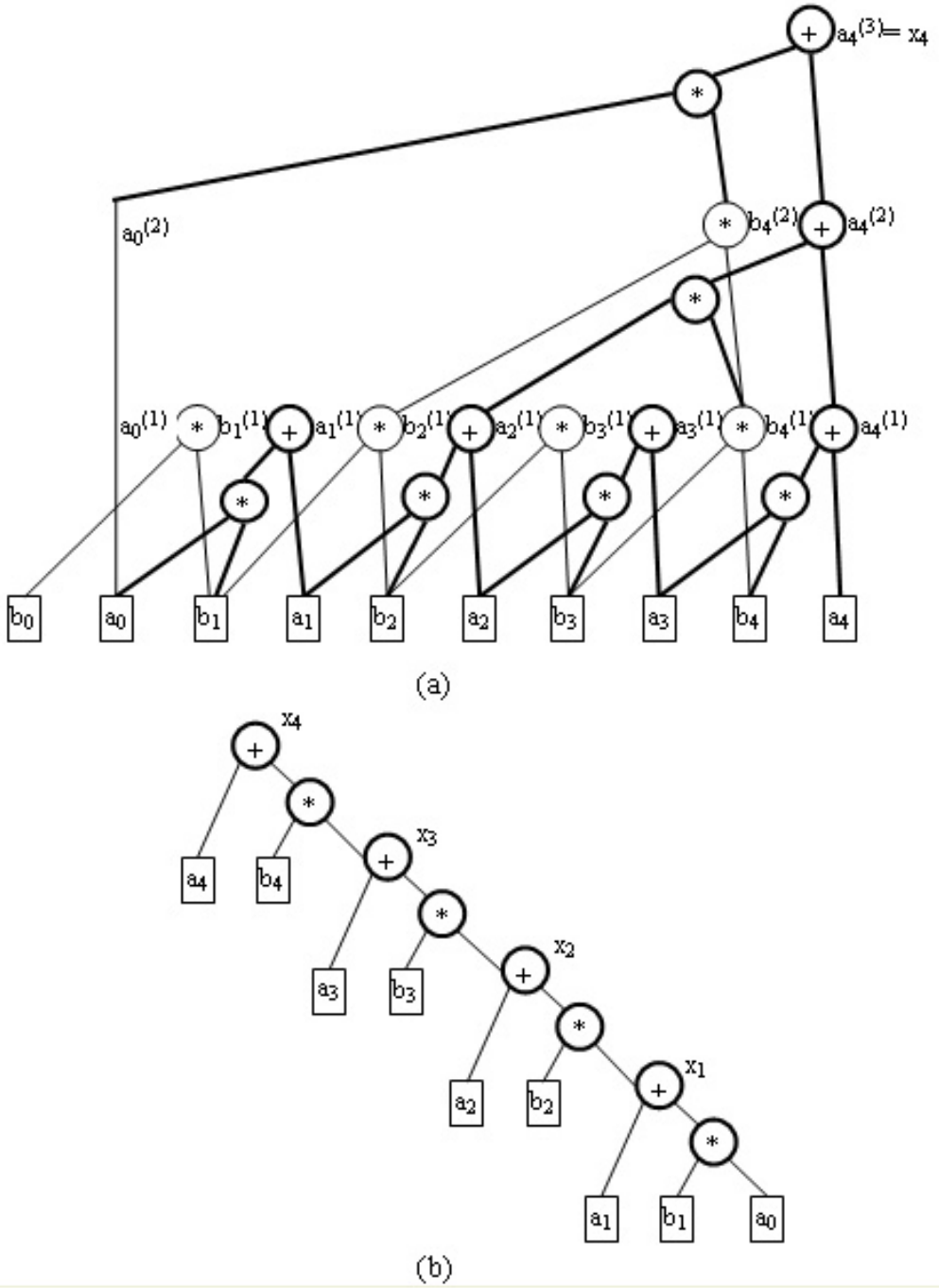

Figura 4.5: Reţele de calcul pentru relaţia de recurenţă de ordin $n = 4$. (a) Cazul folosirii reducerii ciclice par-impar. (b) Cazul calculului direct secvenţial.

ALGORITHM REDUCERE-CICLICA$< n = 2^r, a, b, x >$
  for $id = 0, n/2 - 1$ **in parallel do**
    for $p = 1, r$ **do**
      if $(id < n/2^p)$ **then**
        $i \leftarrow id * 2^p$;
        $b_i^{(p)} = b_i^{(p-1)} b_{i+2^{p-1}}^{(p-1)}$;
        $a_i^{(p)} = a_i^{(p-1)} + b_i^{(p-1)} a_{i+2^{p-1}}^{(p-1)}$;
      end if
    end for
  end for
  if $(i = 0)$ **then**
    $x_n \leftarrow b_0^r * x_0 + a_0^r$;
  end if

Exemplul 4.10 (Rezolvare sistem tridiagonal) Tehnica reducerii ciclice par-impar se poate utiliza şi pentru rezolvarea sistemelor liniare cu matrice tridiagonală.

Fie sistemul:

$$\begin{pmatrix} b_1 & c_1 & 0 & & & 0 \\ a_2 & b_2 & c_2 & & & 0 \\ 0 & a_3 & b_3 & c_3 & & 0 \\ & & \ddots & \ddots & \ddots & \\ & & & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & & & 0 & a_n & b_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{n-1} \\ d_n \end{pmatrix}$$

Pentru fiecare $i, 2 \leq i < n-1$ avem relaţiile:

$$\begin{aligned} a_{i-1}x_{i-2} + b_{i-1}x_{i-1} + c_{i-1}x_i &= d_{i-1} \\ a_i x_{i-1} + b_i x_i + c_i x_{i+1} &= d_i \\ a_{i+1}x_i + b_{i+1}x_{i+1} + c_{i+1}x_{i+2} &= d_{i+1} \end{aligned}$$

Prin eliminarea termenilor x_{i-1} şi x_{i+1} se obţine

$$a_i^{(1)} x_{i-2} + b_i^{(1)} x_i + c_i^{(1)} x_{i+2} = d_i^{(1)}.$$

Utilizând trei ecuaţii de acest tip se obţine

$$a_i^{(2)} x_{i-4} + b_i^{(2)} x_i + c_i^{(2)} x_{i+4} = d_i^{(2)}.$$

Formula generală este

$$a_i^{(k)} x_{i-2^k} + b_i^{(k)} x_i + c_i^{(k)} x_{i+2^k} = d_i^{(k)}.$$

unde coeficienţii se obţin din următoarele relaţii recursive:

$$\begin{aligned}
a_i^{(k+1)} &= \alpha_i^{(k)} a_{i-2^k}^{(k)}, \\
b_i^{(k+1)} &= \alpha_i^{(k)} c_{i-2^k}^{(k)} + b_i^{(k)} + \gamma_i^{(k)} a_{i+2^k}^{(k)}, \\
c_i^{(k+1)} &= \gamma_i^{(k)} c_{i+2^k}^{(k)}, \\
d_i^{(k+1)} &= \alpha_i^{(k)} d_{i-2^k}^{(k)} + d_i^{(k)} + \gamma_i^{(k)} d_{i+2^k}^{(k)},
\end{aligned}$$

unde $1 \leq k \leq \log_2 n$ şi

$$\alpha_i^{(k)} = -\frac{a_i^{(k)}}{b_{i-2^k}^{(k)}}, \quad \gamma_i^{(k)} = -\frac{c_i^{(k)}}{b_{i+2^k}^{(k)}}.$$

În final

$$b_i^{(k)} x_i = d_i^{(k)}, k = \lceil \log_2 n \rceil, \forall i : 1 \leq i \leq n.$$

Algoritmul poate fi implementat folosind o reţea de tip fluture, datorită faptului că pentru calcularea coeficienţilor $a_i^{(k)}, b_i^{(k)}, c_i^{(k)}, c_i^{(k)}$, se folosesc valorile anterioare cu indici $i - 2^p$ şi $i + 2^p$, pe rând pentru $p = 0, 1, \ldots \lceil \log_2 k \rceil$.

4.6 Tehnica divide&impera

Metoda "divide&impera" este binecunoscută din calculul secvenţial. Rezolvarea problemei se face prin împărţirea, de o manieră recursivă, în subprobleme din ce în ce mai mici, pe cale descendentă, până la atingerea cazului de bază sau elementar. Se rezolvă subproblemele de la acest nivel, după care se combină rezultatele, pe o cale ascendentă. Prin divizarea în subprobleme, la fiecare nivel, se creează un arbore (care este binar dacă descompunerea se face de fiecare dată în două subprobleme). Acest arbore este parcurs descendent când se efectuează apelurile procedurii recursive şi în sens ascendent, la revenirea din apel. Soluţia problemei va fi furnizată, în final, în nodul radăcină. În varianta secvenţială, la un moment dat, este parcurs un singur nod al arborelui. Execuţia paralelă oferă posibilitatea parcurgerii simultane a mai multor noduri, toate de la acelaşi nivel.

Pentru multe probleme, tehnica divide&impera conduce la algoritmi similari cu cei obţinuţi prin tehnica arborelui binar. Explicaţia acestui lucru este oarecum evidentă dacă ţinem cont că tehnica divide&impera conduce la realizarea unei reţele de calcul de tip arbore binar. Tehnica arborelui binar nu include şi parcurgerea descendentă a arborelui, care este de fapt corespunzătoare descompunerii calculului; de aceea algoritmii bazaţi pe tehnica arborelui binar sunt nerecursivi.

Exemplul 4.11 (Sortare prin interclasare)

Sortarea prin interclasare, descompune recursiv secvenţa iniţială A în două subsecvenţe de dimensiuni aproximativ egale A_0 şi A_1, pentru care se apelează acelaşi algoritm de sortare. Etapa de combinare foloseşte interclasarea a două secvenţe sortate. Complexitatea-timp pentru interclasarea a două secvenţe sortate de dimensiuni n_0 şi n_1 este $n_0 + n_1$. Dacă $n = 2^k$ atunci la nivelul $\log_2 n$ se interclasează secvenţe de dimensiune 1, în paralel;

la nivelul $\log_2 n - 1$ secvenţe de dimensiune 2. La nivelul 1 se interclasează secvenţe de dimensiune $n/2$. Complexitatea-timp obţinută este $2 + 2^2 + \ldots + 2^k = 2^k - 2 = O(n)$.

PROCEDURE INTERSORT(A, n)
  if $(n > 1)$ **then**
    IMPARTE$(A, n, A_0, n_0, A_1, n_1)$;
    in parallel
      INTERSORT(A_0, n_0),
      INTERSORT(A_1, n_1)
    end in parallel
    COMBINA$(A_0, n_0, A_1, n_1, A, n)$;
  end if

Exemplul 4.12 (Suma) Considerăm secvenţa A de n numere şi $m = n/p$, unde p este

---

FUNCTION SUMADIV(A, s, d, m)
  if $(d - s \geq m)$ **then**
    $i = (s + d)/2$;
    in parallel
      $s_0 \leftarrow$SUMADIV(A, s, i, m),
      $s_1 \leftarrow$SUMADIV$(A, i + 1, d, m)$
    end in parallel
    SUMADIV$\leftarrow s_0 + s_1$;
  else
    $st \leftarrow 0$;
    for $i = s, d$ **do**
      $st \leftarrow st + a[i]$;
    end for
    SUMADIV$\leftarrow st$
  end if

---

numărul maxim de componente de calcul care pot fi folosite. Pentru calcularea sumei celor n numere se apelează funcţia SUMADIV cu parametrii actuali $(A, 1, n, m)$.

Acest calcul este echivalent cu cel prezentat în Exemplul 4.25.

Tehnica divide&impera poate fi folosită şi pentru o descompunere a calculului pe mai multe căi nu doar două.

Exemplul 4.13 (Căutare) Se consideră o secvenţă de n numere A. Se cere căutarea numărului x în această secvenţă; dacă numărul x se găseste în secvenţă, atunci funcţia

CautaParalel returnează prima poziţie pe care a fost găsit, în caz contrar -1. (Considerăm o constantă $m, m > 2$, care reprezintă numărul minim de elemente pentru care se apelează algoritmul paralel.)

```
Function CautaParalel(A, n, x)
  if (n > m) then
    imparte(A, n, A[0], n[0], A[1], n[1], A[2], n[2]);
    for i = 0, 2 in parallel do
      c[i] ← CautaParalel(A[i], n[i], x);
    end for
    if (c[0] ≠ −1) then
      CautaParalel← c[0];
    else
      if (c[1] ≠ −1) then
        CautaParalel← n[0] + c[1];
      else
        if (c[2] ≠ −1) then
          CautaParalel← n[0] + n[1] + c[2];
        else
          CautaParalel← −1;
        end if
      end if
    end if
  else
    CautaParalel ← CautaSecvential(A, n, x);
  end if
```

Funcţia CautaSecvential implementează o căutare secvenţială clasică.
Complexitatea acestui algoritm este $O(\log_3 n)$.

Exemplul 4.14 (Quicksort) Algoritmul de sortare rapidă – Quicksort – este printre cei mai rapizi algoritmi secvenţiali. Ideea sortării rapide este simplă şi se bazează pe tehnica divide&impera: se împarte secvenţa iniţială A în două părţi; într-una sunt elementele mai mici decât un pivot v, iar în cealaltă elementele mai mari. Sortându-se cele două secvenţe, A va fi şi ea sortată.

Paralelizarea directă a acestui algoritm conduce la apelul simultan al procedurilor de sortare a subsecvenţelor rezultate în urma partiţionării. Totuşi şi etapa de partiţionare poate fi realizată în paralel şi, de fapt, se consideră că o implementare eficientă a sortării rapide în paralel, depinde de alegerea unui algoritm paralel eficient de partiţionare.

```
PROCEDURE QUICKSORT(A, s, d)
  if (s < d) then
    ALEGE-PIVOT(A, s, d, v);
    PARTITIONARE(A, s, d, v, i_v);
    in parallel
      QUICKSORT(A, s, i_v − 1),
      QUICKSORT(A, i_v + 1, d)
    end in parallel
  end if
```

Un exemplu de algoritm paralel de partiţionare se obţine dacă se consideră alegerea pivotului median (elementul de la mijlocul secvenţei) şi executarea comparaţiilor cu acesta în paralel.

Fiind o tehnică extrem de mult folosită în proiectarea algoritmilor paraleli, divide&impera este recunoscută ca o paradigmă a programării paralele.

4.7 Algoritmii generici ASCEND şi DESCEND

Corespunzător algoritmilor de tip arbore binar se pot defini doi algoritmi generici ASCEND şi DESCEND, corespunzători traversării arborelui într-un sens şi în celălalt.

Presupunem că avem $n = 2^k$ elemente date stocate în locaţiile $T[0], T[1], \ldots, T[n-1]$. Notăm cu $OPER(m, j; U, V)$ o operaţie care modifică datele din locaţiile U şi V şi depinde de parametrii m şi j, unde $0 \leq m < n$ şi $0 \leq j < k$. Considerăm, de asemenea, funcţia $bit_j(m)$ care returnează al-j-lea bit al reprezentării numărului m în baza 2.

```
ALGORITHM DESCEND
  for j = k − 1, 0, −1 do
    for < fiecare m pentru care 0 ≤ m < n > in parallel do
      if bit_j(m) = 0 then
        OPER(m, j; T[m], T[m + 2^j]);
      end if
    end for
  end for
```

Aceşti doi algoritmi generici pot constitui un punct de plecare în construirea unui algoritm paralel pentru o problemă dată. Dacă putem determina instanţe ale operaţiei $OPER$ care să implice executarea calculelor cerute de specificaţia problemei (şi această implicaţie poate fi demonstrată), atunci algoritmul paralel se construieşte uşor pe baza şablonului oferit către unul din aceşti doi algoritmi.

ALGORITHM ASCEND
for $j = 0, k-1$ **do**
  for $<$ *fiecare* m *pentru care* $0 \leq m < n >$ **in parallel do**
    if $bit_j(m) = 0$ **then**
      $OPER(m, j; T[m], T[m + 2^j])$;
    end if
  end for
end for

În multe cazuri, algoritmii paraleli nu se încadrează în nici una din clasele de mai sus, dar pot fi descompuşi în secvenţe de algoritmi care se încadrează.

Operaţia $OPER(m, j; T[m], T[m + 2^j])$ poate fi de multe ori privită ca o aplicaţie a unei perechi de funcţii (f_1, f_2):

$$T[m] \leftarrow f_1(m, j; T[m], T[m + 2^j])$$
$$T[m + 2^j] \leftarrow f_2(m, j; T[m], T[m + 2^j])$$

Dacă complexitatea operaţiei $OPER$ este constantă atunci complexitaţile celor doi algoritmi sunt egale cu $O(\log_2 n)$.

Un exemplu simplu şi direct de instanţă a unui algoritm ASCEND este algoritmul de calcularea a sumei bazat pe reţeaua de calcul arbore binar. Este un algoritm ASCEND pentru care operaţia $OPER(m, j; U, V)$ are următorul efect:

$$U \leftarrow U$$
$$V \leftarrow U + V$$

Exemplul 4.15 (Permutare inversă) Considerăm din nou $n = 2^k$ date de intrare. Se cere permutarea acestora în ordine inversă. Algoritmul DESCEND cu operaţia $OPER(m, j; U, V)$ care interschimbă valorile U şi V, conduce la permutarea datelor de intrare în ordine inversă.

Demonstraţia se poate face prin inducţie după k. În mod evident este adevărat dacă $k = 1$. Presupunem că este adevărat pentru toţi $k \leq t$ şi demonstrăm proprietatea pentru cazul $k = t + 1$. Cheia demonstraţiei este de a considera că $T[i] = i, 0 \leq i < n$. Dacă facem demonstraţia pentru acest caz, atunci concluzia va fi adevărată pentru toate cazurile deoarece valorile numerice ale datelor de intrare nu intervin în calcul. Primul pas al algoritmului DESCEND modifică doar bitul de ordin cel mai mare al datelor de intrare, astfel încât acest bit este pentru $T[i]$ egal cu bitul de ordin cel mai mare al lui $n - 1 - i$. Ceilalţi biţi de la 0 la t nu se modifică şi de asemenea biţii de la 0 la t ai numerelor $\{0, \ldots, n/2 - 1\}$ sunt aceiaşi ca şi ai numerelor $\{n/2, \ldots, n - 1\}$. Următoarele iteraţii ale algoritmului corespund aplicării algoritmului în cazul $k = t$, independent, pe secvenţele stânga şi dreapta ale secvenţei iniţiale. Rezultă deci că, după execuţia algoritmului, $T[i] = n - 1 - i$.

Implementarea pe reţeaua fluture

Considerăm o reţea fluture de rang k. Algoritmii ASCEND şi DESCEND pot fi adaptaţi foarte uşor pentru acest tip de reţea. Algoritmii corespunzători vor specifica programul executat de fiecare procesor.

Presupunem că avem cele $n = 2^k$ date de intrare $T[i], 0 \leq i < n$ astfel încât $T[i]$ aparţine procesorului $d_{0,i}$ al reţelei de tip fluture, pentru fiecare i cu $0 \leq i < n$. Varianta algoritmului DESCEND pe această reţea este:

ALGORITHM DESCEND-FLUTURE

for $j = k-1, 0, -1$ **do**

$<$ *se transmit datele de la procesoarele de rang* $k-1-j$ *către cele de rang* $k-j$ *pe verticală şi pe diagonală*$>$

{acum procesoarele de rang $k-j$ de pe coloana m şi $m+2^j$ conţin vechile valori $T[m]$ şi $T[m+2^j]$}

$OPER(m, j; T[m], T[m+2^j])$;

end for

Pentru algoritmul ASCEND presupunem că data $T[i]$ aparţine procesorului $d_{k,i}$ al reţelei de tip fluture, pentru fiecare i cu $0 \leq i < n$. Varianta algoritmului ASCEND pe această reţea este:

ALGORITHM ASCEND-FLUTURE

for $j = 0, k-1$ **do**

$<$ *se transmit datele de la procesoarele de rang* $k-j$ *către cele de rang* $k-j-1$ *pe verticală şi pe diagonală*$>$

{acum procesoarele de rang $k-j-1$ de pe coloana m şi $m+2^j$ conţin vechile valori $T[m]$ şi $T[m+2^j]$}

$OPER(m, j; T[m], T[m+2^j])$;

end for

Timpul de execuţie al celor doi algoritmi originali se multiplică cel mult cu un factor de multiplicare.

Exemplul 4.16 (Suma) Considerăm $n = 2^k$ numere $A[0], \ldots, A[n-1]$ şi presupunem că avem o reţea fluture de ordin k. Dorim să calculăm suma tuturor acestor numere, iar rezultatul să fie obţinut în toate procesoarele de rang k. Dacă considerăm algoritmul ASCEND cu operaţia $OPER(m, j; U, V)$ instanţiată astfel încât

$$U \leftarrow U + V$$
$$V \leftarrow U + V$$

obţinem în final că locaţiile $A[i]$ vor conţine suma totală. Diferenţa, faţă de algoritmul pe reţeaua arbore binar pentru sumă, este că rezultatul se obţine în toate procesoarele de rang k, nu doar într-un singur procesor rădăcină.

Implementarea pe reţeaua hipercub

Considerăm o reţea hipercub de ordin $n = 2^k$ şi că datele de intrare $T[0], \ldots, T[n-1]$ sunt asociate fiecare nodului cu indicele corespunzător din hipercub.

Implementarea celor doi algoritmi pe această reţea se poate face simplu adaptând cei doi algoritmi.

ALGORITHM DESCEND-HIPERCUB

for $j = k-1, 0, -1$ **do**
  for $<$ *fiecare* m *pentru care* $0 \leq m < n>$ **in parallel do**
    if $bit_j(m) = 0$ **then**
      $<$*procesoarele din nodurile* m *şi* $m + 2^j$ *interschimbă valorile datelor pe care le conţin*$>$;
      $OPER(m, j; T[m], T[m + 2^j])$;
    end if
  end for
end for

ALGORITHM ASCEND-HIPERCUB

for $j = 0, k-1$ **do**
  for $<$ *fiecare* m *pentru care* $0 \leq m < n>$ **in parallel do**
    if $bit_j(m) = 0$ **then**
      $<$ *procesoarele din nodurile* m *si* $m + 2^j$ *interschimbă valorile datelor pe care le conţin*$>$;
      $OPER(m, j; T[m], T[m + 2^j])$
    end if
  end for
end for

Implementările algoritmilor ASCEND şi DESCEND pe hipercub sunt mai directe decât cele pe reţeaua fluture. Ele sunt implementări eficiente ale acestor doi algoritmi generici.

În concluzie, algoritmii care se încadrează în cele două clase descrise de către algoritmii generici ASCEND şi DESCEND pot fi implementaţi eficient pe reţelele fluture şi hipercub.

4.8 Tehnica calculului sistolic (pipeline)

Una dintre cele mai folosite paradigme pentru programarea paralelă pe arhitecturi cu memorie nepartajată este calculul sistolic. Conceptul cheie al algoritmilor sistolici este descompunerea întregului calcul în subcomponente care sunt atribuite câte unui element de procesare. Comunicaţia datelor de-a lungul elementelor de procesare se face între elemente vecine. Se pot considera două proprietăţi importante ale calculul sistolic:

i. *localizarea comunicaţiilor* care impune ca fiecare element de procesare să comunice cu o mulţime mică de alte elemente de procesare,

ii. *structura de comunicaţii regulată.*

În cazul conectării elementelor de procesoare pe o singură dimensiune, astfel încât ieşirea unui element de procesare reprezintă intrare pentru următorul element de procesare; tehnica este întâlnită, în general, sub numele de **tehnica pipeline**.

Tehnica pipeline se bazează pe presupunerea că o anumită componentă a programului foloseşte rezultatele primite de la precedenta şi-şi trimite propriile rezultate următoarei componente. Fiecare componentă poate fi ataşată unui element de procesare. Dacă aplicaţia trebuie executată de mai multe ori, atunci mai multe procesoare pot fi folosite în paralel, permiţând celui de al doilea procesor să lucreze pe componenta a i-a aplicaţiei, în timp ce primul lucrează pe a $(i+1)$-a, iar al treilea pe a $(i-1)$-a, etc.

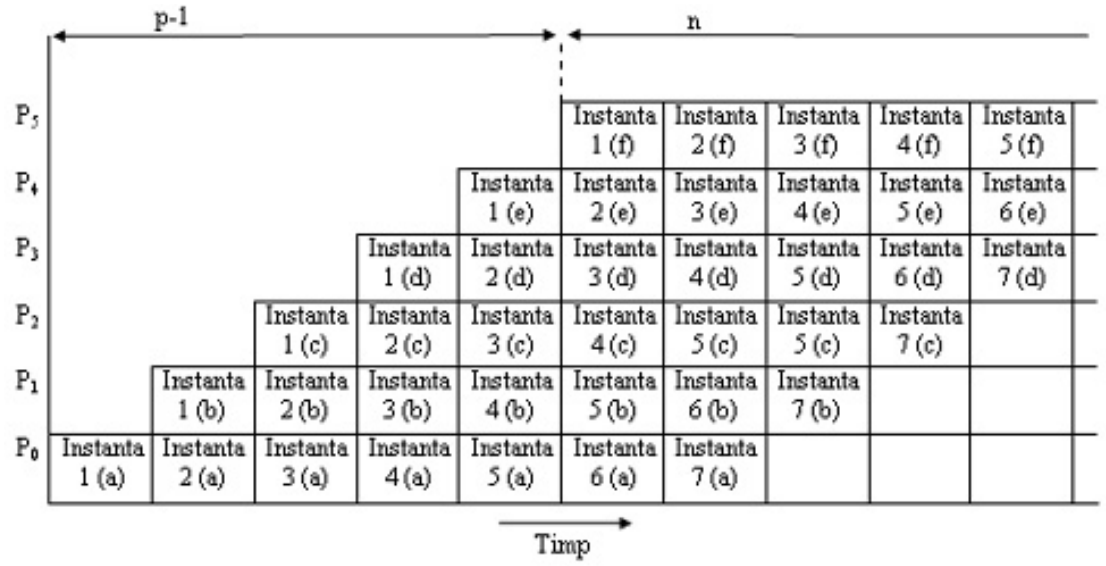

Figura 4.6: Tipul 1 de calcul pipeline. Pentru fiecare instanţă se execută componentele de calcul (a), (b), (c), (d), (e), (f). Timpul de execuţie este proporţional cu $n+p-1$.

Timpul de execuţie al unei singure aplicaţii pe un calculator pipeline nu e mai mic decât timpul de execuţie pe un singur procesor, dar în cazul unui număr mare de execuţii ale aplicaţiei pe un calculator pipeline cu N procesoare, atunci timpul de execuţie va fi

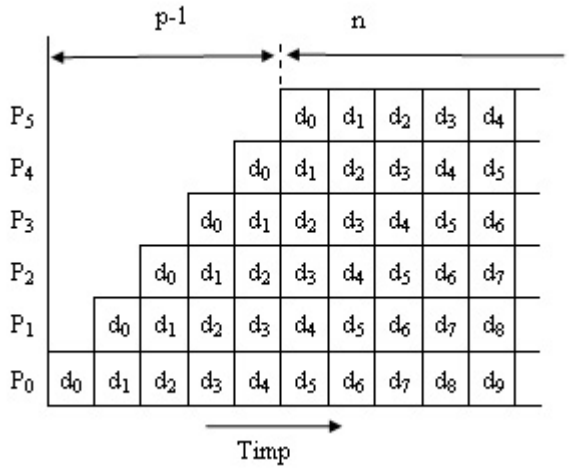

Figura 4.7: Tipul 2 de calcul pipeline. Fiecare instanţă se execută pentru mai multe date de intrare $d_0, d_1, \ldots$.

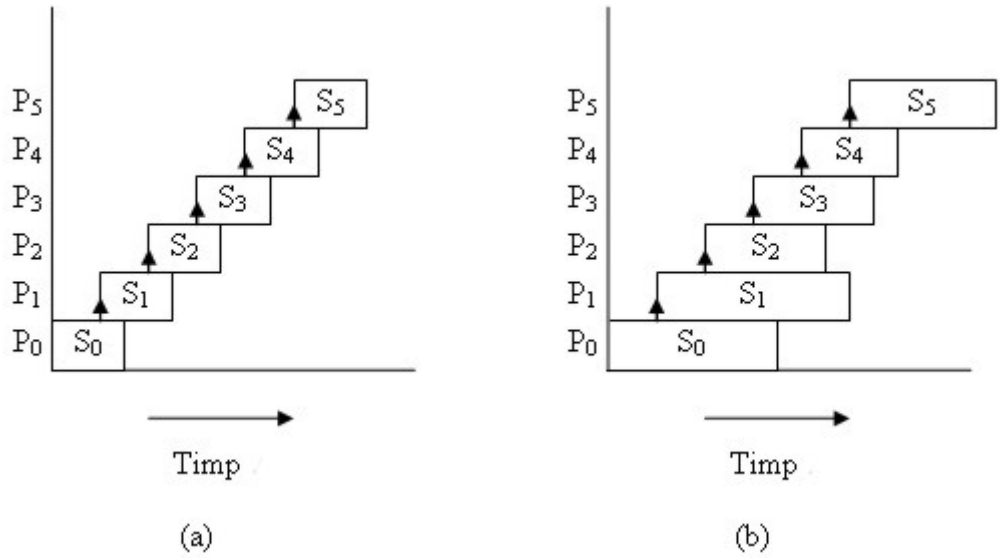

Figura 4.8: Tipul 3 de calcul pipeline. (a) Componente de calcul egale. (b) Componente de calcul inegale.

aproximativ egal cu $1/N$ din timpul serial necesar (presupunând că fiecare componentă a calculului necesită o cuantă de timp egală).

Timpul necesar iniţializării unui pipeline este egal cu timpul (T) necesar unei etape din pipeline pentru a fi executată, multiplicat cu numărul de etape din pipeline. Îndată ce pipeline-ul a fost iniţializat, fiecare rezultat nou se obţine într-un timp egal cu T. De exemplu, un calculator pipeline cu trei procesoare, fiecare lucrând T secunde, va genera primul rezultat în $3T$ secunde, noi rezultate fiind obţinute apoi după fiecare T secunde.

În concluzie, presupunând că problema ce trebuie rezolvată poate fi divizată în componente de calcul secvenţiale, folosirea tehnicii pipeline conduce la algoritmi eficienţi dacă cel puţin una din următoarele variante este adevărată:

1. Mai mult de o instanţă a problemei trebuie să fie executată – Figura 4.6.
2. O serie de date(elemente) trebuie procesate, fiecare procesare necesitând operaţii multiple – Figura 4.7.
3. Dacă informaţia necesară începerii următoarei componente de calcul poate fi transmisă înainte ca procesul curent să îşi finalizeze execuţia – Figura 4.8.

Vom prezenta exemple care corespund celor trei tipuri şi pentru care vom considera programe cu p procese, în care fiecare proces este identificat de către un indice $id, 0 \leq id < p$. Toate procesele vor fi executate în paralel.

Exemplul 4.17 (Suma) Dorim să calculăm suma a p numere $\sum_{i=0}^{p-1} a_i^k$, pentru $n, n > 1$ seturi de mulţimi $a_i^k, 0 \leq k < n$. Pentru aceasta putem folosi un calcul de tip pipeline, cu p procese, în care fiecare proces citeşte(input) de pe câte un suport extern (fişier) numărul a_i^k corespunzător. Rezultatul fiecărei sume este scris(output) tot pe suport extern, de

---

ALGORITHM SUMA< id>
  input($numar$);
  if $(id > 0)$ **then**
    recv($accumulator, id - 1$);
  else
    $acumulator \leftarrow 0$;
  end if
  $acumulator \leftarrow acumulator + numar$;
  if $(id < p - 1)$ **then**
    send($accumulator, id + 1$);
  else
    output($acumulator$);
  end if

---

către ultimul proces. Fiecare proces cu identificatorul id va executa subprogramul SUMA de n ori – este prin urmare o exemplificare a primului tip de calcul pipeline.

Exemplul 4.18 (Generare numere prime) Pentru generarea numerelor prime mai mici decât un număr dat m se poate folosi un algoritm paralel de tip pipeline, bazat pe ideea "ciurului lui Eratostene". Primul număr: 2 este introdus în pipeline şi reţinut în primul proces, fiind număr prim. Cel de-al doilea număr intră în pipeline şi se verifică dacă se divide cu valoarea din primul proces, iar dacă nu, este trimis la procesul următor care îl reţine. Se continuă în acest mod. Dacă un număr primit de către un proces se divide cu valoare stocată de acesta atunci acel număr se ignoră.

ALGORITHM PRIME$< id, x >$
if $(id = 0)$ **then**
input$(numar)$;
else
recv$(numar, id - 1)$;
end if
if $x = 0$ **then**
$x \leftarrow numar$;
else
if $(numar\%x \neq 0)$ **then**
$trimis \leftarrow numar$;
if $(id < p - 1)$ **then**
send$(trimis, id + 1)$;
end if
end if
end if

Iniţial argumentul x al fiecărui proces este 0. Procesoarele nu efectuează acelaşi volum de calcul pentru că ele nu primesc aceeaşi cantitate de numere. La terminarea secvenţei este necesară transmiterea unui mesaj de terminare. De aceea, complexitatea nu poate fi evaluată după formula clasică corespunzătoare unui calcul de tip pipeline. Experimental, se poate însă verifica eficienţa acestui calcul.

Exemplul 4.19 (Sortare prin inserţie) Considerăm o secvenţă de n numere întregi care trebuie sortată descrescător. Considerăm $p = n$ procese, fiecare dintre ele având o valoare stocată local. Iniţial această valoare este egală cu cel mai mic număr întreg reprezentabil. Numerele din secvenţă vor fi "pompate" în pipeline, rând pe rând. La primirea unei valori, un procesor verifică dacă valoarea sa locală este mai mică decât cea primită, caz în care îşi actualizează valoarea locală cu numărul primit şi trimite mai departe vechea sa valoare locală. Dacă numărul primit este mai mic decât valoarea sa locală, acesta este pur şi simplu trimis mai departe.

Complexitatea algoritmului obţinut este $O(n)$. Sortarea se face în acest caz folosind

```
ALGORITHM SORT< id, x>
  if (id = 0) then
    input(numar);
  else
    recv(numar, id − 1);
  end if
  if (numar > x) then
    send(x, id + 1);
    x = numar;
  else
    send(numar, id + 1);
  end if
```

un singur ciclu. În cazul în care nu există suficiente procesoare pe care să se mapeze cele n procese, acestea pot fi grupate mai multe pe acelaşi procesor.

Exemplul 4.20 (Rezolvare sistem liniar triunghiular) Fie A o matrice triunghiulară de de ordin n şi B un vector cu n elemente. Se cere rezolvarea sistemului liniar $A \times X = B$:

$$\begin{aligned}
&a_{0,0}x_0 && = b_0\\
&a_{1,0}x_0 + a_{1,1}x_1 && = b_1\\
&a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 && = b_2\\
&\vdots\\
&a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 + \ldots + a_{n-1,n-1}x_{n-1} && = b_{n-1}
\end{aligned}$$

Pentru rezolvarea sistemului folosim substituţia înapoi si se obţine mai întâi x_0:

$$x_0 = \frac{b_0}{a_{0,0}},$$

apoi x_1, folosind valoarea x_0 deja calculata:

$$x_1 = \frac{b_1 - a_{1,0}x_0}{a_{1,1}}.$$

Se continuă în acest mod până când se obţine x_{n-1}:

$$x_{n-1} = \frac{b_{n-1} - \sum_{k=1}^{n-2} a_{n-1,k}x_k}{a_{n-1,n-1}}.$$

```
Algorithm Sistem< id, x, a, b>
  suma ← 0;
  if (id = 0) then
    x[0] ← b[0]/a[0][0];
    send(x[0], 1);
  else
    for i = 0, id − 1 do
      recv(x[i], id − 1);
      send(x[i], id + 1);
      suma ← suma + a[id][i] ∗ x[i];
    end for
    x[id] ← (b[id] − suma)/a[id][id];
    send(x[id], id + 1);
  end if
```

Prima componentă a soluţiei pipeline calculează x_0 şi trimite valoarea x_0 următoarei componente. Aceasta calculează x_1 şi trimite mai departe atât pe x_0 cât şi pe x_1. Acest tip de calcul se încadrează în tipul 3 de pipeline cu componente inegale. Componenta a doua poate trimite valoarea x_0 componentei a treia înainte de a calcula valoarea pentru x_1 şi astfel componenta a treia poate începe calculul înainte ca şi componenta a doua să îl termine. În acest fel se obţine paralelismul.

Fiecare procesor execută procedura Sistem doar o singură dată, nu este un apel repetat ca şi în cazul celorlalte tipuri de calcul pipeline.

În cazul conectării elementelor de procesoare pe două sau mai multe dimensiuni se ajunge la *tablouri sistolice*. S-au dezvoltat numeroşi algoritmi paraleli pentru aplicaţii din analiza numerică, teoria grafurilor şi procesarea imaginilor care se bazează pe această tehnică.

Exemplul 4.21 (Înmulţire matrice) Considerăm un tablou sistolic de $n \times n$ procese, conectate într-o reţea de tip grilă, şi două matrice pătratice A şi B de dimensiune $n \times n$. Procesele sunt identificate printr-o pereche $(i, j), 0 \leq i, j < n$. Matricea produs $C = A \times B$ se poate obţine folosind un algoritm sistolic. Elementele matricei A sunt "pompate" pe linii, iar elementele matricii B sunt "pompate" pe coloane. Elementul $C[i, j]$ se obţine în procesul (i, j).

Iniţial argumentul c este egal cu 0 în toate cele n^2 procese. Argumentul c din procesul (i, j) reprezintă elementul $C[i, j]$. Operaţiile de intrare ale elementelor $a[i, j]$ şi $b[i, j]$ trebuie să se facă cu o întârziere între două linii, respectiv coloane, consecutive – Figura 4.9.

În general, programele dezvoltate pe baza acestei tehnici sunt implementate pe arhitecturi specifice (sistolice) care să permită obţinerea eficienţei scontate. Totuşi ea poate fi

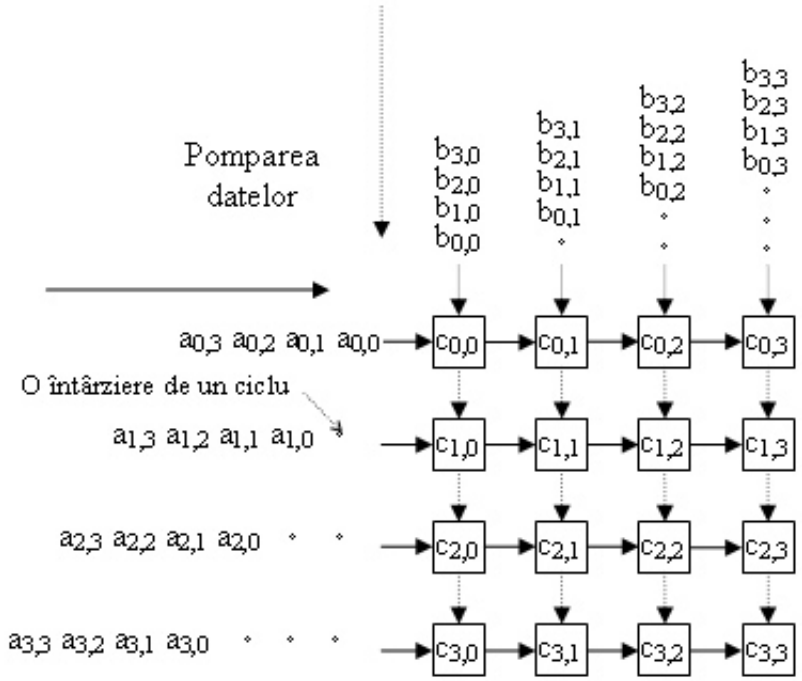

Figura 4.9: Înmulţire matriceală pe un tablou sistolic.

ALGORITHM PRODUS_MATRICE$< i, j, c >$

if $(j = 0)$ **then**
  input(a);
else
  recv$(a, (i, j-1))$;
end if
if $(i = 0)$ **then**
  input(b);
else
  recv$(b, (i-1, j))$;
end if
$c \leftarrow c + a * b$

---

folosită şi ca tehnică generală de partiţionare, aplicabilă şi pentru alte modele de sisteme de calcul paralel. Eficienţa în acest caz depinde de granularitatea aplicaţiei obţinute în raport cu granularitatea sistemului şi de timpul necesar comunicaţiei între două procese vecine.

4.9 Tehnica par-impar

O tehnică foarte simplă, care s-a dovedit a fi însă eficientă în multe cazuri, constă în separarea calculelor asupra datelor cu indice par de cele cu indice impar. Această separare a operării asupra elementelor cu indice impar de cele cu indice par poate determina mărirea numărului de operaţii independente – deci care se pot executa în paralel – prin înlăturarea dependenţelor care există între elementele consecutive. Este o tehnică care se aplică în special pentru algoritmi de sortare bazaţi pe operaţii de comparare-interschimbare, dar nu numai.

Exemplul 4.22 (Sortare par-impar) Considerând o secvenţă A de n numere dorim sortarea acesteia în ordine crescătoare. Sortarea par-impar foloseşte ca operaţie de bază *comparaţie-interschimbare* între două elemente succesive. (Este un algoritm derivat din algoritmul corespunzător metodei de sortare secvenţială "Bubble-Sort".)

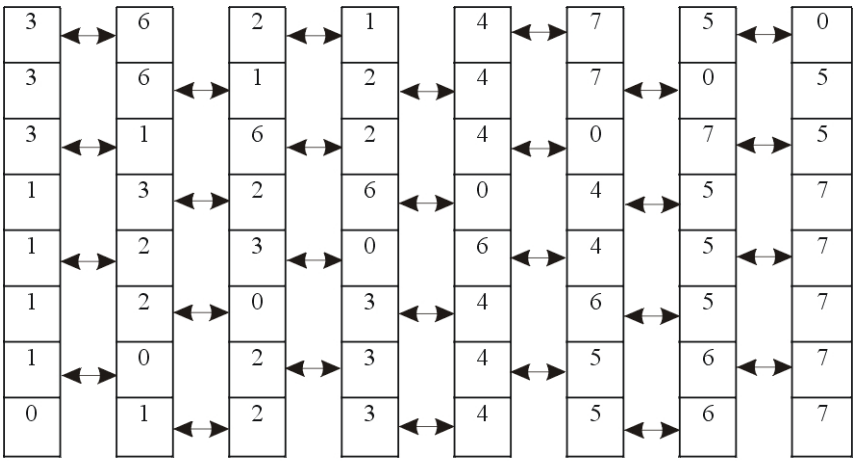

Figura 4.10: Sortare prin transpoziţie par-impar.

La prima etapă, elementele cu indice par sunt comparate cu cele din dreapta lor, iar unde este necesar se fac şi interschimbările corespunzătoare. La următoarea etapă, se compară elementele cu indice impar cu elementele din dreapta lor şi se fac interschimbările necesare. Figura 4.10 evidenţiază aceste etape.

Prin această separare între elementele cu indice par de cele cu indice impar, se asigură faptul că un element nu este folosit la o etapă în mai mult de o comparaţie şi de asemenea

---

ALGORITHM SORTARE_PAR_IMPAR$< n, A[0..n-1]>$

for $k = 0, \lceil n/2 \rceil$ **do**
  { pas par}
  for $i = 0, n-1, 2$ **in parallel do**
    if $i + 1 < n \wedge A[i] > A[i+1]$ **then**
      $A[i] \leftrightarrow A[i+1]$;
    end if
  end for
  { pas impar}
  for $i = 1, n-1, 2$ **in parallel do**
    if $i + 1 < n \wedge A[i] > A[i+1]$ **then**
      $A[i] \leftrightarrow A[i+1]$;
    end if
  end for
end for

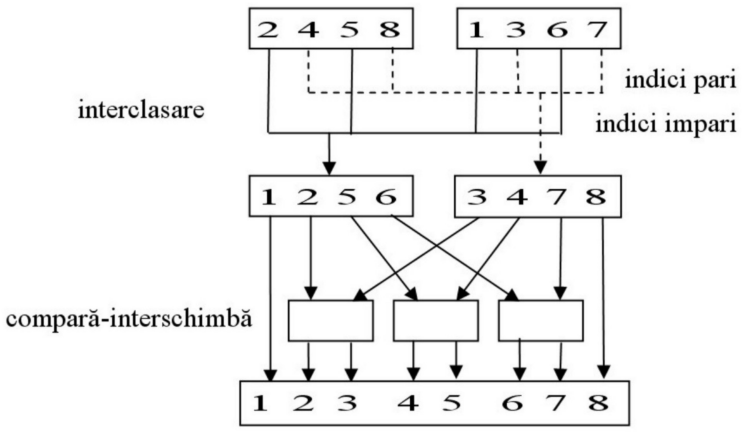

Figura 4.11: Interclasare par-impar a două liste sortate.

că, la fiecare etapă se fac numărul maxim de interschimbări posibile. La fiecare etapă formată din operaţii de comparare-interchimbare între elementele pare urmate de operaţii de comparare-interchimbare între elementele impare (pas par urmat de pas impar), un element "mare" se deplasează cu 2 poziţii. Prin urmare după $n/2$ etape cu siguranţă secvenţa va fi sortată. Fiecare etapă are complexitatea-timp egală cu 2 şi prin urmare complexitatea-timp de sortare este $O(n)$.

Exemplul 4.23 (Sortare prin interclasare par-impar) Tehnica par-impar poate fi folosită şi pentru a îmbunătăţi algoritmul paralel de sortare prin interclasare descris în Exemplul 4.11. Interclasarea care este necesară pentru combinarea a două subsecvenţe ordonate se va face de această dată prin operaţii de comparare-interschimbare între elementele de pe poziţiile impare, respectiv între cele de pe poziţiile pare – Figura 4.11. Dacă interclasăm elementele cu indicii impari din cele două secvenţe simultan cu interclasarea elementelor cu indicii pari, obţinem două secvenţe ordonate care pot fi apoi combinate prin operaţii de interschimbare executate simultan. În acest mod, timpul necesar interclasării devine logaritmic, iar complexitatea-timp a sortării prin metoda de interclasare par-impar devine $O(\log_2^2 n)$. Algoritmul de sortare prin interclasare par-impar a fost propus de către Batcher şi reprezintă un algoritm de sortare paralelă eficient din punct de vedere al costului.

Exemplul 4.24 (Rezolvarea unei ecuaţii diferenţiale prin metoda iterativă Gauss-Siedel) Criteriul par-impar se poate extinde şi la cazul bidimensional, caz în care considerăm vecinii direcţi ai unui punct, separaţi în două submulţimi distincte. Un exemplu elocvent este algoritmul Rosu-Negru folosit pentru aproximarea soluţiei unei ecuaţii diferenţiale în plan.

Fie ecuaţia Poisson

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = a,$$

şi condiţia de mărginire $u(x, y) = u_0(x, y)$ pe o curbă C.

Dacă considerăm regiunea de derivare divizată în dreptunghiuri de dimensiune $\partial x = h$ şi $\partial y = k$ şi reprezentăm coordonatele grilei prin $x = ih$ şi $y = jk$, atunci se pot face următoarele aproximări:

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i+1j} - 2u_{ij} + u_{i-1j}}{h^2},$$

unde $u_{ij} = u(ih, jk)$, iar

$$\frac{\partial^2 u}{\partial y^2} = \frac{u_{ij+1} - 2u_{ij} + u_{ij-1}}{h^2}.$$

Dacă $h = k$, se obţin ecuaţiile de aproximare cu diferenţe finite pentru ecuaţia Poisson:

$$u_{ij} == \frac{1}{4}(u_{i-1j} + u_{i+1j} + u_{ij-1} + u_{ij+1} - a_{ij}).$$

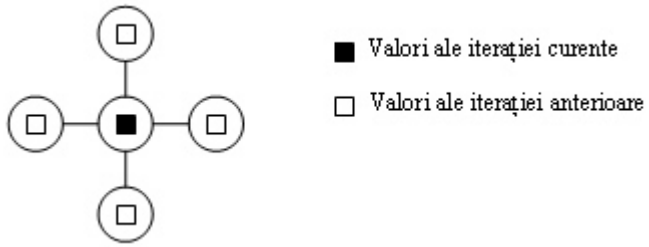

Figura 4.12: Actualizarea aproximărilor în metoda Jacobi.

Metoda Jacobi punctuală este definită de iteraţiile

$$u_{ij}^{(p+1)} == \frac{1}{4}(u_{i-1j}^{(p)} + u_{i+1j}^{(p)} + u_{ij-1}^{(p)} + u_{ij+1}^{(p)} - a_{ij})$$

şi se încadrează în problemele direct paralelizabile. Toate punctele pot fi actualizate simultan. Este necesară memorarea unei copii a situaţiei anterioare a grilei.

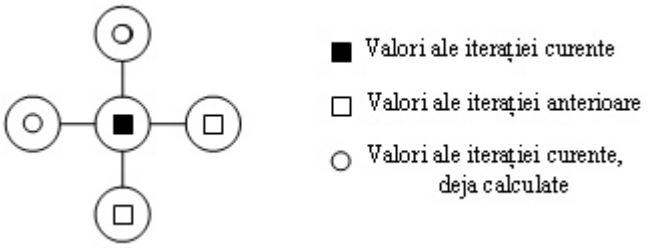

Figura 4.13: Actualizarea aproximărilor în metoda Gauss-Seidel.

În procesul iterativ se actualizează numai punctele interioare grilei. Condiţiile de frontieră sunt în principiu de două tipuri:

- Se specifică valorile funcţiei $u(x, y)$ pe frontiera domeniului de integrare (problema Dirichlet).
- Se specifică valorile derivatelor funcţiei $u(x, y)$ pe frontiera domeniului de integrare (problema Neumann).

Ambele pot fi tratate prin alegerea unor coeficienţi specifici pentru exprimarea lui $u_{ij}^{(p)}$.

O extensie naturală a metodei Jacobi este algoritmul Gauss-Seidel, care utilizează cele mai recente valori calculate pentru u, din partea dreaptă a punctului curent al grilei. Avantajul faţă de metoda anterioară constă în faptul că nu mai este necesar să se stocheze ultima valoare a lui u_{ij} şi se ajunge şi la o convergenţă mai rapidă. Iteraţia în cazul

Gauss-Seidel este definită prin

$$u_{ij}^{(p+1)} = \frac{1}{4}(u_{ij-1}^{(p+1)} + u_{i-1j}^{(p+1)} + u_{ij+1}^{(p)} + u_{i+1j}^{(p)} - a_{ij}).$$

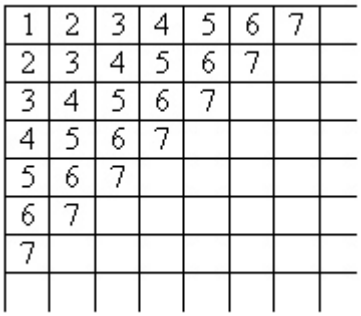

Figura 4.14: Avansarea frontului de aproximări în metoda Gauss-Seidel.

Metoda presupune o anumită ordine şi un front diagonal de lucru, precum se arată în Figura 4.14. La primul pas, punctul 1 este actualizat pe baza condiţiilor de mărginire şi estimărilor iniţiale pentru punctele 2. La pasul doi, punctele 2 sunt actualizate simultan utilizând estimările punctelor 3 şi valoarea calculată pentru punctul 1. La al treilea pas, punctele 3 sunt actualizate şi simultan se poate actualiza punctul 1 a doua oară, deoarece informaţia despre acestă aproximare nu mai este necesară. Prima iteraţie evoluează ca un val în grilă, iar următoarele valuri sunt iteraţiile care o succed.

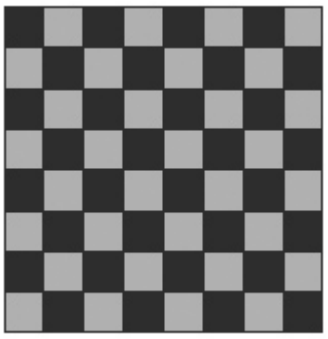

Figura 4.15: Colorarea grilei de noduri în cazul metodei roşu-negru.

Se preferă ca în loc să existe mai multe niveluri în curs de calculare, să se opereze doar la două niveluri – *metoda roşu-negru*. Pentru aceasta se consideră nodurile a fi colorate în roşu şi negru. Aşa cum se observă în Figura 4.15, colorarea se face alternativ, începând

cu prima linie, de la stânga la dreapta. Punctele negre se vor actualiza primele, utilizând punctele roşii anterioare, iar punctele roşii sunt actualizate folosind ultimele valori din nodurile negre.

ALGORITHM ROSU_NEGRU$< n, A[0..n-1][0..n-1], U[0..n-1][0..n-1] >$

```
for i = 1, n-2 in parallel do
  for j = 1, n-2 in parallel do
    if (i+j)%2 = 0 then
      u[i][j] = (u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1] - a[i][j])/4;
    end if
  end for
end for
for i = 1, n-2 in parallel do
  for j = 1, n-2 in parallel do
    if (i+j)%2 ≠ 0 then
      u[i][j] = (u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1] - a[i][j])/4;
    end if
  end for
end for
```

Punctele negre sunt actualizate la un nivel de iterare, iar punctele roşii la următorul nivel de iterare. După o iteraţie, punctele negre sunt la nivelul 1 de iteraţie ca şi în cazul metodei Jacobi, iar punctele roşii au fost deja actualizate şi sunt la nivelul 2. La o iteraţie, metoda Jacobi actualizează toate punctele, iar algoritmul roşu şi negru doar jumătate. Astfel pentru o matrice de $n \times n$ procesoare, o problemă de dimensiune $2n \times n$ poate fi rezolvată utilizând metoda Gauss-Siedel în acelaşi timp în care o problemă de dimensiune $n \times n$ poate fi rezolvată prin metoda Jacobi.

4.10 Prefix paralel – Scan

Fie o secvenţă S de n elemente şi un operator asociativ $\oplus$ definit pe tipul acestor elemente. Operaţia prefix calculează toate "sumele" de forma $S_i' = s_1 \oplus s_2 \oplus \ldots \oplus s_i$, pentru $1 \leq i \leq n$. Calculul secvenţial al acestei operaţii este redat de reţeaua de calcul din Figura 4.16.

Operaţia *prefix* este folosită în multe aplicaţii şi de aceea analiza paralelizării ei este importantă, ea constituind un şablon("skeleton") care se aplică în foarte multe aplicaţii. Operatorul $\oplus$ poate fi orice operator asociativ: plus, ori, minim, maxim, etc. Un exemplu foarte simplu, care foloseşte acest tip de calcul este calcularea factorialelor $1!, 2!, 3!, \ldots, n!$.

Acest calcul este atât de mult folosit încât s-a propus chiar includerea lui ca o primitivă în modelele PRAM [21]. Există foarte multe probleme în care această operaţie poate fi folosită; enumerăm aici doar câteva dintre acestea: sortare, interclasare, arbore de acoperire, componente conexe, flux maxim, rezolvare de sisteme liniare.

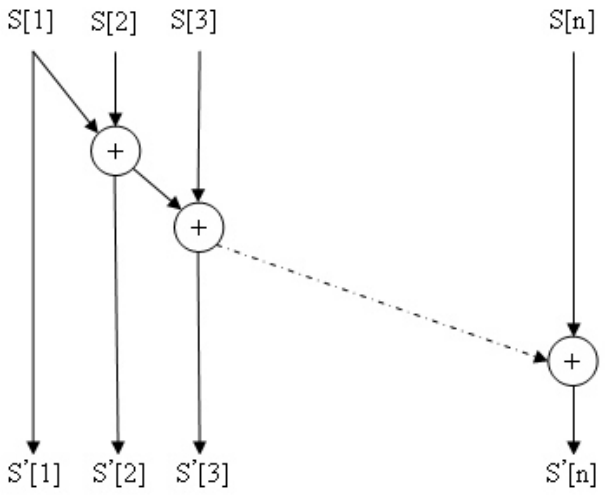

Figura 4.16: Reţeaua de calcul secvenţial pentru sumele prefix.

Fiind un calcul atât de des întâlnit, s-au dezvoltat mai multe variante de paralelizare, în încercarea de a se obţine o eficienţa cât mai mare. Vom prezenta aici trei dintre acestea: *sus-jos, par-impar* şi cea elaborată de *Ladner* şi *Fisher*. O altă variantă este derivată formal şi descrisă în Capitolul 6, Secţiunea 6.3.1. Pentru toate variantele care vor fi prezentate în continuare, argumentul de intrare este secvenţa iniţială de elemente – $S[i], 1 \leq i \leq n$, iar în final această secvenţă va conţine sumele prefix $S'_1, \ldots, S'_n$.

Prefix sus-jos ("upper-lower")

Acest algorim se bazează pe tehnica divide&impera, la fiecare pas secvenţa iniţială de numere fiind împărţită în două subsecvenţe de lungimi (aproximativ) egale. Divizarea se continuă până se ajunge la secvenţe cu cel mult 2 elemente. Pasul de combinare impune adunarea ultimului element al primei secvenţe (cea din stânga) la toate elementele secvenţei a doua (cea din dreapta). Reţeaua de calcul corespunzătoare pentru $n = 8$ este redată în Figura 4.17. Complexitatea-timp obţinută este de $\lceil \log_2 n \rceil$, considerând o complexitate procesor de $n/2$. Numărul total de operaţii efectuat de algoritm este $n/2(\log_2 n)$.

Subalgoritmul PREFIX_PARALEL_UL se apelează iniţial cu $a = 1$ şi $b = n$.

Prefix impar-par("odd-even")

Şi această variantă se bazează pe tehnica divide&impera, care este însă combinată cu tehnica impar-par. La fiecare etapă (cu excepţia cazurilor în care subsecvenţele au lungimea egală sau mai mică decât 4) calculele se împart în două subetape, una corespunzătoare indicilor pari şi cea de-a doua corespunzătoare indicilor impari.

Algoritmul constă în aplicarea următorilor paşi:

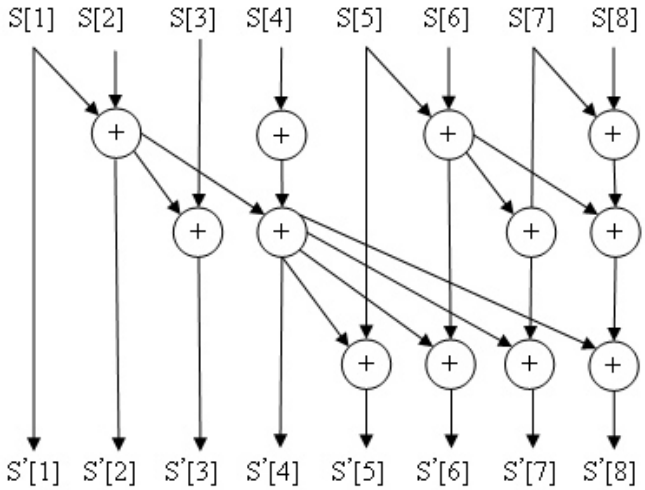

Figura 4.17: Reţeaua de calcul folosind algoritmul prefix sus-jos pentru o secvenţă de 8 elemente.

PROCEDURE PREFIX_PARALEL_UL(S, a, b)

if $b - a = 1$ **then**
  $S[b] \leftarrow S[a] \oplus S[b]$;
else
  if $b - a = 0$; **then**
    skip;
  else
    in parallel
      PREFIX_PARALEL_UL$(S, a, \lfloor (b-a)/2 \rfloor)$;
      PREFIX_PARALEL_UL$(S, \lfloor (b-a)/2 \rfloor + 1, b)$;
    end in parallel
    for $i = \lfloor (b-a)/2 \rfloor + 1, b$ **in parallel do**
      $S[i] \leftarrow S[\lfloor (b-a)/2 \rfloor] \oplus S[i]$;
    end for
  end if
end if

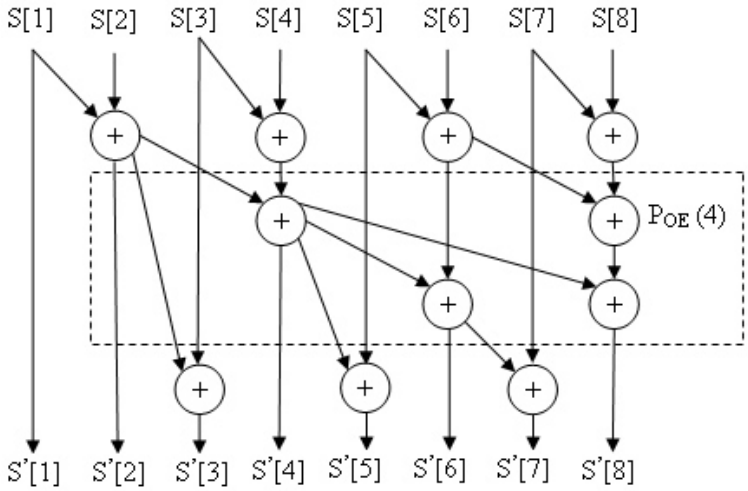

Figura 4.18: Reţeaua de calcul folosind algoritmul prefix impar-par pentru o secvenţă de 8 elemente.

PROCEDURE PREFIX_PARALEL_OE(S, a, b, pas)

if $b - a < 4$ **then**
  PREFIX_PARALEL_UL(S, a, b);
else
  for $i \leftarrow a + 2^{pas-1}, b, 2^{pas}$ **in parallel do**
    $S[i] \leftarrow S[i] \oplus S[i - 2^{pas-1}]$;
  end for
  PREFIX_PARALEL_OE($S, a + 2^{pas-1}, b, pas * 2$);
  for $i = a + 2^{pas}, b, 2^{pas}$ **in parallel do**
    $S[i] \leftarrow S[i - 2^{pas-1}] \oplus S[i]$;
  end for
end if

- Împarte secvenţa în două subsecvenţe – una formată din indicii impari, iar cealaltă formată din indicii pari.
- Combină elementele impare cu elementele pare imediat următoare.
- Apelează operaţia prefix pentru subsecvenţa formată din noile elemente de pe poziţiile pare.
- Combină elementele nou rezultate de pe poziţiile pare cu următoarele elemente de pe poziţii impare.

În cazul unei secvenţe de $n = 8$ elemente, se obţine reţeaua de calcul din Figura 4.18. În acest fel, complexitatea-timp creşte, dar calculele pot fi efectuate cu ajutorul a mai puţine procesoare. Pentru $n = 4$ sau mai mic se foloseşte tehnica anterioară.

Complexitatea-timp este $2\log_2 n - 2$, folosind tot $n/2$ procesoare. Algoritmul are avantajul că face mai puţine calcule (în total $2n - \log_2 n - 2$ faţă de $n/2\log_2 n$) şi aşadar se poate obţine o complexitate foarte bună cu mai puţine procesoare.

Subalgoritmul PREFIX_PARALEL_OE se apelează iniţial cu $a = 1$, $b = n$ şi $pas = 1$.

Prefix Ladner-Fisher

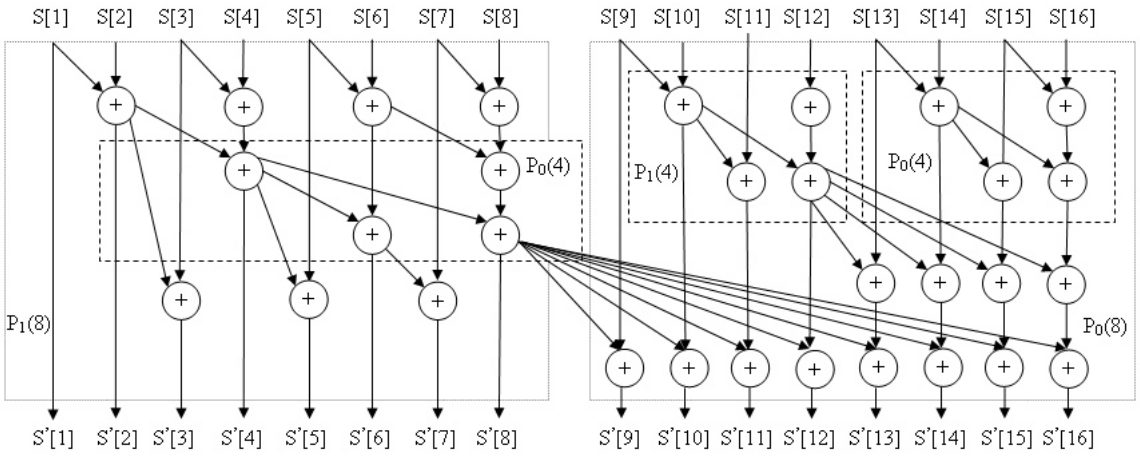

Figura 4.19: Reţeaua de calcul folosind algoritmul prefix Ladner-Fisher pentru o secvenţă de 16 elemente.

Algoritmul următor a fost propus de R.E Ladner şi M.J Fisher şi are avantajul unei complexităţi foarte bune, cu un număr total de operaţii scăzut [103]. Aceasta conduce la o eficienţă foarte bună şi în cazul folosirii unui număr mai mic de procesoare. Varianta aceasta combină cele două de mai înainte.

Se poate face o generalizare prin care se defineşte o clasă de algoritmi $P_j(n)$, pentru suma prefix. Aceşti algoritmi se definesc astfel:

- Pentru $j \geq 1$, $P_j(n)$ este definit de construcţia impar-par("odd-even") folosind $P_{j-1}(\lceil n/2 \rceil)$.
- $P_0(n)$ se construieşte pe baza combinării lui $P_1(\lceil n/2 \rceil)$ cu $P_0(\lceil n/2 \rceil)$ folosind construcţia sus-jos ("upper-lower").

PROCEDURE PREFIX_PARALEL_LF(S, a, b)

if $b - a < 4$ **then**
  PREFIX_PARALEL_UL(S, a, b);
else
  do in parallel
    PREFIX_PARALEL_OE$(S, a, \lfloor (b-a)/2 \rfloor, 1)$,
    PREFIX_PARALEL_UL$(S, \lfloor (b-a)/2 \rfloor + 1, b)$
  end do
  for $i = \lfloor (b-a)/2 \rfloor + 1, b$ **in parallel do**
    $S[i] \leftarrow S[\lfloor (b-a)/2 \rfloor] \oplus S[i]$;
  end for
end if

Vom examina aici doar construcţia fundamentală care foloseşte $P_0(n)$ şi $P_1(n)$. În esenţă, construcţia impar-par se foloseşte pentru a construi $P_1(n)$ din $P_0(\lceil n/2 \rceil)$, iar construcţia sus-jos este folosită pentru a defini $P_0(n)$ în funcţie de $P_1(\lceil n/2 \rceil)$ şi $P_0(\lceil n/2 \rceil)$.

Pentru o secvenţă de 8 elemente construcţia este echivalentă cu cea din cazul sus-jos deoarece pentru 4 elemente construcţiile sus-jos şi impar-par sunt echivalente. Pentru o secvenţă de 16 elemente reţeaua de calcul este dată în Figura 4.19.

Adâncimea reţelei de calcul rămâne egală cu $\lceil \log_2 n \rceil$ deoarece pasul de combinare al construcţiei sus-jos poate începe înainte de finalizarea calculului prefixului obţinut prin construcţia impar-par. Acest lucru este posibil datorită faptului că în cazul construcţiei impar-par, ultimul element este calculat cu un pas înainte de finalizarea calculelor.

Algoritmul PREFIX_PARALEL_LF prezentat este dat doar pentru a sugera mai clar soluţia; pentru a se obţine complexitatea logaritmică însă, el trebuie transformat astfel încât ultima intructţiune de ciclare **for** să înceapă execuţia cu un pas înainte ca subalgoritmul PREFIX_PARALEL_OE să îşi finalizeze execuţia. Aceasta ar conduce la înlocuirea apelului subalgoritmilor cu codul lor şi execuţia în paralel corespunzătoare.

Complexitatea-timp este la fel de bună ca şi în cazul primei variante şi este de $\lceil \log_2 n \rceil$, dar are avantajul obţinerii unei eficienţe foarte bune şi în cazul folosirii a mai puţine procesoare. (Numărul total de operaţii este $S_0(2^k) = 4(2^k) - F(2-k) - 2F(3+k)$, $S_1(2^k) = 3(2^k) - F(1+k) - 2F(2+k)$, unde $F(k)$ este al $k-$ lea element al şirului Fibonacci.)

Numele original al acestei operaţii a fost *scan*. Ladner şi Fisher au introdus termenul de operaţie *prefix paralel*. Operaţia prefix a fost introdusă de către Iverson ca operaţie

pentru limbajul APL [91]. O implementare a acestei operaţii pe o reţea amestecare perfectă a fost sugerată apoi de către Stone [160] pentru a fi folosită la evaluarea polinoamelor. Ladner şi Fisher au prezentat prima dată un circuit eficient pentru implementarea operaţiilor prefix [103]. Wyllie a arătat cum poate fi executată această operaţie pe o listă înlănţuită folosind modelul P-RAM [175]. Prefixul paralel a fost studiat şi performanţele implementării lui îmbunătăţite şi de Schwartz [153], Mago [114], Fich [58], Lubachevsky şi Greenberg [113].

4.11 Branch-and-Bound

Există multe probleme care sunt rezolvate prin căutari într-un spaţiu foarte mare de soluţii posibile. Exemple de acest tip sunt: problema comis voiajorului, a celor n regine, a proiectarii VLSI, proces care presupune atingerea unui optim în plasarea circuitelor şi traseelor electrice.

Metoda "Branch-and-Bound" construieşte soluţia pas cu pas, evaluând de fiecare dată soluţia parţială obţinută. Se parcurge un arbore numit arbore de căutare *"search-tree"*. Rădăcina corespunde soluţiei iniţiale, iar fiecare nod neterminal reprezintă o descompunere.

Un algoritm de tip Branch-and-Bound este caracterizat de:

- cum se construieşte marginea inferioară pentru valoarea optimă;
- cum sunt generate sub-problemele – regula de ramnificare;
- cum sunt alese sub-problemele pentru continuare – regula de selecţie;
- cum sunt eliminate cazurile fără speranţă – regula de eliminare;
- cum se termină algoritmul.

O posibilă descriere a metodei Branch-and-Bound este următoarea:

Considerăm problema de optimizare P caracterizată de o mulţime de soluţii posibile $S(P)$ şi de o funcţie $v : S(P) \to R$. Mulţimea soluţie a problemei $\overline{S(P)}$ conţine toate elementele din $S(P)$ pentru care valoarea funcţiei v este optimă. Considerăm $L(P)$ tehnica de construcţie a marginii inferioare. Notăm cu UB un element din $S(P)$ care reprezintă optimul la un moment dat. O *subproblemă activă* este o subproblemă generată care nu a fost eliminată. La fiecare etapă a calculului există o *mulţime activă* care conţine toate subproblemele active şi care este gestionată ca şi o coada Q. Următorii paşi descriu fazele importante ale algoritmului:

1. `Iniţializare.` P este trecută în coada Q. Se setează $UB = \emptyset$, iar $v(\emptyset) = \infty$.
2. `Terminare.` Dacă Q este vidă atunci UB este soluţia căutată.
3. `Selecţie.` Se alege o problemă $\overline{P}$ din Q.
4. `Mărginire inferioară.` Dacă $L(\overline{P})$ nu este mai mică decât $v(UB)$ atunci se renunţă la $\overline{P}$ şi se revine la pasul (2).

5. `Ramnificare.` Se crează noi subprobleme $\overline{P}_1, \overline{P}_2, \ldots, \overline{P}_k$ din problema $\overline{P}$ astfel încât $S(\overline{P}_1) \cup S(\overline{P}_2) \cup \ldots \cup S(\overline{P}_k)$ conţine $S(\overline{P})$.

Paralelizarea metodei Branch-and-Bound se poate face în două moduri diferite:

- la nivelul calculului unei soluţii: procesoarele cooperează pentru dezvoltarea unui singur nod al arborelui de căutare;
- la nivel algoritmic înalt, când sunt explorate simultan mai multe soluţii posibile; în acest caz comunicarea între procese este mai scăzută datorită independenţei subproblemelor.

Mai concret operaţiile care pot fi paralelizate sunt:

- Calcularea marginii inferioare: paralelismul poate fi introdus şi la acest nivel dacă calcularea acestei margini este laborioasă astfel încât să justice paralelismul.
- Selecţia: La anumite etape ale execuţiei, numărul de subprobleme active poate ajunge foarte mare. Prin urmare, selecţia unei probleme şi extragerea ei din mulţimea activă poate implica un volum mare de calcul.
- Eliminarea: Testarea directă $L(\overline{P}) < v(UB)$ este imediată, dar aplicarea unor teste care să verifice dacă o subproblemă poate produce o soluţie fiabilă poate fi dificilă şi laborioasă.
- Ramnificarea: Alegerea mai multor subprobleme simultan.

Alte tehnici specifice şi programării secvenţiale pot fi folosite cu succes în dezvoltarea programelor paralele. *Programarea structurată* are un rol deosebit şi în programarea paralelă, regulile ei rămânând la fel de importante şi în cazul programării paralele. Recursivitatea poate de asemenea permite derivarea simplă de programe paralele. Metoda divide&impera, analizată anterior este implementată de cele mai multe ori pe baza recursivităţii.

4.12 Adaptarea algoritmilor paraleli

Proiectarea algoritmilor paraleli se face în prima fază considerând că există un număr nelimitat de procesoare şi deci pot fi construite oricâte procese de calcul. Algoritmii trebuie adaptaţi pentru a satisface cerinţele sistemelor de calcul pe care vor urma a fi implementaţi: adaptarea în funcţie de numărul de procesoare şi în funcţie de granularitatea sistemului.

Această adaptare a programelor paralele dezvoltate plecând de la premiza unui număr nelimitat de procesoare este necesară pentru implementarea lor şi în plus poate conduce la reducerea costului, existând posibilitatea obţinerii unor algoritmi optimali din punct de vedere al costului.

Transformarea algoritmului la cerinţele unui paralelism limitat poate fi făcută fie prin păstrarea componentelor de calcul depistate şi asignarea mai multor componente unui singur procesor, fie printr-o modificare consistentă a lor.

Conform principiului lui Brent, pentru o problemă de dimensiune n micşorarea numărului de procesoare de la p la $q, q < p$ prin asignarea mai multor componente de calcul unui element de procesare conduce la următoarea modificare a timpului paralel de execuţie: $T_q(n) = T_p(n)\lceil \frac{p}{q} \rceil$. Un procesor poate să execute concurent mai multe componente de calcul prin multiprogramare. În acest caz nu are loc un paralelism veritabil, ci mai degrabă unul virtual. Această variantă se foloseşte în special în cazurile în care partiţionarea s-a făcut prin descompunere funcţională. Operaţiile/funcţionalităţile independente, descoperite, pot fi executate pe acelaşi procesor folosindu-se multiprogramarea.

Dacă se foloseşte descompunerea domeniului de date, atunci se poate ajunge la granularitatea dorită prin reproiectarea algoritmului plecând de la premiza că există doar un număr p de procesoare, care devine un parametru al algoritmului. Partiţionarea datelor se face în p blocuri (distribuţie iniţială a datelor). Este posibil ca algoritmul nou obţinut să difere destul de mult de algoritmul derivat iniţial plecând de la premiza unui paralelism nelimitat. Apar noi operaţii necesare, cum ar fi, de exemplu, cele de compunere a datelor locale.

Pentru exemplificare, prezentăm două exemple de adaptare a algoritmilor paraleli pentru suma a n numere şi pentru sortarea par-impar.

Exemplul 4.25 (Suma a n numere cu p procese) Pentru a obţine un algoritm de însumare a n numere folosind doar $p \ll n, p|n(n$ se divide exact cu $p)$ procese, putem să aplicăm divizarea secvenţei iniţiale în subsecvenţe.

ALGORITHM SUMA-P$< A[0..n-1], n = p * k, p>$

```
for i = 0 to p − 1 in parallel do
  for j = 0 to n/p − 1 do
    A[i ∗ n/p] ← A[i ∗ n/p] + A[i ∗ n/p + j];
  end for
end for
for k = 1 to ⌈log2 p⌉ do
  for i = 0, p − 1 in parallel do
    if (i %2^k = 0 ∧ i + 2^(k−1) < p) then
      A[i ∗ n/p] ← A[i ∗ n/p] + A[(i + 2^(k−1)) ∗ n/p];
    end if
  end for
end for
```

Şirul iniţial se împarte în p grupuri a câte n/p elemente. Astfel, iniţial procesul $i, 0 \leq i < p$ va aduna elementele $A[i*n/p], A[i*n/p+1], \ldots, A[i*n/p+n/p-1]$, iar suma este depusă în locaţia elementului $A[i*n/p]$. În faza a doua se realizează calculul specificat

de prima variantă a algoritmului, dar pentru p procese. În acest caz complexitatea-timp este egală cu $O(n/p + \log_2 p)$; costul este $n + p\log_2 p$, iar dacă $p\log_2 p = O(n)$ atunci avem un algoritm optim.

Exemplul 4.26 (Sortarea par-impar cu subsecvenţe) Un alt exemplu, pentru care adaptarea la un număr p de procese duce la modificarea consistentă a algoritmului iniţial, este sortarea par-impar. Algoritmul rezultat este o combinaţie între algoritmul de sortare par-impar iniţial, interclasarea a două secvenţe sortate şi algoritmul quicksort secvenţial folosit pentru sortarea subsecvenţelor locale fiecărui proces. Secvenţa iniţială de numere A este împărţită în p subsecvenţe care se sortează folosind quicksort; în locul operaţiei de comparare-interschimbare folosită de algoritmul par-impar se va folosi interclasarea a două subsecvenţe sortate urmată de divizarea rezultatului – ALGORITM PAR_IMPAR-P.

ALGORITHM PAR_IMPAR-P$< A[0..p-1], n, p>$
  for $i = 0, p-1$ **in parallel do**
    QUICKSORT (A_i);
  end for
  for $i = 0, (p-1)/2$ **do**
    $\{pas\ par\}$
    for $i = 0, p-1, 2$ **in parallel do**
      INTERCLASARE(A_i, A_{i+1}, B);
      $A_i \leftarrow \{b_0, b_1, \ldots, b_{n/p-1}\}$;
      $A_{i+1} \leftarrow \{b_{n/p}, b_{n/p+1}, \ldots, b_{2n/p-1}\}$;
    end for
    $\{pas\ impar\}$
    for $i = 1, p-1, 2$ **in parallel do**
      INTERCLASARE (A_i, A_{i+1}, B);
      $A_i \leftarrow \{b_0, b_1, \ldots, b_{n/p-1}\}$;
      $A_{i+1} \leftarrow \{b_{n/p}, b_{n/p+1}, \ldots, b_{2n/p-1}\}$;
    end for
  end for

Algoritmul QUICKSORT poate fi înlocuit de orice alt algoritm secvenţial de sortare. Sortarea folosind quicksort este de preferat datorită complexităţii reduse.

Complexitatea-timp în acest caz este $T_p.n = O(n/p\log_2(n/p) + p * (n/p))$; sortarea subsecvenţelor locale folosind quicksort are complexitatea $O(n/p\log_2(n/p)$, iar interclasarea a două subsecvenţe de lungime n/p are complexitatea $O(n/p)$.

Prin această parametrizare a algoritmilor în funcţie de numărul de procese, se poate ajunge la algoritmi optimi din punct de vedere al costului. Costul algoritmului de sortare par-impar cu subsecvenţe este $C_p.n = p * T_p.n = O(n\log_2(n/p) + np)$. Prin urmare, dacă $p \leq \log_2 n$ atunci algoritmul este optim din punct de vedere al costului.

Adaptarea algoritmilor se poate face şi în funcţie de tipul de reţea de interconectare, care se doreşte a fi folosit sau care se dovedeşte a fi cel mai potrivit pentru algoritmul respectiv.

Exemplul 4.27 (Quicksort pe hipercub) Considerăm sortarea rapidă a n elemenente pe un hipercub cu p noduri ($p \ll n$). Fiecare procesor deţine $m = n/p$ elemente din secvenţa iniţială A. Cele p procesoare vor coopera la partiţionarea secvenţei A în două, apoi câte $p/2$ procesoare vor lucra pe fiecare din secvenţele rezultate ş.a.m.d. până când fiecare procesor va trebui să sorteze secvenţial secvenţa pe care o deţine. Datorită faptului că numărul de procesoare se împarte mereu la 2 este necesară alegerea pivotului astfel încât subsecvenţele rezultate să fie de dimensiuni cât mai apropriate. Este evident că cea mai bună alegere este chiar mediana secvenţei A; din păcate calculul exact al medianei e destul de costisitor şi prin urmare va trebui calculată o aproximantă a ei.

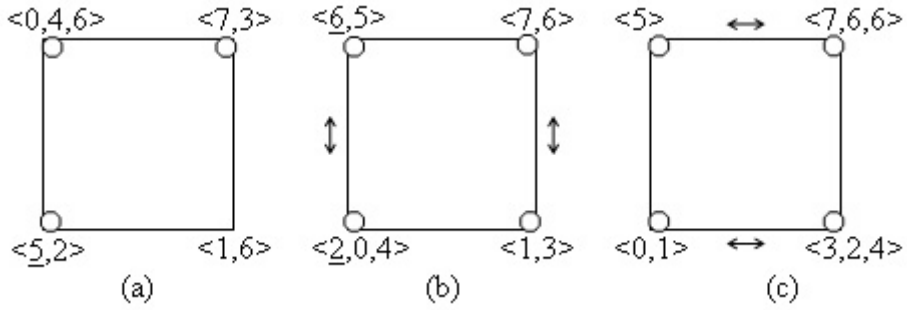

Figura 4.20: Sortarea rapidă pe un hipercub cu 4 procesoare: (a) situaţia iniţială; (b) situaţia după comunicaţia pe direcţia 1; (c) situaţia după comunicaţia pe direcţia 0.

Pe un hipercub cu $p = 2^d$, acest mod de operare se poate pune uşor în practică. Presupunem că procesorul P_0 alege pivotul v dintre elementele sale locale şi îl transmite tuturor. Fiecare procesor partiţionează secvenţa proprie: $\overline{A} =< \overline{A_0}, \overline{A_1} >$ astfel încât $\overline{A_0} < v \leq \overline{A_1}$. Apoi toate procesoarele vecine pe direcţia $d-1$ (care au diferit doar primul bit al reprezentării binare a adresei; pentru P_k acest bit este notat k_{d-1}) schimbă între ele câte o secvenţă, astfel încât sevenţele mai "mici" $\overline{A_0}$ să ajungă în jumătatea inferioară a hipercubului (unde $k_{d-1} = 0$), iar cele mai "mari" $\overline{A_1}$ în jumătatea superioară (unde $k_{d-1} = 1$). Astfel procesoarele P_i, cu $0 \leq i \leq p/2 - 1$, deţin elemente mai mici decât pivotul şi deci mai mici decât cele ale oricărui procesor P_j, cu $p/2 \leq j \leq p - 1$. În continuare, totul se repetă în cele două subhipercuburi ($H_d^{d-1}(0)$, jumătatea inferioară şi $H_d^{d-1}(1)$, jumătatea superioară), în paralel; se comunică pe direcţia $d-2$ de această dată. Se procedează similar mai departe, la fiecare iteraţie comunicându-se pe direcţia imediat inferioară, până la direcţia 0. În final sunt p grupuri formate dintr-un singur procesor, moment în care se sortează local secvenţele existente în fiecare procesor.

Figura 4.20 ilustrează algoritmul pentru cazul $p = 4$ şi $n = 8$. Elementele pivot sunt subliniate.

ALGORITHM QUICKSORT_H$< p = 2^d \ll n, id = k >$

for $l = d-1, 0, -1$ **do**
  if (<*ultimii* $l+1$ *biţi din* k *sunt egali cu* 0>) **then**
    <*alege pivotul* v *dintre elementele locale*>;
    broadcast v în $H_d^{l+1}(k)$;
  end if
  <*partiţionează* $\overline{A} =< \overline{A_0}, \overline{A_1} >, \overline{A_0} < v \leq \overline{A_1}$>;
  if $k_l = 0$ **then**
    { P_k este în subhipercubul inferior }
    in parallel
      send$(\overline{A_1}, l)$,
      recv(X, l)
    end in parallel
    $\overline{A} =< \overline{A_0}, X >$; { concatenare}
  else
    {$k_l = 1$ subhipercubul superior}
    in parallel
      send$(\overline{A_0}, l)$,
      recv(X, l)
    end in parallel
    $\overline{A} =< \overline{A_1}, X >$; { concatenare}
  end if
end for
<*sortează secvenţa proprie* $\overline{A}$>;

Complexitatea-timp poate fi aproximată în ipoteza unei încărcări echilibrate (fiecare procesor are în permanenţă aproximativ n/p elemente):

$$T(n,p) = \frac{n}{p}(\log \frac{n}{p}) + \sum_{i=0}^{d-1} \left[\frac{n}{p} + (1 + \frac{n}{2p}\alpha) \right] = \frac{n}{p}(\log n) + (1 + \frac{n}{2p}\alpha) \log p,$$

unde α reprezintă raportul dintre timpul necesar unei operaţie de comunicaţie şi timpul necesar unei operaţii de comparare.

Primul termen reprezintă complexitatea sortării locale, iar în sumă, $\frac{n}{p}$ este numărul de comparaţii necesare partiţionării, iar celălalt termen corespunde comunicaţiei; s-a neglijat timpul de difuzare(broadcast), aceasta făcându-se în paralel pe hipercuburi din ce în ce mai mici.

Algoritmul poate crea totuşi anumite probleme în practică: de exemplu, dacă P_0 alege un pivot care este cel mai mic element al său, care poate fi întâmplător mai mic şi decât oricare element in $P_{2^{d-1}}$, după prima iteraţie P_0 rămâne fără nici un element, deci nu mai poate să aleagă un pivot.

Hyperquicksort. O variantă mult mai stabilă numită hyperquicksort a fost propusă de către Wager [169]. În această variantă fiecare procesor începe prin a-şi sorta propria

---

ALGORITHM HYPERQUICKSORT$< p = 2^d \ll n, id = k >$
  $<$ *sortează secvenţa proprie* $\overline{A}>$;
  for $l = d-1, 0, -1$ **do**
    if ($<$*ultimii biţi din k sunt egali cu* $0>$) **then**
      $<$*alege mediana secvenţei locale* $\overline{A}$ *ca pivot* $v>$;
      broadcast v *în* $H_d^{l+1}(k)$;
    end if
    $<$*partiţionează* $\overline{A} =< \overline{A_0}, \overline{A_1} >, \overline{A_0} < v \leq \overline{A_1}$ *prin căutare binară*$>$;
    if $k_l = 0$ **then**
      { P_k este în subhipercubul inferior}
      in parallel
        send$(\overline{A_1}, l)$,
        recv(X, l)
      end in parallel
      $\overline{A} = \overline{A_0} \bowtie X$; {$\bowtie$ = interclasare}
    else
      { $k_l = 1$ subhipercubul superior}
      in parallel
        send$(\overline{A_0}, l)$,
        recv(X, l)
      end in parallel
      $\overline{A} = \overline{A_1} \bowtie X$; {$\bowtie$ = interclasare}
    end if
  end for

---

secvenţă. Alegerea pivotului este acum imediată: P_0 alege elementul său median – deci mediana unui eşantion de n/p elemente, care este o bună aproximare a medianei secvenţei A. Partiţionarea este şi ea mai simplă datorită faptului că avem o secvenţă sortată şi prin urmare căutarea binară poate fi folosită. În urma comunicaţiei subsecvenţelor "mici" rezultă în fiecare procesor două subsecvenţe ordonate care pot fi interclasate pentru a se obţine o secvenţă sortată. Se continuă cu iteraţii similare cu cele descrise pentru algoritmul iniţial.

Considerând din nou ipoteza unei încărcări echilibrate – care este mult mai probabilă

în acest caz– complexitatea-timp poate fi aproximată ca fiind

$$T(n,p) = \frac{n}{p}(\log\frac{n}{p}) + \sum_{i=0}^{d-1}\left[\log\frac{n}{p} + \frac{n}{p} + (1+\frac{n}{2p}\alpha)\right] \approx \frac{n}{p}(\log n) + (1+\frac{n}{2p}\alpha)\log p$$

În sumă există un termen în plus faţă de complexitatea primei variante, care este corespunzător căutării binare, egal cu $\log(n/p)$ şi care se poate neglija. Parţionarea unei secvenţe neordonată de lungime n/p are aceeaşi complexitate cu interclasarea a două secvenţe de dimensiune $n/(2p)$. Astfel, algoritmul hiperquicksort are aceeaşi complexitate în cazul cel mai favorabil ca şi algoritmul iniţial, dar o va atinge cu o mai mare probabilitate.

4.13 Şabloane de programare (Skeletons)

Şabloanele de calcul paralel au fost introduse cu scopul de a furniza o mulţime de abstractizări de nivel înalt care să furnizeze suport pentru cele mai folosite paradigme de programare paralelă. O paradigmă de programare poate fi definită ca fiind o clasă de algoritmi care rezolvă diferite probleme, dar care au aceeaşi structură de control [84]. Paradigmele de programare încapsulează în general informaţii atât despre date cât şi despre structurile de comunicaţie. Un *şablon* corespunde unei paradigme de programare specifice şi încapsulează primitivele de control şi comunicaţie ale aplicaţiei într-o singură abstractizare.

După determinarea componentelor care pot fi paralelizate şi identificarea algoritmului corespunzător, este nevoie în general de foarte mult timp pentru dezvoltarea rutinelor de programare strâns legate de paradigmă şi nu a celor specifice aplicaţiei curente. Acest timp de dezvoltare poate fi mult redus cu ajutorul unui set de rutine de interacţiune şi de şabloane.

Şabloanele ascund detaliile de implementare şi permit programatorului să specifice calculul în termenii interfeţei specifice paradigmei. Acest mod de programare – *"skeleton oriented programming" (SOP)* – a fost identificat ca fiind o soluţie foarte promiţătoare pentru impunerea programării paralele [36, 24, 45].

Şabloanele pot fi implementate pe diverse modele de calcul paralel: transmitere de mesaje, memorie partajată, orientare pe obiecte şi furnizează un suport mărit pentru programarea paralelă.

De exemplu, operaţia de reducere (`reduce` ori `fold`) prin aplicarea unui operator asociativ asupra unui număr de n operanzi (de exemplu suma a n numere) poate fi implementată eficient folosind tehnica arborelui binar. Aceasta poate constitui un şablon de calcul paralel, care poate fi implementat atât bazat pe transmitere de mesaje cât şi bazat pe memorie partajată. Având la dispoziţie aceste şabloane, programatorul nu trebuie decât să precizeze operatorul, operanzii şi eventual numărul de procese care se vor folosi.

Folosind aceste şabloane, o aplicaţie paralelă devine o secvenţă de apeluri a unor asemenea şabloane, eventual întreţesute cu calcule locale.

Deoarece funcţiile de nivel înalt (funcţionale) sunt specifice limbajelor funcţionale, multe din abordările bazate pe şabloane folosesc asemenea limbaje funcţionale. (Mai multe detalii vor fi discutate în Capitolul 7.) Totuşi, şi limbajele imperative au fost extinse prin implementarea de şabloane.

În funcţie de tipul de paralelism folosit, şabloanele pot fi clasificate în şabloane ale paralelismului funcţional şi şabloane ale paralelismului de date. În primul caz, un şablon creează în mod dinamic un sistem de procese care comunică între ele. Câteva exemple de acest tip sunt `pipe, farm, divide&impera, Branch-And-Bound`. În cel de-al doilea caz, un şablon lucrează pe o structură de date distribuită, aplicând aceeaşi operaţie pe câteva sau pe toate elementele structurii – `map, fold, scan`.

4.14 Câţiva algoritmi paraleli remarcabili

În acest capitol am descris câteva din cele mai folosite tehnici şi paradigme care se folosesc pentru proiectarea algoritmilor paraleli. Aceste tehnici sunt foarte importante, ele putând constitui o bază de la care poate pleca analiza. În partea a doua a cărţii vom prezenta modele formale care includ o metodologie de dezvoltare a softului paralel mai riguroasă, prin care se asigură o dezvoltare corectă şi eficientă.

Aceste modele se dezvoltă pentru că nu ne putem baza mereu pe intuiţie şi ingenuozitate şi de asemenea pentru că complexitatea programelor paralele poate creşte într-atât încât să nu mai poată fi stăpânită fără ajutorul unor formalisme. Totuşi, ingenuozitatea şi intuiţia pot duce la dezvoltarea unor algoritmi deosebit de performanţi. În această secţiune vom prezenta câteva exemple de asemenea algoritmi, care au fost construiţi nu bazat pe aplicarea unei tehnici anume, ci mai degrabă pe baza unor idei excepţionale.

4.14.1 Sortare bitonică

Algoritmul de sortare bitonică este un exemplu tipic de algoritm paralel care nu se inspiră din cele secvenţiale (având o complexitate secvenţială mai mare decât $O(n \log n)$). El este datorat ca idee lui Batcher [14].

Definiţii şi proprietăţi. Fie $A =< a_0, a_1, \ldots, a_{n-1} >$ secvenţa de ordonat. Pentru $i = 1, 2, \ldots, n-2$ vom spune că a_i este un *minim local* dacă a_{i-1} şi a_{i+1} sunt amândouă mai mari decât a_i şi că a_i este un *maxim local* dacă a_{i-1} şi a_{i+1} sunt amândouă mai mici decât a_i. Secvenţa A este *unimodală* dacă are cel mult un element care este minim local sau maxim local şi *bitonică* dacă se poate obţine printr-o deplasare ciclică a elementelor unei secvenţe unimodale.

Lema 4.1 (Batcher) *Fie $A =< a_0, a_1, \ldots, a_{2N-1} >$ o secvenţă bitonică de lungime pară. Se definesc secvenţele $m(A)$ şi $M(A)$ astfel:*

$$m(A) =< min(a_0, a_N), min(a_1, a_{N+1}), \ldots min(a_{N-1}, a_{2N-1}) >$$
$$M(A) =< max(a_0, a_N), max(a_1, a_{N+1}), \ldots max(a_{N-1}, a_{2N-1}) > .$$

Atunci secvenţele $m(A)$ şi $M(A)$ sunt bitonice şi orice element din $m(A)$ este mai mic decât orice element din $M(A)$.

Demonstraţia poate fi găsită în [14].

Ideea sortării bitonice. Algoritmul decurge în două etape. Mai întâi se transformă A într-o secvenţă bitonică şi apoi aceasta este sortată folosind rezultatul lemei: se formează secvenţele bitonice $m(A)$ şi $M(A)$, după care se sortează $m(A)$ şi $M(A)$, ceea ce poate fi făcut în paralel.

Pentru a rezolva prima etapă, se observă că din concatenarea unei secvenţe descrescătoare cu una crescătoare se obţine o secvenţă bitonică. Pornind de la o secvenţă de n elemente, aceasta poate fi privită ca o listă de $n/2$ secvenţe de lungime 2, din care se poate construi o listă de $n/4$ secvenţe de lungime 4, apoi o listă de $n/8$ secvenţe de lungime 8, şi aşa mai departe până când se obţine o secvenţă de lungime n. Pentru a transforma o listă de $n/2^i$ secvenţe bitonice de lungime 2^i într-o listă de $n/2^{i+1}$ secvenţe bitonice de lungime 2^{i+1} este suficient să se ordoneze secvenţele de lungime 2^i alternativ descrescător şi crescător, utilizând algoritmul de sortare al unei secvenţe bitonice.

$$a_0 \leq a_1,\ a_2 \geq a_3,\ a_4 \leq a_5,\ a_6 \geq a_7,\ \ldots$$
$$a_0 \leq a_1 \leq a_2 \leq a_3,\ a_4 \geq a_5 \geq a_6 \geq a_7,\ \ldots$$
$$\ldots$$

Figura 4.21 prezintă comparaţiile care se efectuează pentru sortarea bitonică a unei secvenţe de lungime 8, sub forma unei reţele de sortare. O săgeată indică operaţia efectuată de un procesor: comparaţie urmată de o eventuală interschimbare; elementul mai mare va fi în poziţia indicată de vârful săgeţii. Se observă că permanent gradul paralelismului este $n/2$ (au loc $n/2$ comparaţii în paralel).

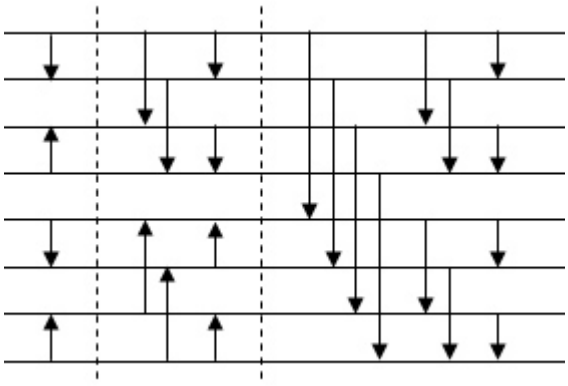

Figura 4.21: Sortarea bitonică pentru o secvenţă de $n = 8$ elemente.

Complexitatea algoritmului. Dacă $B(n)$ este numărul de comparaţii necesar pentru sortarea unei secvenţe bitonice folosind $p = n/2$ procesoare, atunci $B(2^i) = i$ (din

lema lui Batcher). Atunci timpul de sortare a unei secvenţe oarecare este:

$$T(n) = \sum_{i=1}^{\log p} B(2^i) = 1 + 2 + \ldots + \log p = O(\log^2 p)$$

Costul algoritmului este deci $O(n \log^2 n)$; cum costul secvenţial este de $O(n \log n)$, algoritmul este eficient.

Sortare bitonică pe reţea amestecare perfectă

Pe o arhitectură cu memorie distribuită, sortarea bitonică poate fi implementată eficient dacă se utilizează reţeaua de interconectare amestecare perfectă (Secţiunea 1.3.1). Această reţea are anumite proprietăţi care o fac bine adaptabilă la acest algoritm de sortare [7].

Proprietatea 1. Dacă avem $n, n = 2^k$ termeni sub aplicaţia amestecării perfecte, ei se reîntorc în poziţia iniţială după k amestecări.

Proprietatea 2. Se consideră termenii datelor din perechile de procesoare a căror indici au reprezentările binare $b_{k-1}b_{k-2}\ldots b_1b_0$ şi $b'_{k-1}b'_{k-2}\ldots b'_1b'_0$ care diferă numai în poziţia $k-r$, unde $1 \leq r \leq k$. După r amestecări, aceşti termeni se vor găsi în procesoare adiacente. (Acest lucru se observă mai uşor dacă se consideră deplasarea ciclică a reprezentării binare.)

A doua proprietate se referă la datele aşezate în perechi adiacente de procesoare: $(P_0, P_1), (P_2, P_3), \ldots$ Indicii binari, pentru fiecare pereche, diferă doar la bitul cel mai din dreapta.

Considerăm sortarea bitonică pentru $n = 8$ – Figura 4.21, cu cele 8 intrări etichetate cu $000, 001, 010, 011, 100, 101, 110, 111$. Orice pereche de etichete, corespunzătoare liniilor orizontale legate printr-o săgeată, diferă exact într-o singură poziţie binară. Fie eticheta i cu reprezentarea binară $b_{k-1}b_{k-2}\ldots b_1b_0$. Biţii pivot pentru paşii succesivi ai sortării bitonice sunt după cum urmează:

$$\begin{array}{ll} 1: & b_0 \\ 2: & b_1, b_0 \\ 3: & b_2, b_1, b_0 \\ \ldots & \\ k: & b_{k-1}, \ldots, b_1, b_0 \end{array}$$

Proprietăţile amestecării perfecte ne indică faptul că putem implementa sortarea bitonică pentru sortarea secvenţei A utilizând conexiunile din Figura 4.22 care derivă direct din reprezentarea grafică a reţelei amestecare perfectă reprezentată în Figura 1.12.

După o amestecare, fiecare comparator primeşte două numere din A, la care indicii diferă în poziţia cea mai din stânga a reprezentării lor binare; fiecare amestecare care urmează va deplasa biţii cu câte o poziţie mai la dreapta. Pentru a implementa sortarea bitonică pe interconexiunea amestecării perfecte, secvenţa A trebuie deplasată de un număr de ori înaintea fiecărui pas, pentru a asigura faptul că bitul pivot la pasul s este b_{s-1}. Acest lucru se poate realiza conform următorilor paşi:

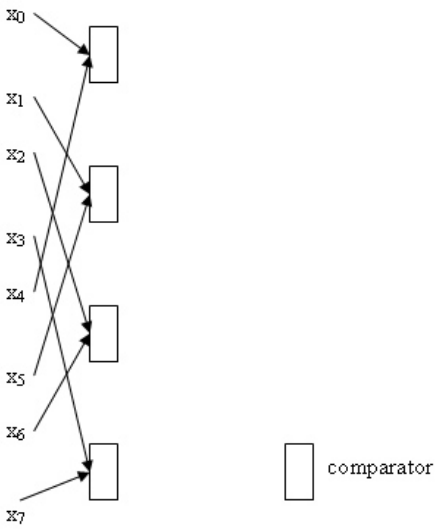

Figura 4.22: Intrarea datelor în comparatori după o amestecare.

i Pentru fiecare pivot, un pas de comparare-interschimbare este urmat de o amestecare a lui A până când se atinge b_0.

ii Dacă există s pivoţi la fiecare pas s, secvenţa trebuie amestecată de $k - s + 1$ ori înaintea pasului s, care este urmat de s paşi de comparare-interschimbare şi $s - 1$ paşi de amestecare.

În Figura 4.23 sunt prezentate etapele implicate în sortarea a $n = 8$ numere.

Sortare bitonică pe reţea hipercub

Sortarea bitonică poate fi implementată eficient şi pe o arhitectură cu reţea de interconectare hipercub. Considerăm mai întâi cazul în care există un singur element pe fiecare procesor.

Cazul $p = n = 2^d$. Din Figura 4.21 se observă că un procesor q comunică doar cu procesoarele $q + 2^i$, deci comunicaţiile au loc doar între vecini. Singura problemă constă în stabilirea sensului interschimbărilor. Algoritmul este format din d paşi, iar la pasul l un procesor q face parte dintr-o secvenţă care trebuie ordonată crescător dacă bitul $l+1$ al reprezentării binare a lui q este 0 şi descrescător dacă acesta este 1 (prin convenţie q_{d+1} este 0). Pentru un l fixat, interschimbările se fac astfel încât, pentru o pereche de procesoare P_i, P_j (cu $i = j \pm 2^k$) care face interschimbare, elementul mai mic să ajungă în procesorul cu număr mai mic, dacă se ordonează crescător şi în procesorul cu număr mai mare, dacă se ordonează descrescător.

Cazul $p \ll n$. Aplicând direct principiul lui Brent şi deci păstrând algoritmul neschimbat, modificând doar alocarea procesoarelor s-ar obţine un timp de execuţie de

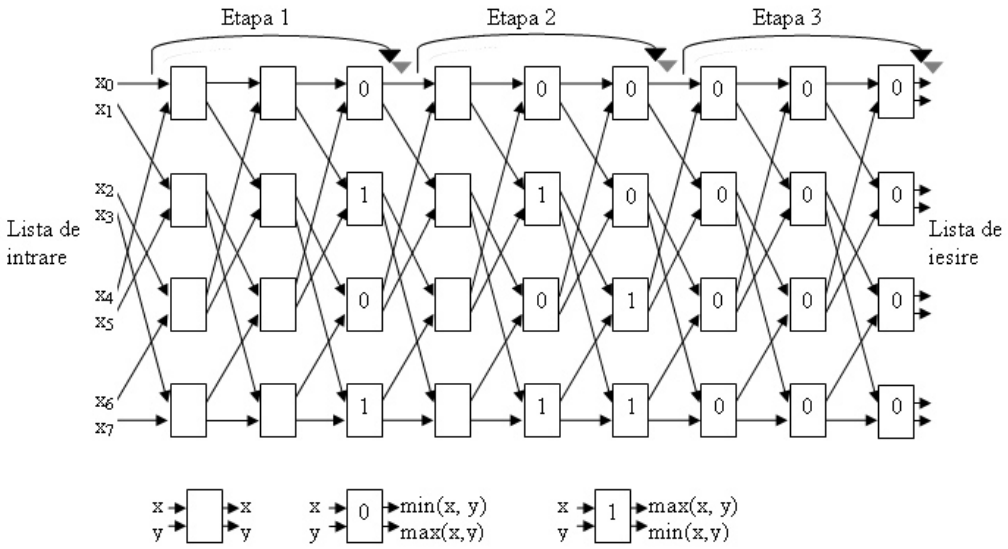

Figura 4.23: Sortarea bitonică pe interconexiunea amestecare perfectă, pentru o secvenţă de $n = 8$ elemente.

$O((n/p)\log^2 p)$, ori pentru a spera la o eficienţă cât mai bună ar trebui să ne apropiem de $O((n/p)\log p)$.

Presupunem că fiecare procesor se ocupă de $m = n/p$ elemente. La început fiecare procesor P_q sortează elementele sale într-o secvenţă A_q crescătoare dacă q e par şi descrescătoare dacă q e impar; această operaţie durează $O((n/p)\log(n/p))$. Apoi, se poate aplica algoritmul de sortare bitonică, înlocuind elementele cu secvenţe ordonate, toate de aceeaşi lungime; formarea secvenţelor de tip $m(A)$ şi $M(A)$ se face simplu, $min(A_i, A_j)$ putând fi obţinută prin alegerea fiecărui minim dintre elementele aflate pe aceeaşi poziţie în secvenţele A_i şi A_j. Operaţiile de interschimbare se vor efectua în acest caz la nivel de subsecvenţă. Se ajunge în final la o complexitate de $O((n/p)\log n + (n/2p)\log^2 p)$.

4.14.2 Înmulţire matriceală

Înmulţirea matriceală este o problemă care a fost foarte mult analizată în vederea paralelizării ei. Acest lucru este explicabil prin faptul că este o operaţie de bază în foarte multe calcule şi deci implementarea ei eficientă conduce la performanţă în multe domenii.

S-au dezvoltat numeroşi algoritmi paraleli pentru înmulţirea matricelor, folosind diferite tehnici cum ar fi: divide&impera, calcul sistolic, descompunerea domeniului de date, etc.

Vom prezenta aici algoritmul lui Cannon, algoritmul lui Fox şi paralelizarea algoritmului lui Strassen. În Capitolul 6, Secţiunea 6.3.2 se va analiza dezvoltarea formală de algoritmi paraleli bazată pe descompunerea domeniului de date pentru înmulţire matriceală, iar în Capitolul 8, Secţiunea 8.5.4 un algoritm bazat pe divide&impera şi pe replicarea datelor. În Capitolul 5, Secţiunea 5.5.3 se prezintă un algoritm foarte intere-

sant şi performant de înmulţirea paralelă a matricelor booleene.

Vom considera două matrice pătratice A şi B de dimensiune $n \times n$. Matricea produs $C = A \times B$ este tot o matrice pătratică, ale cărei elemente se obţin pe baza următoarei formule:

$$c[i,j] = \sum_{k=0}^{n-1} a[i,k] * b[k,j], \forall i,j : 0 \leq i,j < n.$$

Algoritmul lui Cannon

Algoritmul lui Cannon (1969) foloseşte o reţea de interconectare cu p^2 procesoare de tip tor bidimensional (presupunem că p divide n). Matricile A, B şi C sunt descompuse în $(n/p)^2$ submatrice, printr-o partiţionare de tip bloc – Figura 4.24.

Procesorul (s,t) de la locaţia (s,t) conţine iniţial submatricele $A[s,t]$ şi $B[s,t]$ $(0 \leq s,t < p)$. Pe măsură ce execuţia algoritmului progresează, submatricele sunt transmise spre stânga - pentru submatricele din A, şi în sus - pentru submatricele din B. Procesorul (s,t) va calcula submatricea $C[s,t]$.

Algoritmul se bazează pe formula

$$C[s,t] = \sum_{q=0}^{p-1} A[s,q] \times B[q,t], \forall s,t : 0 \leq s,t < p.$$

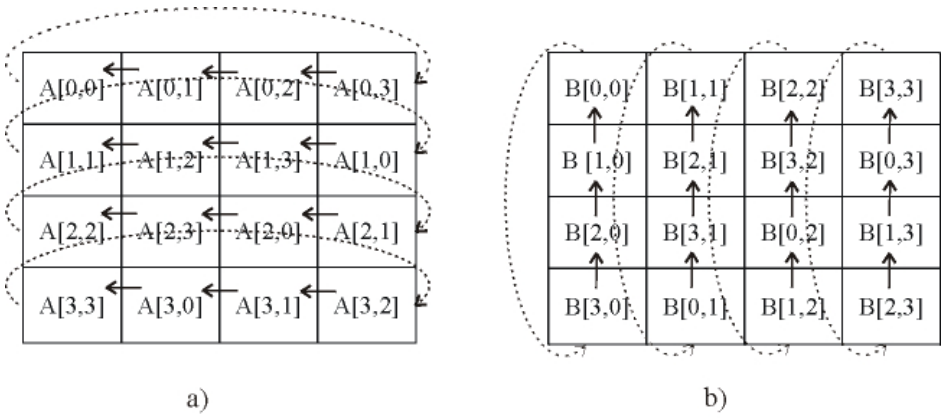

Figura 4.24: Deplasarea datelor în Algoritmul lui Canon. a) Submatricele din A se deplasează circular spre stânga. b) Submatricele din B se deplasează circular în sus.

Paşii algoritmului sunt următorii:

1. Iniţial procesorul $P(s,t)$ începe execuţia cu submatricele $A[s,t]$ şi $B[s,t]$.
2. *Deplasarea iniţială*: elementele sunt mutate din poziţia lor iniţială la o poziţie "aliniată" pentru a se putea înmulţi submatricele corespunzătoare. Aceasta aliniere se face prin deplasarea "liniei" s din A cu s poziţii la stânga, şi prin deplasarea "coloanei" t din B cu t poziţii în sus.

3. Fiecare procesor $P(s,t)$ multiplică submatricele curente şi rezultatul îl adună la submatricea C corespunzătoare.

4. *Deplasare la fiecare etapă*: submatricele sunt deplasate la stânga şi în sus cu o poziţie.

5. Pasii 3. şi 4. se repetă de $p-1$ ori.

Deplasarea iniţială se face în p etape, iar deplasarea la fiecare pas se face într-o singură etapă. La fiecare etapă, un procesor trimite submatricea sa A locală, la procesorul din stânga şi submatricea sa locală B, la procesorul de deasupra lui şi recepţionează o submatrice A de la procesorul din dreapta lui şi o submatrice B de la procesorul de sub el.

Pasul 3. de calcul are o complexitate-timp egală cu n^3/p^3.

Algoritmul lui Cannon este foarte eficient atunci când elementele matricei C rezultate vor fi folosite local şi nu este nevoie colectarea lor. Poate fi considerat un algoritm sistolic, dat fiind faptul că comunicaţiile se fac doar între vecini.

Algoritmul lui Fox

Un alt algoritm asemănător cu cel al lui Cannon este cel propus de Fox care foloseşte acelaşi tip de partiţionare a datelor. Spre deosebire de metoda lui Cannon, nu este necesară o aliniere iniţială a datelor, dar se folosesc şi comunicaţii globale de tip broadcast, nu doar comunicaţii locale între vecini.

Paşii de calcul pentru acest algoritm sunt următorii:

1. Se alege o submatrice A din fiecare linie de procese.

2. *Broadcast A*: Submatricea A aleasă este transmisă prin broadcast tuturor celorlalte procese de pe linia respectivă. (O operaţie broadcast reprezintă o operaţie de transmitere de la un proces la mai multe procese a unei informaţii.)

3. Fiecare procesor $P(s,t)$ multiplică submatricele curente şi rezultatul îl adună la submatricea C corespunzătoare.

4. *Transmisie B*: Submatricele B sunt transmise procesului aflat pe poziţia imediat superioară (ciclic).

5. Repetă de $p-1$ ori paşii anteriori.

Alegerea submatricei A de la pasul 1. se face în funcţie de iteraţia curentă: la prima iteraţie se alege submatricea $A[\cdot,0]$, la a doua iteraţie submatricea $A[\cdot,1]$ ş.a.m.d.

Faţă de algoritmul lui Cannon, cererea de memorie este mai mare, deoarece elementele submatricei A transmise prin broadcast trebuie stocate local. Totuşi, un avantaj, faţă de algoritmul lui Cannon, îl reprezintă faptul că datele matricelor iniţiale rămân neschimbate.

Paralelizarea algoritmului lui Strassen

Metoda lui Strassen este una din cele mai inovative în multiplicarea secvenţială a matricelor, reducând complexitatea clasic obţinută de $O(n^3)$ la o complexitate de $O(n^{\log_2 7})$. Algoritmul este de tip divide&impera şi foloseşte o partiţionare a matricelor în 4 submatrice, la fiecare pas - Figura 4.25.

A_{00}	A_{01}
A_{10}	A_{11}

B_{00}	B_{01}
B_{10}	B_{11}

C_{00}	C_{01}
C_{10}	C_{11}

Figura 4.25: Partiţionarea matricelor în submatrice pentru algoritmul lui Strassen.

Submatricele $C_{ij}, 0 \leq i, j \leq 1$ se obţin pe baza următoarelor formule:

$$
\begin{aligned}
P_0 &= (A_{00} + A_{11}) \times (B_{00} + B_{11}) \\
P_1 &= (A_{10} + A_{11}) \times B_{00} \\
P_2 &= A_{00} \times (B_{01} - B_{11}) \\
P_3 &= A_{11} \times (B_{10} - B_{00}) \\
P_4 &= (A_{00} + A_{01}) \times B_{11} \\
P_5 &= (A_{10} - A_{00}) \times (B_{00} + B_{01}) \\
P_6 &= (A_{01} - A_{11}) \times (B_{10} + B_{11}) \\
C_{00} &= P_0 + P_3 - P_4 + P_6 \\
C_{01} &= P_2 + P_4 \\
C_{10} &= P_1 + P_3 \\
C_{11} &= P_0 + P_3 - P_1 + P_5
\end{aligned}
$$

Fiind un algoritm de tip divide&impera paralelizarea se face foarte simplu prin calcularea în paralel a submatricelor $P_0 \ldots P_6$ şi apoi prin calcularea în paralel a submatricelor $C_{00}, C_{01}, C_{10}, C_{11}$, în funcţie de submatricele P_i calculate anterior.

4.15 Memorie partajată versus memorie distribuită

Din punct de vedere al proiectării, două sunt clasele cele mai importante de arhitecturi care trebuie luate în considerare: clasa arhitecturilor cu memorie partajată şi clasa arhitecturilor bazate pe transmitere de mesaje. De la aceste clase se ajunge la două clase specifice de programare paralelă: programare paralelă bazată pe memorie partajată şi programare paralelă bazată pe transmitere de mesaje.

4.15.1 Programare paralelă bazată pe transmitere de mesaje

Modelarea transmiterii de mesaje se poate face considerând operaţii de transmitere punct-la punct sau considerând canale de comunicaţie. Vom considera prima variantă care se bazează pe două operaţii:

- `send` - transmiterea unui mesaj către un proces precizat;
- `receive`- recepţionarea unui mesaj de la un proces precizat.

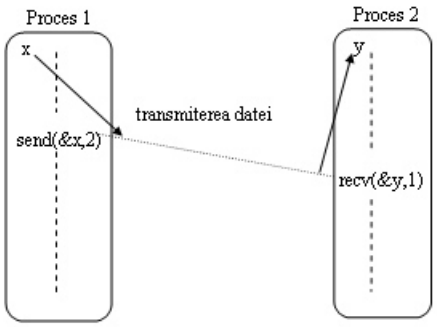

Figura 4.26: Transmiterea unei date între două procese.

Pentru programarea paralelă bazată pe transmitere de mesaje sunt necesare două mecanisme primare:

1. O metodă de creare de procese separate pentru execuţie pe diferite calculatoare.
2. O metodă de trimitere şi recepţionare de mesaje.

MPI("Message Passing interface") [179] şi PVM("Parallel Virtual Machine")[180] sunt două exemple de biblioteci de programe care permit programarea paralelă bazată pe transmitere de mesaje, pe arhitecturi cu memorie distribuită.

Creare procese

Pentru crearea proceselor putem să evidenţiem două metode: una corespunzătoare modelului SPMD("Single Program Multiple Data"), iar cealaltă corespunzătoare modelului MPMD("Multiple Program Multiple Data").

SPMD Procese diferite sunt descrise în acelaşi program, în care instrucţiuni de control selectează diferite părţi care să fie executate de fiecare proces. După compilare rezultă mai multe executabile care vor începe execuţia în acelaşi timp pe procesoare diferite - Figura 4.27. Este metoda folosită de biblioteca MPI.

MPMD În acest caz se foloseşte în general o abordare de tip "master-slave". Un proces execută procesul master, iar celelalte procese sunt pornite de către procesul master - Figura 4.28. Este metoda folosită de biblioteca PVM.

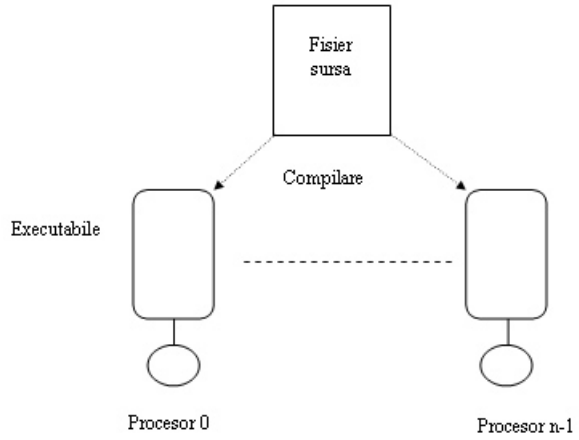

Figura 4.27: Modelul SPMD de crearea a proceselor (MPI).

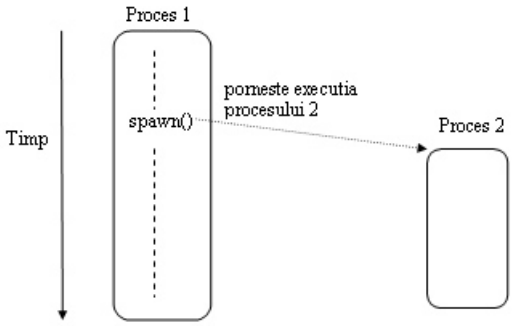

Figura 4.28: Modelul MPMD de crearea a proceselor (PVM).

Transmitere de mesaje

Transmiterea de mesaje poate fi sincronă sau asincronă. În cazul transmiterii de mesaje sincronă, rutinele de transfer (`send/recv`) se termină doar după terminarea transferului mesajului. Rutina `send` aşteaptă până când întreg mesajul este acceptat de către procesul receptor şi de abia după aceea trimite efectiv mesajul. Rutina `recv` sincronă aşteaptă până ce mesajul pe care îl aşteaptă soseşte. Rutinele sincrone de transmitere realizează şi o sincronizare a procesele pe lângă acţiunea propriu zisă de schimb de date.

Rutinele asincrone de transmitere nu aşteaptă până când acţiunea de transmitere se încheie şi de aceea este necesar în general un spaţiu local de stocare al mesajelor. Acest tip de rutine de transmitere nu realizează sincronizarea proceselor, dar pot duce la reducerea timpilor de aşteptare. Trebuie folosite cu multă grijă, astfel încât să se asigure corectitudinea programelor în orice situaţie.

Programarea paralelă bazată pe transmitere de mesaje poate folosi şi rutine de transmitere în grup, nu doar punct-la-punct. Exemple de acest tip de rutine sunt:

- `broadcast`: transmite o dată de la un proces(rădăcină) la toate celelalte;
- `scatter`: un proces rădăcină transmite câte un element al unui tablou fiecărui proces existent;
- `gather`: un proces rădăcină preia câte un element al unui tablou din fiecare proces existent.

Sincronizarea în programarea paralelă bazată pe transmitere de mesaje

Sincronizarea în programarea paralelă bazată pe transmitere de mesaje se bazează pe rutinele de bază şi poate fi locală sau globală. Sincronizarea globală presupune sincronizarea tuturor proceselor programului.

O *barieră de sincronizare* este un mecanism de bază în sincronizarea globală. Este introdusă în punctul în care fiecare proces trebuie să le aştepte pe celelalte, iar execuţia se reia doar după ce toate procesele au atins bariera. Implementarea unei bariere de sincronizare se poate realiza prin diferite tehnici. Câteva dintre acestea sunt:

1. liniară - bazată pe un contor centralizat (Figura 4.29);
2. arbore (Figura 4.30);
3. fluture (Figura 4.31);

Bariera de sincronizare poate fi implementată şi doar pentru un grup de procese.

Pentru a evidenţia problemele legate de sincronizarea locală vom considera următorul exemplu:

Procesul P_i trebuie să se sincronizeze şi să interschimbe date cu procesele P_{i-1} şi P_{i+1}, înainte de a continua calculul.

O barieră între cele trei procese nu rezolvă problema deoarece procesul P_{i-1} trebuie să se sincronizeze doar cu procesul P_i şi va continua imediat ce P_i va permite.

Este o situaţie tipică de *"deadlock"* sau *impas*, când o pereche de procese sunt implicate în transmitere şi recepţionare de date între ele. Procesele se aşteaptă reciproc şi astfel nici

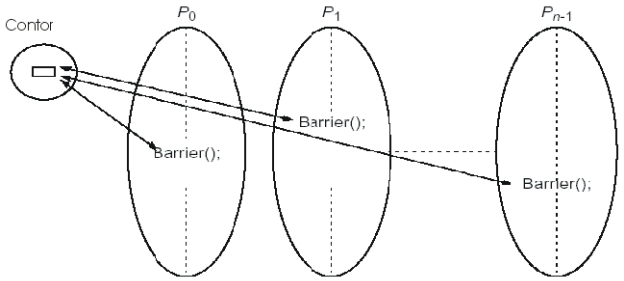

Figura 4.29: Barieră de sincronizare folosind o actualizare a unui contor aflat într-un proces considerat master. Când contorul ajunge egal cu n, bariera de sincronizare a fost atinsă de toate procesele.

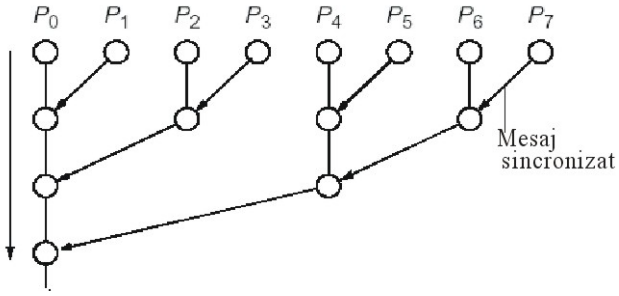

Figura 4.30: Barieră de sincronizare folosind o comunicaţie cu structură de tip arbore binar.

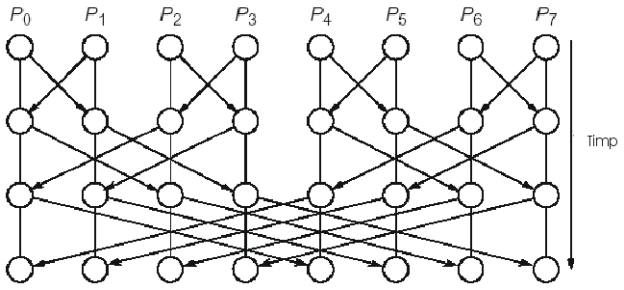

Figura 4.31: Barieră de sincronizare folosind o comunicaţie cu structură de tip fluture.

unul dintre ele nu se poate termina. Această situaţie poate apare dacă ambele procese execută o rutină `send` sincronă - cele două rutine nu se vor termina niciodată deoarece ele aşteaptă ca rutina `recv` corespunzătoare să fie atinsă.

O soluţie poate fi a aranja ca un proces să recepţioneze mai întâi şi doar apoi să transmită, iar celălalt să trimită mai întâi şi apoi să recepţioneze.

Sub paradigma transmiterii de mesaje, comunicaţia este gestionată direct de către programator, care are cunoştinţă deplină asupra şablonului de folosire a datelor. Este mai uşor să se optimizeze programele bazate pe transmitere de mesaje, deoarece este uşor de văzut unde apar comunicaţiile în program. În programarea bazată pe memoria partajată, întârzierea datorată accesului la memorie poate fi lungă şi variabilă, pe când în cazul transmiterii de mesaje citirile şi scrierile referă doar memoria privată, iar transmisiile cu pachete mari de date pot masca întârzierea ("overhead") lungă şi variabilă a reţelei de interconectare. Programarea globală a comunicaţiilor poate ajuta la evitarea coliziunilor între mesaje. Provocarea pentru programatori în cazul paradigmei transmiterii de mesaje este distribuţia cât mai avantajoasă a datelor pe procese.

4.15.2 Programare paralelă bazată pe memorie partajată

Un sistem (multiprocesor) bazat pe memorie partajată se consideră a fi caracterizat de următoarele proprietăţi:

- Orice locaţie de memorie poate fi accesibilă de către orice procesor.
- Există un singur *spaţiu de adrese*, aceasta însemnând că fiecare locaţie de memorie are o adresă unică într-un singur rang de adrese.

În general se consideră că programarea bazată pe memorie partajată este mai convenabilă decât cea bazată pe transmitere de mesaje, deşi ea necesită ca accesul la datele partajate să fie bine controlat de către programator pentru a se asigura consistenţa.

Există mai multe alternative de programare paralelă a multiprocesoarelor cu memorie partajată. Câteva dintre acestea se bazează pe:

- *Fire de execuţie* (PTreads, Java). Programatorul descompune programul în secvenţe paralele individuale, fiecare fiind asociată unui fir de execuţie care poate accesa variabile declarate în afara firelor.
- Un limbaj de programare secvenţială cu *directive compilator de preprocesare* prin care se pot declara variabile partajate şi prin care se poate specifica paralelismul (OpenMP).
- Un limbaj de programare secvenţială cu *biblioteci la nivel de utilizator* pentru a declara şi accesa variabilele partajate.
- Un *limbaj de programare paralelă* cu sintaxă pentru paralelism, în care compilatorul creează codul executabil corespunzător pentru fiecare procesor (nu foarte comun azi).

- Un limbaj de programare secvenţială şi un *compilator de paralelizare* care converteşte la cerere codul secvenţial în cod executabil paralel (nu foarte comun azi).

Secţiuni critice

Pentru a asigura corectitudinea programelor paralele bazate pe memorie partajată este necesară existenţa unui mecanism prin care să se asigure că doar un singur proces accesează o anumită resursă la un anumit moment de timp. Aceasta se realizează prin stabilirea unor aşa numite *secţiuni critice*, care sunt secţiuni de cod care folosesc o anumită resursă şi pentru care este asigurat faptul că doar o asemenea secţiune critică va fi executată la un anumit moment de timp. Acest mecanism este cunoscut sub numele de *excludere mutuală*. Este un concept foarte întâlnit în studiul sistemelor de operare.

Implementarea excluderii mutuale se realizează prin folosirea de blocaje("lock"), semafoare, monitoare, etc..

Situaţia de *deadlock* poate apare şi în acest caz între două procese, dacă unul cere o resursă deţinută de către celalalt, iar acel proces cere o resursă deţinută de către primul.

Consistenţa secvenţială [104] este definită de către L.Lamport astfel: Un multiprocesor este consistent secvenţial dacă rezultatul oricărei execuţii este acelaşi cu rezultatul obţinut în cazul în care toate procesele sunt executate într-o anumită ordine secvenţială şi operaţiile fiecărui procesor individual apar în această secvenţă în ordinea specificată de programul său.

Altfel spus, efectul global al unui program paralel nu este influenţat de nici o întreţesere arbitrară a execuţiei instrucţiunilor în timp.

Sincronizarea în cazul programării paralele bazată pe memorie partajată se poate asigura prin folosirea secţiunilor critice, sau prin testarea unei anumite stări a unei locaţii din memoria partajată. Spre exemplu, implementarea unei bariere de sincronizare poate fi realizată prin incrementarea unui contor global, fiecare proces incrementând contorul în cadrul unei secţiuni critice (contorul reprezintă în acest caz resursa comună).

Memoria partajată conduce la întârzieri mai mici decât la transmiterea de mesaje pentru maşinile SPM ("Symetric Processors Machines"). Este, de asemenea, mai uşor să se scrie o versiune funcţională a unui program în paradigma memorie partajată. Dependenţa la nivel de timp şi expertiza necesară este mai scăzută pentru paradigma memoriei partajate decât pentru transmitere de mesaje. Încărcarea de calcul este mai directă în paradigma memoriei partajate, iar tratarea aleatoare a cererilor poate reduce probabilitatea coliziunilor. Sincronizarea reprezintă marea provocare într-un sistem cu memorie partajată.

4.15.3 Memorie partajată distribuită

O abordare nouă specifică sistemelor de tip cluster permite ca memoria distribuită a unui cluster să "pară" (să poate fi folosită ca şi când ar fi) o singură memorie cu acelaşi spaţiu de adrese. În acest fel, se pot folosi tehnici ale programării bazată pe memorie partajată.

Mecanismele de transmitere de date sunt necesare, dar acestea sunt ascunse utilizatorului.

Avantajele acestui mod de programare sunt:

- Conduce la sisteme scalabile.
- Ascunde mecanismele de transmitere de mesaje.
- Poate folosi extensii simple ale programării secvenţiale.
- Poate gestiona baze de date complexe şi mari fără a fi nevoie de replicare sau transmitere de date între procese.

Există totuşi şi anumite dezavantaje:

- Poate conduce la scăderea performanţei.
- Trebuie să asigure protecţie împotriva accesului simultan la datele partajate.
- Programatorul are un control foarte scăzut asupra mesajelor reale care vor fi generate.
- Asigurarea performanţei pentru problemele neregulate este dificilă.

Sumar

După cum am văzut în capitolul precedent, paradigmele programării paralele sunt strâns legate de modul de descompunere a calculului. Partiţionarea este o etapă foarte importantă şi pentru ajunge la algoritmi paraleli cât mai eficienţi putem folosi diferite tehnici consacrate de proiectare.

În acest capitol, am făcut o trecere în revistă a celor mai cunoscute tehnici de construcţie a algoritmilor paraleli: paralelizare directă, tehnica arbore binar, dublare recursivă, contracţia arborescentă, reducere ciclică par-impar, divide&impera, algoritmii generici ASCEND şi DESCEND, calcul sistolic, tehnica par-impar, prefix paralel şi Branch-and-Bound.

S-au prezentat, de asemenea, şi modalităţi de adaptare a algoritmilor paraleli pentru paralelism limitat, precum şi diferenţele care apar în cazul memoriei distribuite faţă de cazul memoriei partajate.

Exemplele date ilustrează tehnicile prezentate, dar reprezintă şi soluţii paralele pentru unele probleme foarte des întâlnite.

Partea III

Dezvoltare formală

Capitolul 5

Modelul UNITY

UNITY "Unbounded Nondeterministic Iterative Transformations" reprezintă o teorie, care este în acelaşi timp şi un model de calcul şi un sistem de demonstrare. Această teorie a fost dezvoltată de Chandy şi Misra [31] şi are ca scop introducerea unei metodologii de dezvoltare de programe, care să fie independentă de arhitectura secvenţială sau paralelă, pe care se vor implementa. Teoria ţine cont de nedeterminism, absenţa fluxului de execuţie, sincronism/asincronism, stări şi atribuiri, dezvoltarea de programe prin rafinarea succesivă a specificaţiilor, decuplarea corectitudinii de complexitate şi programelor de arhitecturi.

5.1 Prezentarea generală a teoriei

Un program UNITY constă dintr-o declaraţie de variabile, o specificaţie a valorilor lor iniţiale şi o mulţime de intrucţiuni de atribuire multiple. O execuţie a unui program începe din orice stare care satisface condiţiile iniţiale şi continuă nedefinit; în fiecare pas al execuţiei se selectează nedeterminist o instrucţiune de atribuire, care apoi e executată. Alegerea nedeterministă a instrucţiunilor este constrânsă de următoarea regulă de 'echitate': orice instrucţiune este selectată oricât de des.

O stare a unui program se numeşte *punct fix* dacă şi numai dacă execuţia oricărei atribuiri a programului, în această stare, lasă starea neschimbată (starea e caracterizată de mulţimea valorilor tuturor variabilelor). Un predicat numit FP, caracterizează punctele fixe ale unui program.

5.1.1 Separarea noţiunilor: programe şi implementări

Un program UNITY descrie *ce* trebuie să fie făcut, în sensul că specifică starea iniţială şi transformările de stare (atribuirile). Un program UNITY nu specifică exact *când* trebuie să se execute o instrucţiune de atribuire, nu specifică *unde* (pe care procesor într-un sistem multiprocesor) se execută o anumită atribuire şi nici cărui procesor îi aparţine o atribuire. Deasemenea un program UNITY nu specifică *cum* trebuie să fie executate atribuirile, sau *cum* poate o implementare să oprească execuţia unui program.

Ce trebuie făcut este specificat în program, iar *când, unde* şi *cum* sunt specificate în maparea programului pe o arhitectură. Prin această separare de noţiuni, se obţine o notaţie a programelor simplă care este adecvată pentru o mare varietate de arhitecturi.

5.2 Notaţia programelor UNITY

Structura unui program UNITY este:

program :: **Program** *nume_program*
declare *sectiune_declare*
always *sectiune_always*
initially *sectiune_initially*
assign *sectiune_assign*
end

Un *nume-program* este orice şir de caractere. Secţiunea **Program** poate fi omisă. Secţiunea **declare**, declară variabilele folosite de către program şi tipul lor. Sintaxa este similară sintaxei Pascal. Secţiunea **always** este folosită pentru a defini anumite variabile în funcţie de altele şi este opţională. Secţiunea **initially** este folosită pentru a defini valorile iniţiale ale anumitor variabile; variabilele neiniţializate au iniţial valori arbitrare. Secţiunea **assign** conţine setul de intrucţiuni de atribuire.

Execuţia programului începe într-o stare în care valorile variabilelor sunt cele specificate în secţiunea **initially**. În fiecare pas se execută o intrucţiune de atribuire, care este selectată arbitrar.

Programele nu au instrucţiuni de intrare/iesire. Se presupune că toate intrările şi ieşirile sunt realizate prin adăugare şi ştergere de elemente din secvenţe, într-o manieră 'primul intrat primul ieşit'.

5.2.1 Instrucţiunea de atribuire

Printr-o instrucţiune de atribuire multiplă se permite unui număr de variabile să fie atribuite simultan, aşa ca în următoarea instrucţiune:

$$x, y, z := 0, 1, 2.$$

Aceasta instrucţiune poate fi scrisă şi în forma:

$$x, y := 0, 1 \parallel z := 2,$$

sau

$$x := 0 \parallel y := 1 \parallel z := 2.$$

Pentru o atribuire a elementelor unui vector se foloseşte o notaţie cuantificată:

$$< \|i : 0 \leq i < n : A(i) := B(i) > .$$

Pentru atribuirile alternative se foloseşte notaţia exemplificată de următorul exemplu:

$$\begin{array}{rl} x := -1 & if\ y < 0\ \sim \\ 0 & if\ y = 0\ \sim \\ 1 & if\ y > 0 \end{array}$$

Sintaxa generală a instrucţiunii de atribuire poate fi dată folosind notaţia BNF:

$$\begin{array}{lcl} instructiune\_atribuire & \rightarrow & componenta\_atribuire \\ & & \{\| \ componenta\_atribuire\} \\ componenta\_atribuire & \rightarrow & atribuire\_enumerata \\ & & | \ atribuire\_cuantificata \\ atribuire\_enumerata & \rightarrow & lista\_variabile := lista\_expr \\ lista\_variabile & \rightarrow & variabila\{, variabila\} \\ lista\_expr & \rightarrow & lista\_expr\_simpla | lista\_expr\_conditionala \\ lista\_expr\_simpla & \rightarrow & expr\{, expr\} \\ lista\_expr\_conditionala & \rightarrow & lista\_expr\_simpla\ if\ expr\_bool \\ & & \{\sim lista\_expr\_simpla\ if\ expr\_bool\} \\ atribuire\_cuantificata & \rightarrow & < \| \ cuatificare\ instructiune\_atribuire > \\ cuantificare & \rightarrow & lista\_variabile : expr\_bool : \end{array}$$

În cazul atribuirii cuantificate, mulţimea specificată de cuantificare trebuie să fie finită. Dacă mulţimea este vidă, atunci instrucţiunea denotă instrucţiunea vidă.

5.2.2 Secţiunea assign

Sintaxa generală este:

$$\begin{array}{lcl} sectiune\_assign & \rightarrow & lista\_instructiuni \\ lista\_instructiuni & \rightarrow & instructiune\ \{[]\ instructiune\} \\ instructiune & \rightarrow & instructiune\_atribuire | \\ & & lista\_instructiuni\_cuantificata \\ lista\_instructiuni\_cuantificata & \rightarrow & < []\ cuantificare\ instructiune\_atribuire > . \end{array}$$

Simbolul [] este separator de instrucţiuni. Există o restricţie importantă legată de expresiile booleene care pot să apară în cuantificare: acestea nu trebuie să refere variabile ale căror valori se pot schimba pe parcursul execuţiei programului. Această restricţie asigură faptul că mulţimea de instrucţiuni este constantă; instrucţiunile nu se creează sau se şterg în timpul execuţiei programului.

Exemplul 5.1 Pentru a iniţializa matricea U de dimensiune $n \times n (n > 0)$ cu matricea unitate se pot folosi trei variante de program:

$$\begin{array}{l} < []\ i, j : 0 \leq i < n \wedge 0 \leq j < n : \\ \quad U(i,j) := 0\ if\ i \neq j\ \sim\ 1\ if\ i = j \\ > \end{array}$$

sau

$$< \| \; i, j : 0 \leq i < n \wedge 0 \leq j < n : U(i,j) := 0 >$$
$$[] \; < \| \; i : 0 \leq i < n : U(i,i) := 1 >$$

sau

$$< [] \; i : 0 \leq i < n : U(i,i) := 1$$
$$[] \; < \| \; j : 0 \leq j < n \wedge i \neq j : U(i,j) := 0 >$$
$$>$$

Prima soluţie are N^2 instrucţiuni, una pentru fiecare element al matricei. A doua soluţie are două instrucţiuni, iar a treia are N instrucţiuni, câte una pentru fiecare linie.

5.2.3 Secţiunea initially

Sintaxa acestei secţiuni este aceeaşi cu cea a secţiunii **assign** cu diferenţa că semnul := se înlocuieşte cu semnul =. Secţiunea **initially** defineşte valorile iniţiale ale anumitor variabile; valoarea iniţială a unei variabile este dată ca o funcţie de valori iniţiale ale altor variabile şi de constante. Ecuaţiile care definesc valorile iniţiale nu trebuie să fie circulare.

O mulţime de ecuaţii este *corectă* dacă şi numai dacă satisface următoarele condiţii:

1. o variabilă apare cel mult o dată în partea stângă a unei ecuaţii,
2. există o ordonare a ecuaţiilor, astfel încât orice variabilă dintr-o cuantificare este fie o variabilă legată de cuantificare, fie o variabilă care apare înainte în partea stângă a unei ecuaţii (programul poate fi 'compilat'), şi
3. există o ordonare a tuturor ecuaţiilor după expandarea ecuaţiile cuantificate, astfel încât orice variabilă care apare în partea dreaptă a unei ecuaţii ori ca indice, apare înainte în partea stângă a unei ecuaţii (valorile iniţiale sunt bine definite).

Secţiunea **initially** este o mulţime *corectă* de ecuaţii.

5.2.4 Exemple

În această secţiune prezentăm două exemple scurte pentru a evidenţia noţiunile introduse până acum.

Sortare

Programul UNITY *sort*1 sortează tabloul A de $n(n > 0)$ întregi în ordine ascendentă.

Program *sort*1
assign
$< [] \; i : 0 \leq i < n : A(i), A(i+1) := A(i+1), A(i) \; if \; A(i) > A(i+1) >$
end{*sort*1}

Instrucţiunile programului se execută până când se ajunge la un punct fix, adică se ajunge la o stare care nu mai poate fi modificată de execuţia unei instrucţiuni (tabloul este ordonat).

O euristică folositoare este de a combina atribuirile care folosesc variabile diferite într-o singură instrucţiune. O altă variantă pentru sortare, se obţine separând variabilele pare de cele impare (reprezintă varianta UNITY pentru sortarea par-impar).

Program *sort*2
assign
$< \| \ i : 0 \leq i < n \wedge par(i) : A(i), A(i+1) := A(i+1), A(i) \ if \ A(i) > A(i+1) >$
$[]$ $< \| \ i : 0 \leq i < n \wedge impar(i) : A(i), A(i+1) := A(i+1), A(i) \ if \ A(i) > A(i+1) >$
end$\{sort2\}$

Această variantă poate fi rescrisă astfel:

Program *sort*3
assign
$< [] \ j : 0 \leq j < 2 :$
$< \| \ i : 0 \leq i < n \wedge j = i\%2 : A(i), A(i+1) := A(i+1), A(i) \ if \ A(i) > A(i+1) >$
$>$
end$\{sort3\}$

Coeficienţi binomiali

Următorul program calculează coeficienţii binomiali C_n^k, într-un tablou de elemente $C(n,k)$, $\forall n : 0 \leq n < N \wedge \forall k : 0 \leq k \leq n$, pentru un N dat. Se folosesc relaţiile: $C_n^0 = C_n^n = 1$ şi $C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$. Presupunem că C şi N sunt declaraţi în afara programului.

Program *binomial*
assign
$< [] \ n : 0 \leq n < N :$
$C(n,0) := 1 \ \| \ C(n,n) := 1$
$[] < \| k : 0 < k < n : C(n,k) := C(n-1,k-1) + C(n-1,k) >$
$>$
end$\{binomial\}$

Observaţii:

1. Ordinea de execuţie a instrucţiunilor într-un program UNITY este arbitrară. De aceea, $C(n,k)$ poate fi atribuit înainte ca $C(n-1,k-1)$ şi $C(n-1,k)$ să fie calculate. Dar până la urmă toţi $C(n,k)$ vor fi atribuiţi corect.

2. Orice || din acest program poate fi înlocuit cu [] (acest lucru nu este valabil pentru orice program).

3. Pentru $n = 0$, $C(0, 0)$ apare de două ori în stânga unei atribuiri, dar de fiecare dată valoarea atribuită este 1. Deci, se ajunge la punct fix.

5.2.5 Secţiunea always

Secţiunea **always** este folosită pentru a defini anumite variabile ca funcţii de alte variabile. Sintaxa folosită este ca şi cea a secţiunii **initially**. O variabilă care apare în partea stângă a unei ecuaţii se numeşte *transparentă*. O variabilă transparentă este o funcţie de variabile netransparente şi deci nu apare în partea stânga a nici unei iniţializări sau atribuiri, dar poate apare în partea dreaptă. Pentru a asigura că o variabilă transparentă este o funcţie bine definită de variabile netransparente se impun aceleaşi restricţii ca şi pentru secţiunea **initially**.

5.3 Reguli de demonstrare

În această secţiune se introduce o logică pentru specificarea, construcţia şi verificarea programelor UNITY. Logica se bazează pe aserţiuni de forma $\{p\}s\{q\}$ (p – precondiţie, q – postcondiţie), care semnifică faptul că în urma execuţiei intrucţiunii s în orice stare care satisface predicatul p, se ajunge la o stare care satisface predicatul q, dacă s se termină.

Sistemul de demonstrare al programelor UNITY diferă de cele tradiţionale, în primul rând prin faptul că neexistând un control al fluxului de execuţie, nu există aserţiuni asociate cu puncte în textul programului. Se asociază întregului program, proprietăţi de tipul "predicatul I e întotdeauna adevărat" sau "dacă predicatul p devine adevărat atunci predicatul q va fi până la urmă adevărat". În al doilea rând, execuţia unui program UNITY reprezintă o secvenţă infinită de execuţii de instrucţiuni. Prin urmare, logica asociată trebuie să poată lucra cu secvenţe infinite de stări ale programului. Dacă, de exemplu, $\{p\}s\{q\}$ e adevărat pentru toate instrucţiunile s ale unui program, poate fi folosită inducţia asupra numărului de execuţii ale instrucţiunii, pentru a arăta că odată ce p devine adevărat el rămâne adevărat.

5.3.1 Noţiuni de bază

Notăm cu $p, p', q, q', \ldots$ predicate oarecare şi cu $s, t, \ldots$ instrucţiuni program. În această secţiune considerăm un singur program generic, fără nume. Prin convenţie orice program are cel puţin o instrucţiune.

Demonstrarea aserţiunilor asupra instrucţiunilor de atribuire

Pentru a demonstra $\{p\}x := E\{q\}$, se substituie toate aparţiile lui x din q cu E, rezultatul se notează cu q^x_E şi apoi se demonstrează că $p \Rightarrow q^x_E$. Dacă E este o expresie condiţională

de forma

$$e_0 \; if \; b_0 \; \sim \; \ldots \; \sim \; e_n if \; b_n,$$

atunci q_E^x este

$$(b_0 \Rightarrow q_{e_0}^x) \wedge \ldots \wedge (b_n \Rightarrow q_{e_n}^x) \wedge ((\neg b_0 \wedge \ldots \wedge \neg b_n) \Rightarrow q).$$

Precondiţia unei atribuiri se obţine din postcondiţie prin înlocuirea simultană cu variabilele din partea stânga a expresiilor corespunzătoare din partea dreaptă.

Aserţiuni cuantificate

Pentru un program F dat, putem folosi aserţiunile cuantificate

$$(\forall s : s \in F : \{p\}s\{q\})$$

şi

$$(\exists s : s \in F : \{p\}s\{q\})$$

pentru a exprima faptul că $\{p\}s\{q\}$ este adevărat pentru toate instrucţiunile sau respectiv, pentru câteva instrucţiuni din F. Dacă programul conţine o lista de instrucţiuni cuantificate, atunci cuantificatorul asupra instrucţiunilor se aplică fiecărei instrucţiuni individuale din listă. Astfel

$$(\forall s : s \in < []i : b(i) : t(i) >: \{p\}s\{q\})$$

se demonstrează arătând că $\forall j : \{p \wedge b(j)\}t(j)\{q\}$. Similar

$$(\exists s : s \in < []i : b(i) : t(i) >: \{p\}s\{q\})$$

se demonstrează arătând că există j astfel încât $b(j)$ şi $\{p \wedge b(j)\}t(j)\{q\}$ sunt adevărate.

5.3.2 Un model al execuţiei programului

Asociem fiecărui program o mulţime de secvenţe de execuţie, care sunt secvenţe infinite, fiecare reprezentând o execuţie posibilă a programului. Notăm cu R o asemenea secvenţă, iar R_i reprezintă al i-lea element al secvenţei, $(i \geq 0)$. Fiecare R_i este o pereche $(R_i.stare, R_i.eticheta)$, unde $R_i.eticheta$ este numele instrucţiunii selectate pentru execuţie, iar $R_i.stare$ este starea programului, adică valorile tuturor variabilelor, în această execuţie înainte de pasul al i-lea. Astfel $R_0.stare$ este starea iniţială.

Restricţia selectării 'echitabile' conduce la următoarea condiţie: pentru orice R şi orice s, $R_i.eticheta = s$, pentru un număr infinit de indici i.

Reprezentăm prin $p[R_i]$ faptul că predicatul p este adevărat în starea $R_i.stare$.

Aserţiunea $\{p\}s\{q\}$ semnifică faptul că pentru orice R şi i avem

$$(p[R_i] \wedge R_i.eticheta = s) \Rightarrow q[R_{i+1}].$$

O consecinţă a acestui model de execuţie este că numărul de maşini pe care este partiţionată execuţia unui program UNITY este irelevant pentru dezvoltarea unei teorii de demonstrare. Considerăm că execuţiile pe maşini diferite sunt 'corect interţesute', adică efectul execuţiei simultane a două instrucţiuni este acelaşi cu acela al execuţiei într-o ordine arbitrară a instrucţiunilor. În acest fel, mulţimea execuţiilor posibile pentru un program este aceeaşi indiferent de numărul de maşini implicate.

5.3.3 Concepte fundamentale

În această secţiune se introduc trei relaţii logice fundamentale: ***unless*** (şi cazurile ei speciale *stable* şi *invariant*), ***ensures*** şi ***leads-to***. Se introduce formal şi noţiunea de *punct fix* al unui program.

Relaţia *unless*

Definiţia 5.1 *Pentru un program dat F, se defineşte relaţia p unless q astfel:*

$$p \ unless \ q \equiv (\forall s : s \in F : \{p \wedge \neg q\} s \{p \vee q\}). \tag{5.1}$$

Relaţia *unless* exprimă faptul că dacă predicatul p devine fals, atunci sigur predicatul q va fi adevărat.

Din definiţia relaţiei *unless* se deduce că

$$(p \wedge \neg q)[R_i] \Rightarrow (p \vee q)[R_{i+1}].$$

Cazuri speciale ale relaţiei *unless*

Definiţia 5.2 *Pentru un program dat*

$$stable \ p \equiv p \ unless \ false \tag{5.2}$$

$$invariant \ q \equiv (conditia \ iniţială \Rightarrow q) \wedge stable \ q. \tag{5.3}$$

Axioma substituţiei

Dacă $x \equiv y$ este un invariant al programului F, x poate fi înlocuit cu y în toate proprietăţile lui F.

Există o formă particulară a acestei axiome, care este foarte folositoare: înlocuirea lui *true* printr-un invariant I şi viceversa.

O consecinţă a acestei axiome este următoarea propoziţie:

Propoziţia 5.1

$$\frac{p \ unless \ q, invariant \ \neg q}{stable \ p} \tag{5.4}$$

Demonstraţie:

$\neg q \equiv true$	, axioma substituţiei
$p \ unless \ q$	, din premiză
$p \ unless \ false$	, din cele două de mai sus
$stable \ p$	, din definiţia pentru *stable*

Relaţia *ensures*

Relaţia *ensures* este folosită pentru a exprima progresul de bază al programului.

Definiţia 5.3 *Pentru un program dat F, se defineşte relaţia p ensures q după cum urmează:*

$$p \text{ ensures } q \equiv (p \text{ unless } q \wedge (\exists s : s \in F : \{p \wedge \neg q\} s \{q\})). \tag{5.5}$$

Adică, dacă p este adevărat la un anumit moment, p rămâne adevărat cât timp q este fals şi până la urmă q devine adevărat.

Din p *ensures* q se poate deduce, în termenii modelului de execuţie a programului, că

$$p[R_i] \Rightarrow (\exists j : j \geq i : q[R_j] \wedge (\forall k : i \leq k < j : p[R_k])).$$

Relaţia *leads-to*

Proprietăţile progresului programelor se exprimă folosind relaţia *leads-to*($\mapsto$).

Definiţia 5.4 *Un program dat are proprietatea $p \mapsto q$ dacă şi numai dacă această proprietate poate fi derivată printr-un număr finit de aplicaţii ale următoarelor reguli de inferenţă:*

- $$\frac{p \text{ ensures } q}{p \mapsto q} \tag{5.6}$$

- *(tranzitivitate)*

$$\frac{p \mapsto q, q \mapsto r}{p \mapsto r} \tag{5.7}$$

- *(disjuncţie) Pentru orice mulţime W,*

$$\frac{(\forall m : m \in W : p(m) \mapsto q)}{(\exists m : m \in W : p(m)) \mapsto q} \tag{5.8}$$

Semnificaţia lui $p \mapsto q$ pentru un program este că odată ce p devine adevărat, q este sau va deveni adevărat. Dar nu este certificat faptul că p rămâne adevărat atâta timp cât q nu este adevărat. Aceasta este diferenţa majoră între *ensures* şi *leads-to*.

În termenii modelului de execuţie al programelor, din $p \mapsto q$ se deduce că

$$p[R_i] \Rightarrow (\exists j : j \geq i : q[R_j]).$$

Punct fix

Un punct fix al unui program, dacă există, este o stare a programului în care execuţia oricărei instrucţiuni lasă starea neschimbată. Dacă un program se execută concurent cu alte programe şi partajează anumite variabile cu ele, atunci un punct fix este o stare care poate fi schimbată doar de către alte programe. Dacă programul se execută independent, ajungerea la un punct fix este echivalentă cu terminarea conform standardului programării secvenţiale. O implementare a programului poate alege să termine execuţia programului, în acest caz.

Definiţia 5.5 *Pentru un program F se defineşte predicatul FP în următorul mod:*

$$FP \equiv (\forall s : s \in F \wedge s \text{ este } X := E : X = E). \tag{5.9}$$

Adică, pentru orice atribuire din F, partea stângă şi cea dreaptă sunt egale în valoare.

Se poate observa că predicatul FP este satisfăcut doar de punctele fixe ale programului şi toate punctele fixe ale programului satisfac FP.

În termenii modelului de execuţie al programelor, pentru toţi R şi i avem

$$FP[R_i] \equiv (\forall j : j \geq i : R_i.stare = R_j.stare).$$

Orice stare care poate fi 'atinsă' în timpul execuţiei unui program satisface orice invariant I şi deci orice punct fix satisface $I \wedge FP$.

5.3.4 Un exemplu complet – împărţire întreagă

Vom construi un program care împarte numărul întreg M la numărul întreg N, $M \geq 0, N > 0$; câtul este reţinut în x, iar restul în y. Specificaţia programului constă în faptul că predicatul

$$x \cdot N + y = M \wedge 0 \leq y < N \tag{5.10}$$

va fi adevărat în orice punct fix şi că până la urmă se atinge un punct fix.

Program *impartire*
declare $x, y, z, k : integer$
initially $x, y, z, k = 0, M, N, 1$
assign

$$\begin{array}{lllll} z, k := & 2z, & 2k & if & y \geq 2z \sim \\ & N & ,1 & if & y < 2z \\ []x, y := & x + k, & y - z & if & y \geq z \end{array}$$

end{*impartire*}

Relaţii *unless*

Arătăm că y nu creşte atâta timp cât $y \geq z \geq N$. De asemenea, y nu-şi schimbă valoarea, iar z nu creşte, atâta timp cât $y < z$ şi $z > N$.

Propoziţia 5.2 *Pentru orice m, n:*

$$y \geq z \geq N \wedge y = m \ unless \ y < m \tag{5.11}$$

$$(y, z) = (m, n) \wedge y < z \wedge z > N \ unless \ y = m \wedge z < n. \tag{5.12}$$

Demonstraţie: Dăm demonstraţia pentru prima proprietate. Pentru prima instrucţiune, trebuie arătat că

$$\begin{array}{l}\{y \geq z \geq N \wedge y = m \wedge y \geq m\} \\ z, k := \begin{array}{cccc} 2z, & 2k & if & y \geq 2z \sim \\ N & ,1 & if & y < 2z \end{array} \\ \{(y \geq z \geq N \wedge y = m) \vee y < m\}.\end{array}$$

Această aserţiune poate fi dedusă din următoarele două, care sunt demonstrate prin aplicarea directă a regulii de demonstrare pentru instrucţiunea de atribuire:

$$\{y \geq 2z \wedge z \geq N \wedge y = m\} z, k := 2z, 2k \{y \geq z \geq N \wedge y = m\}$$

şi

$$\{2z > y \geq z \geq N \wedge y = m\} z, k := N, 1 \{y \geq z \geq N \wedge y = m\}.$$

Pentru a încheia demonstraţia pentru aserţiunea 5.11, demonstrăm următoarea aserţiune pentru a doua instrucţiune:

$$\{y \geq z \geq N \wedge y = m\} x, y := x + k, y - z \{y < m\},$$

care rezultă direct din aplicarea regulii de demonstrare pentru atribuire.

Aserţiunea 5.12 se demonstrează analog.

Demonstrarea invariantului

Propoziţia 5.3 *Predicatul I definit astfel:*

$$I \equiv y \geq 0 \wedge k \geq 1 \wedge z = N \cdot k \wedge x \cdot N + y = M. \tag{5.13}$$

este invariant al programului.

Demonstraţie: Mai întâi arătăm că I e stabil,

$$\begin{array}{l}\{I\} z, k := 2z, 2k \ if \ y \geq 2z \ \sim \ N, 1 \ if \ y < 2z \{I\} \\ \{I\} x, y := x + k, y - z \ if \ y \geq z \ \{I\}.\end{array}$$

Aceste relaţii se reduc la următoarele aserţiuni:

$$\{I \wedge y \geq 2z\}z, k := 2z, 2k\{I\}$$
$$\{I \wedge y < 2z\}z, k := N, 1\{I\}$$
$$\{I \wedge y \geq z\}x, y := x + k, y - z\{I\}$$

Aceste aserţiuni se demonstrează prin aplicarea directă a regulii de demonstrare a instrucţiunii de atribuire.

Pentru a termina demonstraţia mai trebuie demonstrat faptul că condiţia iniţială implică I. Condiţia iniţială a programului este $(x, y, z, k) = (0, M, N, 1) \wedge M \geq 0 \wedge N > 0$, care satisface I.

Relaţii *ensures*

Propoziţia 5.4

$$y \geq z \wedge y = m \ ensures \ y < m \tag{5.14}$$

$$(y, z) = (m, n) \wedge y < z \wedge z > N \ ensures \ y = m \wedge z < n. \tag{5.15}$$

Demonstraţie: Aceste relaţii, cu *ensures* înlocuit cu *unless*, au fost demonstrate. Pentru a finaliza demonstraţia, se observă că

$$\{y \geq z \wedge y = m\}x, y := x + k, y - z \ if \ y \geq z\{y < m\}$$

şi

$$\begin{array}{l}\{(y, z) = (m, n) \wedge y < z \wedge z > N\} \\ z, k := \begin{array}{lll} 2z, 2k & if & y \geq 2z \\ N, 1 & if & y < 2z \end{array} \sim \\ \{y = m \wedge z < n\}\end{array}$$

În a doua aserţiune, $y < z$ din precondiţie implică $y < 2z$. Astfel aserţiunea poate fi simplificată:

$$\{(y, z) = (m, n) \wedge z > N\}z, k := N, 1\{y = m \wedge z < n\}.$$

Relaţii *leads-to*

Propoziţia 5.5

$$(y, z) = (m, n) \wedge (y \geq z \vee z > N) \mapsto (y, z) \prec (m, n), \tag{5.16}$$

unde $\prec$ reprezintă relaţia de ordine lexicografică pe mulţimea perechilor de întregi.

Demonstraţie: Din faptul că *ensures* implică *leads-to* avem

$$y \geq z \wedge y = m \ \mapsto \ y < m \tag{5.17}$$

şi

$$(y, z) = (m, n) \wedge y < z \wedge z > N \ \mapsto \ y = m \wedge z < n. \tag{5.18}$$

Folosind proprietăţile disjuncţiei, se ajunge la:

$$(y,z) = (m,n) \wedge (y \geq z \vee z > N\ ) \mapsto\ (y < m \vee (y = m \wedge z < n)). \tag{5.19}$$

Deoarece $y < m \vee (y = m \wedge z < n)$ înseamnă $(y,z) \prec (m,n)$, rezultă proprietatea dorită.

Propoziţia 5.6

$$true \mapsto y < z \leq N. \tag{5.20}$$

Demonstraţie: Din invariant rezultă că y, z sunt ambele nenegative. Mulţimea perechilor de întregi nenegativi este convenabilă ("well found") pentru relaţia de ordine lexicografică $\prec$. Prin urmare se poate aplica inducţia asupra relaţiei:

$$(y,z) = (m,n) \wedge (y \geq z \vee z > N) \mapsto (y,z) \prec (m,n) \tag{5.21}$$

pentru a deduce că $(y \geq z \vee z > N)$ nu poate fi adevărată pentru totdeauna. Aşadar

$$true \mapsto y < z \leq N. \tag{5.22}$$

Punct fix

Propoziţia 5.7 *Pentru programul dat,*

$$\begin{array}{rcll} FP & \equiv & (y < 2z \vee (z = 2z \wedge k = 2k)) & \wedge \\ & & (y \geq 2z \vee (z = N \wedge k = 1)) & \wedge \\ & & y < z \vee (x = x + k \wedge y = y - z)). & \end{array} \tag{5.23}$$

Demonstraţie: conform definiţiei 5.5.

Din invariantul I şi predicatul FP rezultă

$$x \cdot N + y = M \wedge 0 \leq y < N. \tag{5.24}$$

Mai trebuie arătat că programul ajunge până la urmă la un punct fix, ceea ce înseamnă

$$true \mapsto FP. \tag{5.25}$$

Din proprietatea $true \mapsto I \wedge y < z \leq N$ demonstrată mai înainte şi din faptul că $I \wedge y < z \leq N \Rightarrow FP$, rezultă

$$true \mapsto I \wedge y < z \leq N \Rightarrow FP. \tag{5.26}$$

Pentru orice p, q, din $p \Rightarrow q$ rezultă p *ensures* q şi atunci $p \mapsto q$. Prin urmare $true \mapsto FP$.

5.4 Maparea programelor pe arhitecturi

Considerăm în continuare următoarele arhitecturi: arhitecturi von Neumann, arhitecturi multiprocesor sincrone cu memorie partajată, arhitecturi multiprocesor asincrone cu memorie partajată şi arhitecturi distribuite.

O mapare pe o maşina von Neumann, specifică ordinea de execuţie a atribuirilor şi maniera în care execuţia unui program se termină. Ordinea de execuţie este dată printr-o listă secvenţială de atribuiri, în care fiecare atribuire din program apare cel puţin o dată. Calculatorul execută repetat lista de atribuiri. Deoarece execuţia unui program UNITY nu se termină niciodată, terminarea se consideră a fi o caracteristică a implementării.

O cale de a implementa un program este de a-l opri după ce a ajuns la un punct fix.

Într-un sistem sincron cu memorie partajată, un număr fix de procesoare identice partajează o memorie comună care poate fi citită şi scrisă de către orice procesor. Există un ceas comun; la fiecare tact al ceasului, fiecare procesor execută un pas al calculului. Sincronismul inerent care apare într-o instrucţiune de atribuire multiplă, face ca maparea pe un astfel de sistem să fie foarte convenabilă.

Un multiprocesor asincron cu memorie partajată constă dintr-un număr fix de procesoare şi o memorie comună, dar nu există ceas comun. Dacă două procesoare accesează aceeaşi zonă de memorie simultan, atunci accesele la acea zonă se efectuează într-o ordine arbitrară. Un program UNITY poate fi mapat pe o asemenea arhitectură prin partiţionarea instrucţiunilor programului între procesoare. În plus, trebuie specificată o ordine de execuţie pentru fiecare procesor, care să garanteze o execuţie corectă pentru fiecare partiţie. Dacă execuţia pentru orice partiţie este corectă atunci orice întreţesere corectă a acestor execuţii determină o execuţie corectă a întregului program. Această mapare presupune că este respectată următoarea regulă: două intrucţiuni nu sunt executate concurent dacă una modifică o variabilă pe care cealaltă o foloseşte. Prin urmare efectul execuţiei multiprocesor este acelaşi cu întreţeserea corectă a execuţiilor.

Pentru mapare pe arhitecturi distribuite se procedează analog cu cazul multiprocesoarelor asincrone cu memorie partajată, specificând în plus şi mesajele care trebuie transmise.

5.5 Aplicaţii

În continuare prezentăm câteva programe UNITY pentru rezolvarea sistemelor liniare folosind metoda lui Gauss, inversarea unei matrice, folosind aplicarea succesivă de paşi Gauss-Jordan şi înmulţirea de matrice booleene.

5.5.1 Eliminare Gauss-Jordan nedeterministă

Considerăm metoda lui Gauss de rezolvare a unui sistem de ecuaţii,

$$A \times X = B,$$

unde A o matrice de dimensiune $n \times n$ şi B un vector de dimensiune n sunt date şi soluţia este reţinută în vectorul X ($n \in \mathbb{N}^*$). Presupunem determinantul matricei A nenul (sistemul este unic determinat). Metoda Gauss-Jordan reprezintă o secvenţă de n paşi de pivotare. Următorul program UNITY permite alegeri nedeterministe pentru selecţia liniilor de pivotare.

Soluţia

Fie $M(A;B)$ (matricea extinsă) matricea cu n linii şi $n+1$ coloane, unde primele n coloane sunt din A şi ultima din B. Prin eliminarea Gaussiană, $M(A;B)$ este transformată în $M(A';B')$ prin anumite operaţii astfel încât

$$A \times X = B$$

şi

$$A' \times X = B'$$

au aceeaşi soluţie X.

Scopul algoritmului este de a aplica secvenţe de astfel de operaţii, pentru a converti $M(A;B)$ în $M(I_n;X_F)$, unde I_n este matricea unitate, iar X_F vectorul soluţie.

Program *Gauss*
  declare
    $M(i,j : 0 \le i < n, 0 \le j < n+1) : array\ of\ real$
  assign
    {pivotare cu linia u dacă $M(u,u) \neq 0$}
    $< []u : 0 \le u < n :$
      $< \|v,j : 0 \le j < n \wedge 0 \le v < n \wedge v \neq u :$
        $M(v,j) := M(v,j) - M(v,u) \cdot M(u,j)/M(u,u)$
      $> if\ M(u,u) \neq 0$
      $\| < \|j : 0 \le j < n :$
        $M(u,j) := M(u,j)/M(u,u)$
        $> if\ M(u,u) \neq 0$
    $>$
    $[]$ {interschimbare a două linii dacă ambele au elementele de pe diagonala egale cu zero; rezultă cel puţin unul dintre aceste elemente diferit de zero}
    $< []u,v : 0 \le u < n \wedge 0 \le v < n \wedge u \neq v :$
      $< \|j : 0 \le j < n : M(u,j), M(v,j) := M(v,j), M(u,j)$
      $> if\ M(u,u) = 0 \wedge M(v,v) = 0 \wedge (M(u,v) \neq 0 \vee M(v,u) \neq 0)$
    $>$
end{*Gauss*}

O operaţie de pivotare a matricei în funcţie de linia u ($M(u,u) \neq 0$), transformă fiecare element $M(v,j), u \neq v$ în $M(v,j) - M(v,u) \cdot M(u,j)/M(u,u)$ şi fiecare element $M(u,j)$ al liniei u în $M(u,j)/M(u,u)$.

Programul *Gauss* conţine două tipuri de instrucţiuni:

1. Pivotare cu linia u, presupunând că $M(u,u) \neq 0$; aceasta are ca efect setarea variabilei $M(u,u)$ cu 1 şi $M(v,u)$ cu 0, pentru toţi $v, v \neq u$.
2. Interschimbare a două linii u şi v, în cazul în care $M(u,u) = 0$ şi $M(v,v) = 0$ şi cel puţin unul dintre elementele $M(u,v), M(v,u)$ e diferit de 0; aceasta are ca efect înlocuirea unui zero de pe diagonală cu un element diferit de zero pe diagonală.

Corectitudinea

Fie M^0 matricea M iniţială. Deoarece fiecare instrucţiune din program modifică M astfel încât soluţia sistemului este conservată, avem

$$invariant\ M^0, M \text{ au aceeaşi soluţie.}$$

Notăm cu A matricea de dimensiune $n \times n$ din partea stângă a matricei M şi cu B ultima coloană a lui M. În continuare, se arată că programul *Gauss* atinge un punct fix şi că la orice punct fix, matricea A este matricea unitate. Apoi, din invariant rezultă că B este vectorul soluţie căutat. Numim *coloană unitate* o coloană în care elementul de pe diagonală e 1 şi restul elementelor sunt 0. Adică, coloana u este o coloană unitate dacă

$$M(v,u) = 0\ if\ u \neq v\ \ \sim 1\ if\ u = v.$$

Pentru a arăta că se ajunge la un punct fix, demonstrăm că perechea (p,q), unde

$p =$ numărul de coloane unitate din A
$q =$ numărul de elemente nonzero de pe diagonală din A,

creşte lexicografic cu fiecare schimbare de stare.

Considerăm fiecare instrucţiune pe rând. Pivotarea cu linia u, dacă coloana u este coloana unitate, nu provoacă nici o modificare de stare. Schimbarea de stare rezultă în urma unei operaţii de pivotare cu linia u doar dacă coloana u nu este coloană unitate; efectul operaţiei este de a transforma coloana u în coloana unitate şi astfel incrementarea lui p.

Interschimbarea a două linii u şi v se face doar dacă $M(u,u) = 0 \wedge M(v,v) = 0 \wedge (M(u,v) \neq 0 \vee M(v,u) \neq 0)$. În general, numărul de coloane unitate p nu se modifică; poate creşte dacă, de exemplu, $M(u,v) = 1 \wedge M(i,v) = 0, \forall i \neq u$. Dar, cel puţin un element diagonal $M(u,u)$ sau $M(v,v)$ devine diferit de zero şi rezultă că q creşte.

Datorită faptului că atât p cât şi q sunt mărginite de n, rezultă ca programul *Gauss* ajunge la un punct fix.

Rămâne să arătăm că A este matricea unitate la orice punct fix. Demonstrăm mai întâi, în Lema 5.1, că oricare ar fi elementul diagonal $M(u,u)$, dacă este diferit de zero la un punct fix, atunci coloana u este coloana unitate. Apoi, în Lema 5.2 arătăm că dacă există elemente diagonale zero la un punct fix, atunci toate elementele acelei linii sunt 0. Aceasta contrazice presupunerea din ipoteză, că determinatul matricei A este diferit de 0. Ca urmare, toate elementele de pe diagonală sunt diferite de zero şi conform Lemei 5.1, A este matricea unitate.

Lema 5.1 *La orice punct fix al programului Gauss,*

$$M(u,u) \neq 0 \Rightarrow \textit{coloana } u \textit{ este coloana unitate.}$$

Demonstraţie: La orice punct fix, fiind dat $M(u,u) \neq 0$, rezultă că oricare ar fi j şi $v, u \neq v$,

$$M(v,j) = M(v,j) - M(v,u) \cdot M(u,j)/M(u,u)$$

şi

$$M(u,j) = M(u,j)/M(u,u).$$

Pentru $j = u$,

$$M(v,u) = M(v,u) - M(v,u) \cdot M(u,u)/M(u,u) = 0$$

şi

$$M(u,u) = M(u,u)/M(u,u) = 1.$$

Lema 5.2 *La orice punct fix al programului Gauss,*

$$M(u,u) = 0 \Rightarrow M(u,v) = 0, \forall v \neq u.$$

Demonstraţie: Considerăm două cazuri: $M(v,v) = 0$ şi $M(v,v) \neq 0$.

În primul caz, considerăm instrucţiunea de interschimbare a liniilor u, v. La punct fix, dacă $M(u,u) = 0 \wedge M(v,v) = 0$ avem

$$(M(u,v) = 0 \wedge M(v,u) = 0) \vee (\wedge j : M(u,j) = M(v,j)).$$

Considerăm cazul particular $j = v$. Atunci

$$(M(u,v) = 0 \wedge M(v,u) = 0) \vee (M(u,v) = M(v,v)).$$

Datorită faptului că $M(v,v) = 0$, rezultă că $M(u,v) = 0$.

În al doilea caz, dacă $M(v,v) \neq 0$ se poate folosi Lema 5.1 şi rezultă $M(u,v) = 0$.

Mapări posibile

Programul *Gauss* poate fi implementat într-o varietate de moduri pe diferite arhitecturi. Pentru o maşină secvenţială, poate fi mai eficient să se aleagă liniile de pivotare într-o anumită ordine. Corectitudinea acestei scheme este evidentă pentru că a fost obţinută din programul dat prin eliminarea nedeterminismului. Pentru o arhitectură asincronă cu memorie partajată sau pentru o arhitectură distribuită, programul dat admite mai multe implementări; cea mai simplă este de a asocia unei linii un proces. Pentru a permite operaţia de interschimbare putem permite schimbarea numărului liniei unui proces. Aşa, două linii pot fi interschimbate doar prin interschimbul numărului lor de linie. Pe o arhitectură paralelă sincronă cu $O(n)$ procesoare operaţia de interschimbare poate fi executată într-un timp constant, iar operaţia de pivotare în $O(n)$ paşi.

5.5.2 Inversa unei matrice şi rezolvare de sistem liniar

Următorul exemplu calculează inversa unei matrice, folosind aplicarea repetată de paşi Gauss-Jordan [132]. Aplicând de n ori câte un pas de eliminare *Gauss-Jordan* asupra matricei A de dimensiune $n \times n$ ($n \in \mathbb{N}^*$) se obţine matricea inversă A^{-1}[25]. Un pas *Gauss-Jordan* cu elementul pivot $A(u,v) \neq 0$ transformă elementele matricei A astfel:

$$A(i,j) = \begin{cases} 1/A(u,v) & ,i = u \wedge j = v \\ -A(i,j)/A(u,v) & ,i = u \wedge j \neq v \\ A(i,j)/A(u,v) & ,i \neq u \wedge j = v \\ (A(i,j) \cdot A(u,v) - A(i,v) \cdot A(u,j))/A(u,v) & ,i \neq u \wedge j \neq v \end{cases}$$

Soluţia

Presupunem că matricea A este nesingulară.

Program *inversa*
declare
$A(i,j : 0 \leq i < n, 0 \leq j < n) : array\ of\ real$
$ind1, ind2(i : 0 \leq i < n) : array\ of\ integer$
$x, y(i : 0 \leq i < n) : array\ of\ integer$
initially
$< u : 0 \leq u < n : ind1(u), ind2(u) = 0, 0 >$
assign
{pivotare cu elementul $A(u,v)$ dacă $A(u,v) \neq 0$}
$< []u, v : 0 \leq u < n \wedge 0 \leq v < n :$
$< \|i, j : 0 \leq i < n \wedge 0 \leq j < n :$

$$\begin{aligned} A(i,j) &:= 1/A(u,v) && if\ i = u \wedge j = v \ \sim \\ &:= -A(u,j)/A(u,v) && if\ i = u \wedge j \neq v \ \sim \\ &:= A(i,v)/A(u,v) && if\ i = u \wedge j \neq v \ \sim \\ &:= (A(i,j) \cdot A(u,v) - A(u,j) \cdot A(i,v))/A(u,v) && if\ i \neq u \wedge j \neq v \end{aligned}$$

$>$
$\|ind1(u), ind2(v), x(u), y(v) := 1, 1, v, u$
$if \quad A(u,v) \neq 0 \wedge ind1(u) = 0 \wedge ind2(v) = 0$
$>$
end{$inversa$}

Tablourile x şi y se folosesc pentru a marca permutările indicilor pe linii, respectiv coloane. Alegerea elementului pivot se face nedeterminist, cu condiţia să fie diferit de zero. Deoarece o operaţie de pivotare trebuie să se facă o singură dată pentru o linie u şi o coloană v, după execuţia unui pas de eliminare cu pivotul $A(u,v)$ marcăm $ind1(u) = 1$ şi $ind2(v) = 1$. Un pas de eliminare cu pivotul $A(u,v)$ se face doar dacă $ind1(u) = 0 \wedge ind2(v) = 0$.

În final, o permutare a liniilor şi a coloanelor matricei rezultate ne va da matricea inversă. Permutarile x (pentru linii) şi y (pentru coloane) se obţin în funcţie de alegerea elementelor de pivotare.

Pentru a transforma matricea rezultat în matricea inversă poate fi folosit programul următor.

Program $transformare$
declare
$A(i,j : 0 \leq i < n, 0 \leq j < n) : array\ of\ real$
$x, y(i : 0 \leq i < n) : array\ of\ integer$
assign
$< []u, v : 0 \leq u < n \wedge 0 \leq v < n :$
{interschimbare a două linii }
$< \|j : 0 \leq j < n : A(u,j), A(v,j) := A(v,j), A(u,j) >$
$\| \ x(u), x(v) := x(v), x(u)$
$if\ x(u) = v \vee x(v) = u$
$>$
$[]$
$< []u, v : 0 \leq u < n \wedge 0 \leq v < n :$
{interschimbare a două coloane }
$< \|j : 0 \leq j < n : A(j,u), A(j,v) := A(j,v), A(j,u) >$
$\| \ y(u), y(v) := y(v), y(u)$
$if\ y(u) = v \vee y(v) = u$
$>$
end$\{transformare\}$

Corectitudinea

Pentru demonstrarea corectitudinii programului $inversa$, notăm $p = (\sum u : 0 \leq u < n : ind1(u))$ şi $q = (\sum u : 0 \leq u < n : ind2(u))$; se poate demonstra uşor că $p = q$ la orice moment al execuţiei – deci reprezintă un *invariant*. Numărul $p = q$ creşte cu o unitate, după executarea oricărei instrucţiuni. Valoarea lor este mărginită de n şi ca urmare programul $inversa$ ajunge la punct fix, unde $p = q = n$.

Cu ajutorul indicatorilor se arată că se fac exact n paşi de eliminare Gauss-Jordan, cu elemente pivot de pe linii şi coloane diferite. Acest lucru implică faptul că matricea rezultat este matricea inversă a matricei iniţiale, cu liniile şi coloanele eventual amestecate.

Demonstrarea corectitudinii programului $transformare$ se bazează pe faptul că orice permutare poate fi scrisă ca un produs de inversiuni.

Mapări posibile

Pe o maşină secvenţială programul $inversa$ poate fi mapat prin alegerea primului element pivot găsit; căutarea elementului se face în funcţie de $ind1$ şi $ind2$. Timpul de execuţie

va fi $O(n^3)$.

Programul poate fi implementat pe un sistem paralel asincron cu memorie partajată, prin asocierea unui procesor unei linii, sau prin asocierea unui procesor fiecărui element al matricei – deci şi a operaţiilor asociate lor, în cazul în care sunt disponibile suficiente procesoare.

Pe o arhitectură sincronă cu n^2 procesoare, programul *inversa* se execută într-un timp $O(n)$, executându-se în paralel operaţiile unui pas de eliminare Gauss-Jordan.

Alte aplicaţii

Programul *inversa* poate fi folosit şi pentru aflarea rangului unei matrice. Rangul matricei va fi egal cu $p = q$, care reprezintă numărul paşilor Gauss-Jordan care au putut fi efectuaţi.

De asemenea sunt necesare foarte puţine modificări pentru a folosi acest program pentru rezolvarea unui sistem liniar unic determinat. Matricea A se înlocuieşte cu matricea extinsă $M = [A|B]$ şi în final rezultatul va fi conţinut în ultima coloană a matricei rezultate. Şi în acest caz se va ţine cont de inversiunea liniilor.

Aplicarea a n paşi Gauss-Jordan reprezintă, deasemenea, şi etapa a doua a algoritmului SIMPLEX.

5.5.3 Înmulţirea matricelor booleene

Vom prezenta mai multe variante de programe UNITY pentru înmulţirea matricelor booleene $A \times B$, care folosesc operaţii *ori* între elementele liniilor matricei B [31].

Soluţia generală

Fie A o matrice de dimensiune $m \times n$ şi B o matrice de dimensiune $n \times r$ cu elemente booleene ($m, n, p \in \mathbb{N}^*$). Dorim să calculăm matricea produs C, unde elementele $C(i, j), 0 \leq i < m, 0 \leq j < p$ sunt date de relaţia:

$$C(i, j) = (\vee k : 0 \leq k < n : A(i, k) \wedge B(k, j)). \tag{5.27}$$

Un program direct poate fi executat cu o complexitate timp $O(m \cdot n \cdot r)$ pe un procesor sau în $O(\log_2 n)$, folosind $O(m \cdot n \cdot r)$ procesoare sincrone. Există totuşi o variantă de înmulţire a două matrice booleene de dimensiune $m \times m$, prin care se poate ajunge la complexitatea timp $O(m^3/\log m)$ folosind un procesor.

Observaţia majoră care stă la baza acestui program este că orice linie a matricei produs C este o operaţie *ori* asupra unei submulţimi de linii din B. Fie $B[i]$ linia a i-a a lui B (similar pentru $A[i]$, $C[i]$). Notăm cu $B[i] \vee B[j]$ linia formată prin aplicarea operatorului $\vee$ elementelor corespunzătoare din $B[i]$ şi $B[j]$; aceasta înseamnă că al k-lea element din $B[i] \vee B[j]$ este $B(i, k) \vee B(j, k)$.

Prin urmare

$$C(i, j) = (\vee k : 0 \leq k < n \wedge A(i, k) : B(k, j)). \tag{5.28}$$

Fiecare $C[i]$ se obţine prin aplicarea operaţiei *ori* asupra unei submulţimi de linii din B; $B[k]$ este în submulţime dacă şi numai dacă $A(i,k)$ este adevărat.

O strategie de a calcula C, folosind ecuaţia 5.28, este de a calcula mai întâi matricea D, ale cărui linii se obţin prin aplicarea de operaţii *ori* asupra tuturor submulţimlor posibile ale lui B şi apoi a selecta $C[i]$ din D în funcţie de $A[i]$.

Deoarece B are n linii, rezultă că D are 2^n linii. Indexul liniilor din D determină liniile din B, din care sunt construite.

Pentru orice $r : 0 \leq r < 2^n$:

$$D[r] = (\vee j : 0 \leq j < n \wedge \text{bitul al } j\text{-lea din reprezentarea binară a lui } r \text{ este } 1 : B[j]). \tag{5.29}$$

Într-o reprezentare binară, biţii sunt număraţi începând de la 0 pentru bitul cel mai nesemnificativ.

Pentru orice $A[i]$ din A, notăm cu $\overline{A[i]}$ numărul care are bitul al j-lea din reprezentarea sa binară 1 dacă şi numai dacă $A(i,j)$ este adevărat – pentru a obţine reprezentarea binară pentru $\overline{A[i]}$, interpretăm fals ca fiind 0 şi adevărat ca fiind 1.

Atunci avem:

$$\begin{array}{ll} & D[\overline{A[i]}] \\ = & \{\text{din 5.29}\} \\ & (\vee j : 0 \leq j < n \wedge A(i,j) : B[j]) \\ = & \{\text{din 5.28}\} \\ & C[i]. \end{array}$$

Prima variantă este programul UNITY $P1$.

Program $P1$
declare
$D(i,j : 0 \leq i < 2^n, 0 \leq j < p) : array\ of\ boolean$
assign
$< \| r : 0 \leq r < 2^n :$
$D[r] := (\vee j : 0 \leq j < n \wedge$ bitul al j-lea din reprezentarea binară a lui r este $1 : B[j])$
$>$
$[]$
$< \| i : 0 \leq i < m : C[i] := D[\overline{A[i]}] >$
end$\{P1\}$

Program $P1'$
declare
$D(i, j : 0 \leq i < 2^n, 0 \leq j < p) : array\ of\ boolean$
initially $< \| r : 0 \leq r < 2^n : D[r] = O >$
assign
$< \| r : 0 \leq r < 2^n :$
$< \| j : 0 \leq j < n : D[r] := D[r] \vee B[j]$
if bitul al j-lea din reprezentarea binară a lui r este 1
$>$
$>$
$[]$
$< \| i : 0 \leq i < m : C[i] := D[\overline{A[i]}] >$
end$\{P1'\}$

Programul $P1$ conţine o instrucţiune de atribuire cu valoarea: $(\vee j : 0 \leq j < n \wedge$ bitul al j-lea din reprezentarea binară a lui r este 1 $: B[j])$ pentru care nu se specifică modul de calculare. Pentru a fi mai riguroşi putem transforma programul în programul $P1'$. (S-a folosit vectorul O prin care am notat vectorul cu toate elementele având valoarea $fals$)

Este posibil să se realizeze un calcul mai eficient dacă definim liniile lui D în funcţie de alte linii din D.

Pentru doi întregi x, y notăm $x \vee y$ numărul obţinut prin aplicarea operaţiei *ori* reprezentărilor binare ale lui x şi y. Avem atunci $D[x \vee y] = D[x] \vee D[y]$.

Se observă că $r = 2^j + k$, unde $0 \leq j < n$ şi $0 \leq k < 2^j$; j este cea mai semnificativă poziţie cu bit egal cu 1 şi k este numărul care corespunde următorilor biţi. Astfel $D[2^j + k] = B[j] \vee D[k]$.

Program $P2$
declare
$D(i, j : 0 \leq i < 2^n, 0 \leq j < p) : array\ of\ boolean$
initially
$D[0] = O$
assign
$< \| j, k : 0 \leq j < n \wedge 0 \leq k < 2^j : D[2^j + k] := B[j] \vee D[k] >$
$[] < \| i : 0 \leq i < m : C[i] := D[\overline{A[i]}] >$
end$\{P2\}$

Corectitudinea programului $P2$: Dacă execuţia primei instrucţiuni nu mai produce nici o modificare de stare, execuţia instrucţiunii prin care se iniţializează matricea C conduce starea programului într-un punct fix.

Considerăm predicatul

$$p_i \equiv D[i] = (\vee j : \text{bitul } j \text{ din reprezentarea binară a lui } i \text{ este } 1 : B[j])$$

Alegem $FP \equiv (\forall i : 0 \leq i < 2^n - 1 : p_i) \wedge (\forall i : 0 \leq i < m : C[i] = D[\overline{A[i]}])$. Se poate arăta uşor că acest predicat satisface condiţia de predicat al punctului fix.

Avem relaţia: $p_k \mapsto p_{2^j \vee k}, \forall k, j : 0 \leq j < n \wedge 0 \leq k < 2^j$. Iniţial este adevărat predicatul p_0. Prin inducţie completă (datorită tranzitivităţii relaţiei *leads-to*), se poate arăta că $p_0 \mapsto (\forall i : 0 \leq i < 2^n - 1 : p_i)$. De asemenea, avem $true \mapsto (\forall i : 0 \leq i < m : C[i] := D[\overline{A[i]}])$, datorită celei de-a doua instrucţiuni.

Am arătat că se ajunge la o stare în care este adevărat predicatul $(\forall i : 0 \leq i < 2^n - 1 : p_i)$. Orice execuţie a primei instrucţiuni în această stare nu va produce nici o modificare de stare. Prin urmare, am demonstrat că se ajunge la punct fix şi FP satisface specificaţia.

Numărul de operaţii *ori* este proporţional cu $(+j : 0 \leq j < n : 2^j) = O(2^n)$. Numărul de linii din C care sunt atribuite este $O(m)$. Deci numărul total de operaţii linie este $O(2^n + m)$. Dacă n este $O(\log m)$, înmulţirea matricelor booleene poate fi făcută în $O(m)$ operaţii linie sau într-un timp $O(m \cdot p)$ pe o arhitectură secvenţială.

Două matrice booleene cu dimensiunea $m \times m$ fiecare, pot fi înmulţite prin partiţionarea mai întâi a matricei din stânga în $m/\log m$ submatrice de dimensiunee $m \times \log m$ şi a matricei din dreapta în acelaşi număr de submatrice cu dimensiunea $\log m \times m$ şi apoi înmulţirea perechilor corespunzătoare, fiecare într-un timp $O(m^2)$ pe o arhitectură secvenţială. Întregul produs va fi calculat ca disjuncţie a matricelor produs corespunzătoare. Prin urmare, înmulţirea matricelor booleene are o complexitate-timp de $O(m^3/\log m)$ pe o maşină secvenţială, care reprezintă o îmbunătăţire faţă de $O(m^3)$.

Înmulţirea matricelor booleene pe arhitecturile paralele

Pe arhitectură paralelă fiecare linie din D se calculează prin $O(n)$ operaţii *ori* cu linii din B. Dacă folosim $O(n \cdot p)$ procesoare, poate fi calculată într-o complexitate-timp $O(\log n)$. Toate liniile din D pot fi calculate în paralel; prin urmare matricea D poate fi calculată într-un timp $O(\log n)$ folosind $O(2^n \cdot n \cdot p)$ procesoare. Calcularea matricei C se poate face apoi într-un timp $O(1)$, folosind $O(m \cdot p)$ procesoare.

Este posibil de a obţine un program $P3$ mai bun, folosind ideea programului P2.

Program $P3$
declare
$D(i, j : 0 \leq i < 2^n, 0 \leq j < p) : array\ of\ boolean$
initially
$D[0] = O$
$< \| j : 0 \leq j < n : D[2^j] = B[j] >$
assign
$< [] t : 0 < t < n \wedge t$ este o putere a lui 2 :
$< \| r : t < |r| \leq 2t \wedge 1 \leq r < 2^n :$
$\{r_x$ are biti 1 pe primele t poziţii unde r are 1$\}$
$\{r_y$ are biti 1 pe celelalte poziţii unde r are 1$\}$
$D[r] := D[r_x] \vee D[r_y]$
$>$
$>$
$[] < \| i : 0 \leq i < m : C[i] := D[\overline{A[i]}] >$
end$\{P3\}$

Ideea se bazează pe relaţia $D[r] = D[x] \vee D[y]$, unde $r = x \ \vee y$. Putem alege x şi y, astfel încât fiecare să aibă aproximativ jumătate din numărul de biţi egali cu 1 din reprezentarea binară a lui r. Formal, fie $|r|$ numărul de biţi egali cu 1 din reprezentarea binară a lui r. Dacă $t < |r| \leq 2t$ este posibil să găsim x, y astfel încât $r = x \ \vee y$, $|x| \leq t$ şi $|y| \leq t$. Alegem x, y astfel: x are 1 în reprezentarea sa binară exact în primele t poziţii unde r are 1, iar y are 1 pe poziţiile care rămân. Astfel, toate liniile din D ale căror indici au $2t$ sau mai puţini de 1 în reprezentarea lor binară pot fi calculaţi într-un pas paralel din liniile lui D care au t sau mai puţini 1 în reprezentarea binară a indicilor lor. Un indice de linie din D are cel mult n de 1 în reprezentarea lor binară; deci toate liniile lui D se calculează în $O(\log n)$ paşi.

Calcularea unei linii necesită aplicarea unei operaţii *ori* la două linii cu lungimea p; deci sunt necesare $O(p)$ procesoare pentru această operaţie. Prin urmare sunt necesare $O(2^n \cdot p)$ procesoare.

Corectitudinea programului $P3$: Demonstrarea corectitudinii se face în mod analog cu demonstrarea corectitudinii programului $P2$. Ne vom concentra atenţia pentru a demonstra că prima instrucţiune cuantificată conduce la formarea corectă a matricei D. Orice execuţie a instrucţiunii $< \| i : 0 \leq i < m : C[i] := D[\overline{A[i]}] >$ nu are importanţă înaintea calculării complete a matricei D. Execuţia ei după ce starea matricei D nu se mai schimbă, conduce la punct fix.

Considerăm predicatul $p_i : D[i] = (\vee j :$ *bitul* j *din reprezentarea binară a lui* i *este* $1 : B[j])$. Avem relaţia. $p_x \wedge p_y \mapsto p_{x \vee y}$. Iniţial predicatele p_{2^j} sunt adevărate pentru $0 \leq j < n$. Prin inducţie completă se poate arăta că $(\wedge : 0 \leq j < n : p_{2^j}) \mapsto (\forall i : 0 \leq i < 2^n - 1 : p_i)$. Deci matricea D este corect calculată.

Extindere – înmulţire matrice de numere

Există numeroase aplicaţii (din teoria grafurilor, teoria compilatoarelor,etc.) în care se cere să se realizeze operaţii de înmulţire între matrice ale căror elemente aparţin mulţimii $\{0, 1\}$.

Este posibil să se adapteze programul pentru înmulţirea matricelor booleene pentru a putea fi folosit la înmulţirea a două matrice A şi B de dimensiune $m \times m, (m \in \mathbb{N}^*)$, $A(i, j) \in \{0, 1\}$ [130].

Transformăm programul P3 astfel: înlocuim operatorul *ori* cu operatorul +, iar $\overline{A[i]}$ se calculează în mod analog cu modul de calculare anterior, cu diferenţa că se înlocuieşte valoarea adevărat cu 1 şi valoarea fals cu 0. Se obţine, astfel, un program pentru înmulţire de matrice cu proprietăţile specificate anterior.

Avantajele folosirii algoritmului bazat pe înmulţirea matricelor booleene sunt şi de această dată legate de complexitatea-timp obţinută. Complexitatea unei implementări pe o arhitectură sincronă cu $2^m \cdot m$ procesoare este $O(\log_2 m)$.

Extindere – înmulţire matrice rare

O extindere a algoritmului de înmulţire a matricelor se poate face şi pentru matricele rare [130]. În cazul matricelor rare ordinul m al matricelor este foarte mare, iar numărul elementelor nenule este de ordin $O(m)$.

Vom considera din nou cazul special în care elementele matricii A satisfac condiţia: $A(i, j) \in \{0, 1\}$.

O reprezentare clasică pentru matrici rare foloseşte câte o listă pentru fiecare linie a matricei [177]. În fiecare nod al listei a i-a se memorează câte o pereche (j, v) care reprezintă un număr de coloană şi valoarea $v = A(i, j) \neq 0$ (lista este ordonată crescător în funcţie de j). Numărul de noduri ale unei liste este egal cu numărul elementelor nenule de pe acea linie.

Putem asocia fiecărei linii i a unei matrice rare A, un număr $\overline{A[i]}$ a cărui reprezentare în baza 2 are următoarea proprietate: bitul al j-lea este egal cu 1 dacă şi numai dacă $A(i, j) \neq 0$ (este analog numărului $\overline{A[i]}$ definit pentru matricele booleene).

Numărul $\overline{A} = (\vee i : 0 \leq i < m : \overline{A[i]})$ va furniza informaţii despre numărul de coloane cu toate elementele 0 ale matricei A. Notăm cu m' numărul de biţi diferiţi de 0 din reprezentarea lui $\overline{A}$ ($m' = |\overline{A}|$).

Pentru a adapta programul $P3$ la înmulţirea matricelor rare, calculăm pentru fiecare $i : 0 \leq i < m$, numărul $\widetilde{A[i]}$ din numărul $\overline{A[i]}$ astfel: se elimină biţii de pe poziţiile în care numărul $\overline{A}$ are biţi egali cu 0. De exemplu, dacă $\overline{A} = 1010001001$, iar pentru un i oarecare $\overline{A[i]} = 1000001000$ atunci $\widetilde{A[i]} = 1010$.

Apoi, calculăm matricea B' ($m' \times m$) prin înlăturarea liniilor j din B, pentru care bitul al j-lea din reprezentarea lui $\overline{A}$ este nul. Aceste linii pot fi eliminate pentru că nu vor participa la calcularea elementelor matricei produs, datorită coloanelor complet nule, corespunzătoare, din matricea A.

Putem acum să definim programul $P4$ pentru înmulţirea matricelor rare $A \times B$, $A(i, j) \in \{0, 1\}$. Programul va folosi matricea B' în loc de matricea B şi numerele

$\widetilde{A[i]}$ în locul numerelor $\overline{A[i]}$.

Notăm cu *List* tipul listelor folosite pentru reprezentarea liniilor matricelor rare, iar aici O reprezintă lista vidă.

Program $P4$
  declare
    $D(i : 0 \leq i < 2^{m'}) : array\ of\ List$
  initially
    $D[0] = O$
    $< \| j : 0 \leq j < m' : D[2^j] = B'[j] >$
  assign
    $< []t : 0 \leq t < m' \wedge t$ este o putere a lui $2 :$
      $< \| r : t < |r| \leq 2t \wedge 1 \leq r < 2^n :$
      $\{r_x$ are biti 1 pe primele t poziţii unde r are $1\}$
      $\{r_y$ are biti 1 pe celelalte poziţii unde r are $1\}$
      $D[r] := D[r_x] \oplus D[r_y]$
      $>$
    $>$
    $[] < \| i : 0 \leq i < m : C[i] := D[\widetilde{A[i]}] >$
end$\{P4\}$

Suma $D[r] := D[r_x] \oplus D[r_y]$ reprezintă suma a două linii a căror reprezentare este dată sub formă de listă. Complexitatea-timp a acestei operaţii este dată de suma lungimilor celor două liste; prin urmare depinde de numărul maxim de elemente nenule de pe o linie a matricei. Dacă notăm $LA = (\max\ i : 0 \leq i < m : (+j : 0 \leq j < m \wedge A(i,j) \neq 0 : 1))$ şi analog LB, atunci complexitatea operaţiei este mărginită de $2LB$.

Complexitatea întregului program pe o arhitectură sincronă cu $O(2^{m'})$ procesoare este mărginită de $O(LB \log_2 m')$. Aceasta este mai bună decât complexitatea obţinută printr-un algoritm clasic care este $O(LA \cdot m)$, folosind $O(m)$ procesoare.

Complexitatea calculării numerelor $\overline{A}$ şi $\widetilde{A[i]}, 0 \leq i < m$ este $O(m)$ folosind $O(m)$ procesoare.

Dacă numărul m' este $O(\log_2 m)$ atunci numărul de procesoare necesare este $O(m)$. Dacă nu există suficiente procesoare disponibile, matricele pot fi partiţionate în submatrice (matricea A se partiţionează pe coloane, iar matricea B pe linii) şi apoi se calculează matricea rezultat ca fiind suma submatricelor produs rezultate.

Sumar

Modelul prezentat în acest capitol este nu doar un model de construcţie pentru programe paralele, ci este un model general de construcţie a programelor, fie ele secvenţiale sau paralele. Prin caracterul lui unitar aduce o nouă perspectivă în construcţia programelor şi pe de altă parte conduce la depăşirea inerţiei în construcţia de programe paralele.

Folosind acest model, nu este mai dificil să se construiască un program paralel, decât unul secvenţial; se construieşte de fapt o schemă program unică, care apoi este implementată pe diverse arhitecturi secvenţiale sau paralele.

Metoda permite de asemenea verificarea formală a corectitudinii programelor.

Acestă metodă foloseşte un nivel înalt de abstractizare pentru construcţia de programe şi schimbă modul uzual de abordare în construcţia programelor.

Capitolul 6

Dezvoltare formală din specificaţii a algoritmilor paraleli

În acest capitol, vom prezenta o metodă de dezvoltare formală, care se poate folosi cu succes pentru aplicaţii paralele bazate pe transmitere de mesaje. Prin urmare algoritmii derivaţi prin aceasta metoda pot fi uşor implementaţi folosind biblioteci de tip PVM ("Parallel Virtual Machine") şi MPI ("Message Passing Interface"). De asemenea, metoda poate fi adaptată şi pentru modelul BSP ("Bulk Synchronous Parallel") care se bazează pe superpaşi delimitaţi de bariere de sincronizare. Mai multe detalii despre acestea vor fi date în Capitolul 9.

Metoda prezentată este o metodă de dezvoltare corectă a programelor paralele, pornind de la specificaţii. Ea are la bază folosirea proceselor şi a specificaţiilor parametrizate [111].

Folosind această metodă este analizată şi problema distribuţiei datelor în mod formal şi este evidenţiat de asemenea impactul distribuţiilor de date în dezvoltarea programelor paralele.

Distribuţia datelor este determinantă în construcţia programelor paralele. Se pot folosi *distribuţii simple* prin care o dată este atribuită unui singur proces, sau *distribuţii multivoce*, prin care o dată de intrare se atribuie unei mulţimi de procese. Distribuţiile multivoce se folosesc în special atunci când numărul datelor de intrare este mai mic decât numărul de procese (programe paralele pentru sisteme cu granulaţie fină), dar aşa cum relevă şi ultimul exemplu prezentat în capitol, în anumite cazuri folosirea lor poate conduce la programe paralele generale mai eficiente.

6.1 Descrierea metodei

Un program paralel poate implica multe procese cu interacţiuni complexe. Este ştiut faptul că un grad înalt de complexitate pentru un program paralel îngreunează foarte mult dezvoltarea lui.

Metodelele formale şi principiile de inginerie informatică, cum este separarea obiective-

lor, sunt indispensabile în construcţia programelor paralele. Metoda pe care o prezentăm aici, se bazează pe metodele formale de construcţie a programelor secvenţiale. Esenţială în această metodă este noţiunea de *parametrizare* a proceselor. Un proces parametrizat este asemănător unei proceduri dintr-un program secvenţial; diferenţa constă în faptul că nu avem o singură instanţiere(apel), ci avem mai multe la un moment dat. Un program paralel se obţine din instanţierea a p procese, dintr-un singur proces parametrizat. Se obţin astfel procese cu structură similară. Specificarea proceselor parametrizate este foarte asemănătoare cu specificarea programelor secvenţiale. Un program paralel este specificat folosind specificaţii funcţionale ca şi în programarea secvenţială. O asemenea specificare formează punctul de start pentru dezvoltarea unui program paralel, o construcţie formală a proceselor parametrizate constituind un program paralel. Pentru a ajunge la această construcţie formală folosim precondiţii, postcondiţii şi invarianţi parametrizaţi.

Un proces parametrizat este detaliat în secvenţe de programe secvenţiale ordinare şi în procese de comunicaţie. În acest fel un program este descompus în nivele ale instanţelor proces.

Exemplu:

$$\begin{array}{cccc} & S.0 :: & S.1 :: & S.2 :: \\ L0 & S0.0 & S0.1 & S0.2 \\ L1 & ;C0.0 & ;C0.1 & ;C0.2 \\ L2 & ;S1.0 & ;S1.1 & ;S1.2 \\ L3 & ;C1.0 & ;C1.1 & ;C1.2 \end{array} \tag{6.1}$$

Acest exemplu prezintă un program paralel, constând din 3 instanţe ale procesului parametrizat S. Fiecare proces $S.q, 0 \leq q < 3$, este descompus pe verticală într-o secvenţă de 4 procese $S0.q, C0.q, S1.q, C1.q$. Procesele parametrizate $S0.q$ şi $S1.q$ sunt programe secvenţiale, iar $C0.q, C1.q$ sunt procese de comunicaţie. Un alt mod de a privi un program este prin descompunerea sa pe nivele: $L0, L1, L2, L3$. Fiecare nivel constă din acelaşi proces parametrizat. O regulă importantă pentru asigurarea dezvoltării corecte a programelor impune ca şi comunicaţia să aibă loc doar între procese de comunicaţie din acelaşi nivel.

Această specificare strictă a obiectivelor în construcţia unui program paralel are următoarele avantaje:

- Poate fi dată o specificaţie clară pentru fiecare proces parametrizat.
- Tipul interacţiunii între procese este limitat, deoarece un proces parametrizat este ori un program secvenţial ordinar, ori un proces de comunicaţie.
- Rolul unui proces de comunicaţie devine clar şi este facilitată demonstrarea corectitudinii programelor paralele. Este relativ uşor de analizat influenţa reţelei de comunicaţie asupra complexităţii programelor paralele şi putem considera implementări diferite pentru procesele de comunicaţie.
- Fiecare coloană poate fi considerată ca o unitate – o implementare pe un sistem multiprocesor poate fi făcută prin atribuirea fiecărei coloane unui procesor.

- Distribuirea volumului de calcul şi a comunicaţiilor echilibrat între nivele garantează obţinerea unui program paralel eficient.

6.1.1 Construcţia programelor paralele

6.1.2 Specificaţii funcţionale

Modalitatea de specificare a programelor paralele porneşte de la specificaţia formală Hoare pentru programe secvenţiale:

$$\{Q\}S\{R\}, \tag{6.2}$$

unde Q este precondiţia, R postcondiţia şi S programul. Semnificaţia specificaţiei Hoare este: dacă S începe într-o stare în care predicatul Q este adevărat şi S se termină, atunci după execuţie, R va fi adevărat (condiţia de parţial corectitudine). Tripletul Hoare poate fi extins pentru specificarea programelor paralele astfel: precondiţia şi postcondiţia sunt scrise ca şi conjuncţii de p ($p > 0$) pre- şi post- condiţii locale pentru fiecare proces(fir) al programului paralel:

$$\{Q.q\}S.q\{R.q\}, 0 \leq q < p. \tag{6.3}$$

Dacă S este un program paralel parametrizat, pre- şi post- condiţiile sunt parametrizate. Deci în locul a p specificaţii, poate fi dată doar una, dar parametrizată. O specificaţie parametrizată poate conţine variabile locale care reprezintă părţi ale datelor distribuite(ex. tablou, graf). Deci parametrii specificaţiei sunt q, p, şi distribuţia datelor D. În aserţiuni nu se mai specifică parametrii p şi D, pentru simplificarea notaţiei. Sunt posibile diferite distribuţii a datelor, fiecare având un alt impact asupra complexităţii programului paralel. Presupunem că nu există memorie partajată (deci variabile globale), datele fiind toate partiţionate între procese.

Exemplul 6.1 Prezentăm structura şi specificaţia unui program pentru calcularea sumei a n numere date într-un tablou t de lungime n. Programul paralel este format din p instanţe pentru S, numite $S.q, 0 \leq q < p$, care se vor executa în paralel:

$|[p, n : int;$
$\quad t(i : 0 \leq i < n) : array\ of\ int;$
$\quad \{0 \leq q < p\}$
$\quad \mathbf{par} \quad q : 0 \leq q < p$
$\quad\quad |[w : int;$
$\quad\quad \{Q.q : 0 < n\}$
$\quad\quad S.q$
$\quad\quad \{R.q : w = (\sum i : 0 \leq i < n : t(i))$
$\quad\quad]|$
$\quad \mathbf{rap}$
$]|$

Dacă $p = 1$ avem o specificaţie pentru un program secvenţial.

6.1.3 Invarianţi

Modalitatea de a obţine un proces parametrizat S pornind de la specificaţii este similară cu metodele folosite în programarea secvenţială. Se încearcă obţinerea unui invariant prin calcul. Programul este derivat prin calcularea condiţiilor necesare pentru a menţine invariantul. Într-o derivare, fiecare subproblemă care se identifică va fi mai simplă decât problema iniţială. Procesul de rafinare este repetat până când construcţia programului text care respectă specificaţia devine trivială. Invarianţii sunt de asemenea parametrizaţi, consecinţă naturală a faptului că specificaţiile sunt parametrizate. Se folosesc astfel tehnicile programării secvenţiale. Găsirea invariantului poate fi făcută mai uşor dacă se consideră distribuţia datelor stabilită dinainte. Este necesară rescrierea pre- şi post- condiţiilor în aşa fel încât să poată fi identificate specificaţiile locale şi globale. O *specificaţie locală* face referire la datele locale procesului şi ea poate fi satisfacută de un program secvenţial. O *specificaţie globală* cere anumite forme de coordonare între procese; anumite procese trebuie să interacţioneze între ele prin schimb de mesaje pentru a satisface specificaţia.

Exemplul 6.2 Pentru specificaţia dată în exemplul anterior, fie $O.q$ setul de indecşi ai elementelor tabloului t, care sunt repartizate procesului q. Ca prim pas spre a obţine un invariant se identifică două subprobleme: înregistrarea sumelor locale (formate din însumarea elementelor locale procesului) în variabila locală s şi însumarea acestor valori locale. Acesta poate fi derivat prin rescrierea predicatului $R.q$:

$$
\begin{array}{l}
\quad (\sum i : 0 \le i < n : t(i)) \\
= \{\text{rescrierea domeniului}\} \\
\quad (\sum q : 0 \le q < p : (\sum i : 0 \le i < O.q : t(i)) \\
= \{\text{introducerea sumelor parţiale}\} \\
\quad (\sum q : 0 \le q < p : lsum.q),
\end{array}
$$

unde $lsum.q = (\sum i : 0 \le i < O.q : t(i))$, pentru toţi $q : 0 \le q < p$.

S-au identificat astfel două subprobleme cu postcondiţiile $R0.q$ şi $R1.q$, scrise în specificaţia dată în continuare:

$S.q ::$
$|[s : int;$
$\quad S0.q$
$\quad \{R0.q : s = lsum.q\}$
$\quad ; S1.q$
$\quad \{R1.q : w = (\sum q : 0 \le q < p : lsum.q)\}$
$]|$

Pentru procesul $S0.q$ este uşor să obţinem un invariant; procesul $S1.q$ necesită o comunicaţie globală între procese.

6.1.4 Corectitudinea

Prin acest mod de elaborare a programelor paralele se doreşte obţinerea unor programe corecte prin construcţie. Demonstrarea corectitudinii programelor paralele se reduce la demonstrarea corectitudinii aplicării regulilor de construcţie. În secţiunile necomunicante ale unui proces parametrizat, avem doar aserţiuni şi liste de instrucţiuni. Corectitudinea poate fi demonstrată folosind aserţiunile şi regulile de demonstrare. Singura comunicare permisă între procese este transmiterea de mesaje. Partea unei instanţe proces care conţine asemenea instrucţiuni de comunicaţie este numită subproces de comunicaţie. Înaintea trimiterii unei valori este făcută o aserţiune explicită asupra valorii comunicate. Aserţiunile asupra acestor valori sunt exprimate în termenii unor expresii globale. În acest fel interferenţa demonstraţiilor este evitată.

În general, funcţionalitatea unui proces de comunicaţie este simplă. Este suficient să se studieze câteva procese de comunicaţie frecvent folosite şi implementarea lor pe diferite reţele de comunicaţie. Acestea vor fi adaptate pentru fiecare program paralel concret. Procesele de comunicaţie sunt parametrizate şi ele, şi pot fi specificate izolat. Aceasta permite o singură demonstrare de corectitudine şi facilitează această demonstrare. Se pune totuşi o restricţie puternică asupra proceselor parametrizate de comunicaţie: toate comunicaţiile apar între instanţele aceluiaşi proces parametrizat. Un astfel de proces(instanţă) se numeste *închis la comunicaţii ("communication closed")*. Privind programul paralel descompus pe nivele - fiecare nivel poate fi ori o listă de instrucţiuni ori instrucţiuni de comunicaţie. Nivelele pot fi separate sintactic prin punct şi virgulă şi sunt specificate de pre- şi post- condiţii.

$$
\begin{array}{ll}
S.q :: & |[\\
 & S0.q \\
 & ; C0.q \\
 & ; S1.q \\
 &]|
\end{array}
$$

Instanţele procesului $S0$ formează un nivel care are pre- şi post- condiţii parametrizate; instanţele procesului de comunicaţie $C0$ formează de asemenea un nivel. E posibil ca fiecare proces $S1.q$ să fie mai departe descompus.

$$
\begin{array}{l}
S1.q :: \\
|[\quad \textbf{do } B.q \rightarrow \\
\quad\quad S2.q \\
\quad\quad ; C1.q \\
\quad\quad ; S3.q \\
\quad\quad \textbf{od} \\
]|
\end{array}
$$

$B.q$ este o expresie booleană în termenii variabilelor locale procesului q. Corectitudinea parţială a buclei din $S1$ este dovedită de $\{P.q \wedge B.q\}S2.q; C1.q; S3.q\{P.q\}$ pentru un invariant $P.q$ al lui $S1.q$. Se formează 3 nivele logice în buclă. E posibil ca instanţele proces să se execute în nivele diferite, dar logic $S1$ e descompus în nivele ale căror corectitudine se demonstrează separat.

Termenul *nivel* a fost introdus pentru prima dată, în contextul programării paralele în "Decomposition of distributed programs into communication-closed layers" de T.Elrad şi N.Frances (1982). În metodologia prezentată aici, conceptul de nivel nu este folosit doar pentru verificarea corectitudinii, ci este o parte a metodologiei de elaborare. Programatorul este responsabil de formarea nivelelor logice.

6.1.5 Notaţia programelor

Notaţia programelor se bazează pe limbajul comenzilor gardate, definit de Dijkstra[47]. Declararea variabilelor se face în stilul Pascal, extinsă cu regulile de specificare de domeniu. Simbolurile $|[x \ldots]|$ delimitează domeniul variabilei x. Exemple:

$$x : int; \quad z : real;$$
$$t(i : 0 \leq i < n) : array \quad of \quad real.$$

Construcţiile program sunt:

- *abort* – oprire necondiţionată
- *skip* – instrucţiune inoperantă
- $x := e$ – instrucţiune de atribuire
- $S0; S1$ – compoziţie secvenţială
- **if** $B0 \rightarrow S0 \; [] B1 \rightarrow S1 \; \ldots$ **fi** – construcţie alternativă
- **do** $B0 \rightarrow S0 \; [] B1 \rightarrow S1 \; \ldots$ **od** – construcţie ciclică
  $B0$ şi $B1$ sunt expresii booleene.

Se foloseşte în plus, o extensie a notaţiei Dijkstra:

- **for all** $i : i \in Set : S.i$ **lla rof** – instrucţiune *for* în care ordinea de efectuare a instrucţiunilor $S.i$ este arbitrară.

  Exemplu:

  $$s := 0$$
  $$; \textbf{for all } i : i \in O.q : s := s + t(i) \textbf{ lla rof}$$

  Compoziţia paralelă este notată prin:

  $$\textbf{par } q : 0 \leq q < p : S.q \textbf{ rap}$$

şi specifică faptul că procesele $S.q$ încep în acelaşi moment şi se execută în paralel. O instrucţiune **par** se termină atunci când toate procesele constituente se termină.

Exemplul 6.3 (**specificare implicită**)

$$\textbf{par}\ s,t : 0 \leq s < M, 0 \leq t < N : S.s.t\ \textbf{rap}$$

unde $p = M * N$, specifică compoziţia paralelă a p procese $S.s.t$, fiecare identificat de perechea (s,t) cu $0 \leq s < M, 0 \leq t < N$.

Exemplul 6.4 (**specificare explicită**)

$$\textbf{par}\ C0.q, C1.q\ \textbf{rap}$$

denotă compoziţia paralelă a două procese paralele şi este folosită în procesele de comunicaţie pentru a exprima execuţia simultană a unor comunicaţii.

Comunicaţia este exprimată prin instrucţiunile:

r ! e – procesul curent trimite valoarea e spre procesul r,

s ? x – procesul curent aşteaptă primirea unei valori de la procesul s, pe care o va reţine în variabila x.

Prin expresiile:

$$s :: r!e$$
$$r :: s?x$$

se defineşte un canal de comunicaţie de la procesul s la procesul r. Un asemenea canal este partajat între două procese şi direcţia este de la emiţător la receptor. Orice instrucţiune de intrare(recepţie) are corespunzător o instrucţiune de ieşire(transmisie) şi viceversa. Nu se folosesc nume specifice pentru canale, dar se pot folosi numerele de proces pentru a identifica un canal. Instrucţiunea $(q+1)!10$ în procesul q denotă faptul că procesul q trimite valoarea 10 spre procesul $(q+1)$. Expresia $(q+1)$ este numită *expresie canal*. Mecanismul de comunicare poate fi sincron sau asincron. Se consideră însă, ca mesajele(valorile) transmise de un proces ajung întotdeauna, într-un timp arbitrar, la procesul receptor, fără duplicate şi în ordinea în care au fost transmise. În acest fel, specificaţiile se referă doar la procesele parametrizate şi nu trebuie introduse specificaţii de canal.

6.1.6 Reguli de demonstrare

O demonstraţie "slabă" (parţial corectitudine) a unui program paralel în absenţa "deadlock"-ului este ca şi în cazul unui program secvenţial, legată de adnotarea programului şi de folosirea regulilor de demonstrare. Un program adnotat are aserţiuni înainte şi după textul programului, în conformitate cu specificaţiile şi aserţiuni între instrucţiuni. O asemenea aserţiune sau triplet Hoare:

$$\{Q.q\}S.q\{R.q\}$$

este validă dacă este o axiomă sau dacă se obţine prin aplicarea regulilor de inferenţă. Pornind de la axiomatizarea Hoare avem următoarele 2 axiome şi 4 reguli de inferenţă: (notăm : $P[e\backslash x]$ predicatul care se obţine din predicatul P, dacă se înlocuieşte e cu x).

1. Axioma Skip:

$$\{P\}\ skip\ \{P\}$$

2. Axioma Atribuire:

$$\{P[e\backslash x]\}\ x := e\ \{P\}$$

3. Regula Compoziţie:

$$\frac{\{P\}\ S_0\ \{Q\}, \{Q\}\ S_1\ \{R\}}{\{P\}\ S_0; S_1\ \{R\}}$$

4. Regula If:

$$\frac{(i : 0 \leq i \leq m : \{P \wedge B_i\}\ S_i\ \{Q\})}{\{P\}\ \textbf{if}\ B_0 \rightarrow S_0\ []\ B_1 \rightarrow S_1\ [] \ldots []B_m \rightarrow S_m\ \textbf{fi}\ \{Q\}}$$

5. Regula Do:

$$\frac{(i : 0 \leq i \leq m : \{P\ \wedge B_i\}\ S_i\ \{P\})}{\{P\}\ \textbf{do}\ B_0 \rightarrow S_0\ []\ B_1 \rightarrow S_1\ [] \ldots []B_m \rightarrow S_m\ \textbf{od}\ \{P \wedge \neg(B_0 \vee \ldots \vee B_m\}}$$

6. Regula Consecinţă:

$$\frac{P \Rightarrow P', \{P'\}\ S\ \{Q'\}, Q' \Rightarrow Q}{\{P\}\ S\ \{Q\}}$$

Pentru programele paralele se adaugă reguli şi axiome noi.

7. Regula Par

$$\frac{(\forall i : 0 \leq i < p : \{Q.i\}\ S.i\ \{R.i\})}{\{(\forall i : 0 \leq i < p : Q.i)\}\ \textbf{par}\ q : 0 \leq q < p : S.q\ \textbf{rap}\ \{(\forall i : 0 \leq i < p : R.i)\}}$$

Dacă toate precondiţiile sunt adevărate atunci, dacă procesele $S.q$ se termină, toate postcondiţiile sunt adevărate. Spaţiile de stare pentru procese sunt disjuncte, deoarece fiecare proces are propria mulţime de variabile locale. Execuţia unui proces nu poate altera starea altui proces, decât dacă între cele două procese apar comunicaţii. Regula Par spune că, dacă am construit un proces parametrizat $S.q$ cu pre- şi post- condiţiile corespunzătoare, atunci avem un program paralel. Derivarea unui program paralel poate fi începută în două moduri diferite. Cel mai natural este de a începe de la postcondiţia locală şi a încerca apropierea de precondiţie prin rafinare. Cealaltă cale constă în formularea de invarianţi globali, un invariant despre datele distribuite, ca un întreg. Invarianţii globali servesc ca intermediari pentru invarianţii locali.

Pentru comunicaţii se introduce o axiomă şi o regula de inferenţă.

8. Axioma Input:

$$r ::\ \{true\}\ s?x\ \{M.x\},$$

unde $M.x$ este un predicat în termenii variabilei locale x, a procesului r şi a numărului de proces s care trimite valoarea. Predicatul $M.x$ satisface condiţia: $(\exists x :: M.x)$, adică există valori ale lui x pentru care predicatul $M.x$ este satisfăcut (nu reprezintă o imposibilitate). Axioma asigură faptul că orice poate fi concluzionat după recepţia unei valori. Deşi această axiomă poate fi interpretată ca fiind foarte puternică datorită faptului că în partea dreaptă poate fi o concluzie puternică, totuşi dacă considerăm axioma input izolată cu un singur proces în execuţie, avem o instrucţiune input blocată şi orice predicat $M.x$, care nu e identic fals, poate fi considerat adevărat după terminare.

Regula de inferenţă pentru comunicaţie este următoarea:

9. Regula Comunicaţie:

$$\frac{r :: \{true\}\ s?x\ \{M.x\}}{s :: \{M.x[x\backslash e]\}\ r!e\ \{M.x[x\backslash e]\}}$$

unde $M.x$ este un predicat în termenii variabilei locale x, a procesului r şi numărului de proces s, iar e este o expresie locală procesului s.

Motivaţia introducerii regulii Comunicaţie în construirea unui proces de comunicaţie este următoarea: Avem disponibile postcondiţia procesului receptor r şi în particular o aserţiune $M.x$ asupra valorii care va fi comunicată. Precondiţia procesului emiţător e uşor de obţinut prin substituţie ($M.x[x\backslash e]$). Trimiterea valorii nu schimbă starea procesului s, deci postcondiţia este aceeaşi cu precondiţia. Nu se fac aserţiuni despre starea globală. Această regulă de demonstrare e mai degrabă slabă, pentru că nu se fac aserţiuni decât asupra valorii comunicate. Pentru această metodă de construcţie de programe, regula aceasta este suficientă şi poate fi folosită şi pentru transmiterea asincronă.

Exemplul 6.5 Fie $M.x \equiv impar.x$ şi $e = y + 1$.

$$\begin{aligned} & M.x[x\backslash e] \\ \equiv\ & \{\text{substituţie}\} \\ & impar.(y+1) \\ \equiv\ & \{\text{calcul}\} \\ & par.y\ \{\text{precondiţia pentru } s\} \end{aligned}$$

Aplicarea regulii Comunicaţie:

$$\frac{r :: \{true\}\ s?x\ \{impar.x\}}{s :: \{par.y\}\ r!(y+1)\ \{par.y\}}$$

Se observă că această comunicaţie este analoagă unei atribuiri $x := e$. Partea dreaptă este evaluată de procesul s şi rezultatul e comunicat lui r, care atribuie valoarea primită variabilei x.

Observaţie: Este permisă apariţia numărului de proces în expresia e.

Exemplul 6.6 Considerăm două procese de comunicaţie care se execută în paralel $C.0, C.1$, iar $M.x.q \equiv x = 1 - q$ pentru $q = 0, 1$.

$C.q ::$
$|[x : int;$
$\quad \{0 \leq q < 2 \wedge B.q\}$
$\quad$ **par** $\{B.q\}\ (1-q)!q\ \{B.q\},$
$\qquad \{B.q\}\ (1-q)?x\ \{M.x.q\}$
$\quad$ **rap**
$\quad \{M.x.q \wedge B.q\}$

$]|$

$B.q$ este precondiţia necesară, care nu e încă cunoscută. Pentru a folosi regulile de comunicaţie, trebuie identificate comunicaţiile pereche, adică:

$$\begin{array}{ll} 1-q :: & \{B.(1-q)\}\ q?x\ \{M.x.(1-q)\} \\ q :: & \{B.q\}\ (1-q)!q\ \{B.q\} \end{array}$$

Folosim regula de inferenţă cu $r = 1-q$ şi $s = q$:

$$\frac{(1-q) ::\ \{true\}\ q?x\ \{M.x.(1-q)\}}{q ::\ \{M.x(1-q)[x\backslash q]\}\ (1-q)!(q)\ \{M.x.(1-q)[x\backslash q]\}}$$

Pentru a găsi precondiţia $B.q$ folosim următoarea derivare:

$$\begin{array}{ll} & M.x.(1-q)[x\backslash e] \\ \equiv & \{\text{definiţia lui } M.x.(1-q), e\} \\ & x = q[x\backslash q] \\ \equiv & \{\text{substituţie}\} \\ & true \\ \equiv & \\ & B.q \end{array}$$

În concluzie $B.q = true$.

Aserţiunile în procesele de comunicaţie sunt adnotate în conformitate cu regula Comunicaţie. Adică, pentru procesul care recepţionează, se consideră o postcondiţie după instrucţiunea ?, iar în procesul care trimite, o precondiţie chiar înaintea instrucţiunii !.

6.1.7 Procese de comunicaţie

Pentru a face analiza proceselor de comunicaţie, considerăm un exemplu concret de proces parametrizat de comunicaţie: *broadcast*. Fie s un număr de proces şi X o valoare:

$$\begin{array}{l}\{q \neq s \vee x = X\}\\ \quad C.q\\ \{x = X\}\end{array}$$

Valoarea X este trimisă de procesul s tuturor celorlalte procese. Există variaţii de broadcast: multibroadcast(mai mulţi emităţori), sau broadcast la o submulţime de receptori. Specificaţia procesului e simplă, dar implementarea nu. Este nevoie să se facă anumite presupuneri, referitor la canalele care sunt folosite într-un broadcast.

Există de asemenea restricţii dictate de reţeaua fizică de procesoare. Într-o implementare, procesele sunt atribuite procesoarelor şi canalele căilor de comunicaţie dintre procesoare. De aceea este bine să se evite pe cât posibil acest gen de comunicaţie. Un program paralel poate fi considerat un graf neorientat cu procese în noduri. Fiecare muchie reprezintă două canale de comunicaţie în direcţii opuse. În plus, deseori se cere ca o asemenea *reţea de calcul* să poată să varieze ca structură în timpul calculului. Pe de altă parte o reţea de procesoare poate fi considerată ca un "graf de implementare" fixat, reprezentând un sistem multiprocesor. Este posibil să se scrie procese de comunicaţie independent de orice topologie, folosind apoi diferite transformări de mapare pe diferite reţele. În general, ne limităm la a da specificaţiile unui proces de comunicaţie, lucru care este suficient pentru a demonstra corectitudinea programului.

6.1.8 Complexitatea

Aşa cum am specificat şi în Capitolul 1, este de dorit ca şi complexitatea-timp a programelor paralele, să fie exprimată în formule matematice, în termenii n =dimensiunea datelor de intrare şi p = numărul de procese ale programului. Un proces poate calcula, comunica, sau aştepta la un anumit moment. Considerăm două măsuri: complexitatea computaţională ($T_f.p.n$) şi complexitatea de comunicaţie ($T_c.p.n$), incluzând fiecare o parte de aşteptare. T_c este determinată de numărul de paşi de comunicaţie necesari pentru executarea procesului de comunicaţie. Un pas de comunicaţie este comunicarea unei valori unui proces vecin. Reţeaua de comunicaţie poate permite ca anumiţi paşi de comunicaţie să se execute în paralel. Aceasta înseamnă că $T_c.p.n$ este cel mult egal cu numărul total de paşi de comunicaţie. $T_c.p.n$ depinde şi de mărimea mesajelor transmise. Presupunem că avem comunicaţii de mesaje cu aceeaşi mărime (pentru mesaje mai mari considerăm mai mulţi paşi de comunicaţie).

Exemplul 6.7 (**Broadcast**) Se poate specifica implementarea unui broadcast pe o reţea lanţ. Considerăm că procesul $s, 0 \leq s < p$ trimite valoarea X tuturor celorlalte procese.

$|[x : int;$
$\quad \{0 \leq s < p\}$
$\quad \{q \neq s \vee x = X\}$
$\quad$ **if** $q < s \ \rightarrow \ (q+1)?x$
$\quad\quad [] s < q \ \rightarrow \ (q-1)?x$
$\quad$ **fi**
$\quad \{x = X\}$
$\quad ;$ **par if** $0 < q \leq s \ \rightarrow \ (q-1)!x$ **fi**
$\quad\quad ,$ **if** $s \leq q < p-1 \ \rightarrow \ (q+1)!x$ **fi**
$\quad$ **rap**
$\quad \{x = X\}$
$]|$

Se poate demonstra corectitudinea prin inducţie după p.

În cazul cel mai defavorabil $T_c.p.n = p - 1$. Pentru cazul mediu

$$T_c.p.n = \frac{1}{p}(\sum s : 0 \leq s < p : max(s, p-1-s)) \simeq \frac{3p}{4}.$$

Dacă valoarea se transmite doar de la un proces s la un proces t atunci $T_c.p.n$ în cazul cel mai defavorabil rămâne $p-1$, dar în cazul mediu este $\simeq \frac{p}{3}$, (n_k numărul cazurilor pentru care $|s-t| = k$; $n_k = 2(p-k)$; $T_c.p.n = (\sum k : 0 \leq k < p : \frac{2k(p-k)}{A_p^2})$).

Un proces broadcast pe un hipercub necesită $\log_2 p$ paşi.

Exemplul 6.8 (**Suma**) Considerăm din nou problema adunării a n numere, a cărei specificaţie am dat-o în Exemplul 6.2. Implementarea procesului de comunicaţie $S1.q$ pe un arbore binar complet va conţine două etape:

- calcularea sumei în rădăcină;
- rădăcina (procesul 0) trimite valoarea sumei, în broadcast, tuturor celorlalte procese.

Definim pentru fiecare proces următoarele funcţii:

$$tata.q = \begin{cases} \frac{q-1}{2}, & \text{dacă } q > 0 \\ 0, & \text{altfel} \end{cases}$$

$$\begin{aligned} copii.q &= \{\forall i : i < p \wedge (i = 2q+1 \vee i = 2q+2) : i\} \\ arbore.q &= \{q\} \cup \{\cup k : k \in copii.q : arbore.k\} \end{aligned}$$

Se observă că pentru rădăcină avem $tata.q = q$.

Procesul $S1.q$ combină sumele parţiale în suma globală. Se calculează mai întâi în procesul 0 şi de acolo se transmite şi celorlalte procese. În procesul $S1.q$, calculul porneşte

de la frunze, care trimit valoarea locală s proceselor părinte şi acestea vor însuma cele 2 valori primite. Se continuă astfel până se ajunge la rădăcină. Pentru recepţionarea celor 2 valori ar fi suficiente 2 variabile locale, dar pentru o notaţie mai uniformă folosim un tablou x cu p elemente:

$S1.q ::$
$|[x(0 \leq i < p) : array\ of\ int;$
  par $r : r \in copii.q :$
    $r?x(r)$
    $\{x(r) = (\sum k : k \in arbore.r : lsum.k)\}$
  rap
  $; w := s$
  $\{w = lsum.q\}$
  ; **for all** $r : r \in copii.q : w := w + x(r)$ **lla rof**
  $\{w = (\sum k : k \in arbore.q : lsum.k)\}$
  ; **if** $tata.q \neq q \ \rightarrow tata.q!w$
  $; tata.q?w$
  fi
  $\{w = (\sum q : 0 \leq q < p : lsum.q)\}$
  ; **for all** $r : r \in copii.q : r!w$ **lla rof**
$]|$

Complexitatea $T_c.p.n$ este $O(\log_2 p)$. Dacă implementăm $S1.q$ pe o reţea lanţ atunci $T_c.p.n = O(p)$. Concluzia este că reţeaua de comunicaţie aleasă influenţează mult complexitatea-timp de comunicaţie.

Exemplul 6.9 (**Pipeline**) Considerăm un proces care face o transmisie broadcast a n valori ($n \gg p > 1$). Folosind tehnica pipelining, pe o reţea lanţ, rezultă $T_c.p.n = n - 1 + p - 1$, unde $p - 1$ reprezintă lungimea pipe-ului(conductei), sau timpul de start. Pentru $n = 1$ avem $T_c.p.n = p - 1$. În acest caz $T_c.p.n$ e determinat de numărul de valori de comunicat, nu de reţea.

Programele paralele pe care dorim să le construim sunt structurate pe nivele, fiecare nivel fiind fie un program secvenţial, fie un proces de comunicaţie. Dacă presupunem comunicaţii sincrone, sincronizarea între programele secvenţiale şi procesele de comunicaţie este naturală. Complexitatea-timp a unui nivel computaţional este determinată de cea mai înceată instanţă din acel nivel (cu cele mai multe operaţii). Complexitatea unui nivel de comunicaţie este dată de complexitatea comunicaţiei. Complexitatea unui program paralel se obţine prin însumarea complexităţii fiecărui nivel. Astfel, obţinem o limită superioară a complexităţii unui program paralel. Dacă nivelele sunt bine echilibrate atunci această limită estimează de fapt complexitatea-timp. Dacă comunicaţiile

sunt asincrone, atunci se permite suprapunerea calculelor cu comunicaţiile în interiorul unui proces; pentru această combinaţie e dificil de evaluat complexitatea. Din acest motiv, rezultatele de complexitate prezentate în continuare sunt valabile doar pentru procese de comunicaţie sincrone.

Pentru a lega complexitatea computaţională de cea de comunicaţie, introducem o valoare α definită ca fiind raportul: $\alpha = \frac{t_c}{t_f}$, unde

- t_c = timpul necesar comunicaţiei unei singure valori de mărime fixă (*int* sau *real*)
- t_f = timpul necesar efectuării unei operaţii elementare $(+, -, *, /)$.

Prin experimente [111] s-a constatat că:

$$\alpha \simeq \begin{cases} 4.5, & \text{pentru transputere} \\ 150 - 750, & \text{pe reţele hipercub} \\ 0.5, & \text{pentru VLSI.} \end{cases}$$

Complexitatea totală este:

$$T.p.n = T_f.p.n + \alpha * T_c.p.n$$

$T_f.p.n =$ complexitatea computatională exprimată în număr de t_f,

$T_c.p.n =$ complexitatea comunicaţională exprimată în număr de t_c

.

Timpul executării unui program paralel va fi $T.p.n * t_f$.

Exemplul 6.10 Evaluăm complexitatea-timp a programului paralel prezentat în Exemplul 6.2 şi reluat în Exemplul 6.8 (suma a n elemente folosind p procese). Pentru $S0.q$: avem $T_f.p.n = (\max q : 0 \leq q < p : |O.q| - 1)$, unde $O.q$ este numărul de elemente distribuite procesului q. Dacă distribuţia este echilibrată avem $T_f.p.n = (n + p - 1)/p - 1$. Pentru $S1.q$, complexitatea comunicaţională este $T_c.p.n = 2(\alpha^{-1} + 1) * l$, unde l = lungimea celei mai lungi căi în reţea = $log_2 p$. Fiecare nod neterminal face 2 operaţii de adunare şi 2 comunicaţii. Deci $T.p.n = (n + p - 1)/p + 2(\alpha + 1) * l - 1$ ($T.1.n = n - 1$).

Acceleraţia este

$$S.p.n = \frac{T_{sec}.n}{T.p.n} = \frac{n-1}{(n+p-1)/p + 2(\alpha+1) * l - 1},$$

iar eficienţa este

$$E.p.n = \frac{S.p.n}{p}.$$

Eficienţa este cel puţin 0.2 dacă $\frac{n}{p} > 2(\alpha + 1) * l$.

6.1.9 Regula ParSeq

Se poate considera descrierea unui program paralel, ca fiind compoziţia secvenţială a unui număr de componente paralele, fiecare corespunzătoare unei faze a programului. Această observaţie poate fi folosită atât la construcţia cât şi la verificarea programelor paralele şi a fost evidenţiată prima dată de Elrad şi Frances [56]. Ei au demonstrat că un program paralel poate fi descompus în aşa numitele nivele închise la comunicaţii, comunicaţii între nivele diferite, neputând apare. Se consideră un program paralel:

$$S :: \textbf{par}\ q : 0 \leq q < p : S.q\ \textbf{rap},$$

în care fiecare proces $S.q$ e reprezentat ca fiind $S.q : S_0.q; S_1.q; ...; S_{d-1}.q$, unde S_i sunt simple instrucţiuni : skip, atribuire, sau comunicaţie; d este adâncimea descompunerii, care poate fi uniformă dacă se foloseşte instrucţiunea skip. Un nivel $L_j, 0 \leq j < d$, constă din L_j : **par** $q : 0 \leq q < p : S_j.q$ **rap**. Un nivel este numit *închis la comunicaţii* dacă şi numai dacă acţiunile care au loc în acest nivel nu depăşesc marginile nivelului. Altfel spus: orice operaţie ! are corespunzător o operaţie ? în acelaşi nivel şi viceversa. Descompunerea în nivele este: $S :: L_0; L_1; ...; L_{d-1}$. Descompunerea se numeşte *sigură* dacă toate nivelele sunt închise la comunicaţii. Principalul lor rezultat este:

Un program distribuit este echivalent cu orice descompunere sigură pe nivele a sa.

Două programe se consideră *echivalente* dacă pre- şi post- condiţiile lor sunt aceleaşi. Acest rezultat se poate demonstra prin inducţie după d.

Reciproca:

$$L_{d-1}; L_d \text{ e echivalent cu } \textbf{par}\ q : 0 \leq q < p : S_{d-1}.q; S_d.q\ \textbf{rap},$$

nu este demonstrată.

Pentru cazul $p = 2$ avem :

Definiţia 6.1 (Regula Par-Seq)

Fie 4 *procese* $S_i.j, 0 \leq i, j < 2$, *despre care se ştie că se termină şi care pot interacţiona doar în următorul mod:*

- *fiecare* $S_i.0$ *poate interacţiona prin comunicare cu* $S_i.1$,
- *fiecare* $S_0.j$ *poate interacţiona prin variabile partajate cu* $S_1.j$.

Atunci

$$\textbf{par}\ S_0.0, S_0.1\ \textbf{rap}\ ;\ \textbf{par}\ S_1.0, S_1.1\ \textbf{rap}$$

e echivalent cu

$$\textbf{par}\ S_0.0; S_1.0\ ,\ S_0.1; S_1.1\ \textbf{rap}$$

Regula parseq afirmă că sincronizarea globală poate fi îndepărtată dacă se impun reguli stricte de interacţiune. Ordinea calculelor în procese poate fi exprimată formal şi prin teoria urmei.

În teoria dezvoltată aici, un program paralel constă din p instanţe ale unui singur proces parametrizat. Un asemenea proces e construit prin rafinare, într-o secvenţă de procese secvenţiale şi procese de comunicaţie. Toate instanţele unui proces parametrizat formează un nivel şi nivelele sunt separate sintactic prin simbolul ;. Procesele parametrizate care aparţin unui nivel sunt specificate prin pre- şi post- conditii şi pot fi *studiate în izolare*. Acest principiu strict de elaborare, permite construirea de programe paralele eficiente, deoarece regula parseq arată o modalitate de eliminare a sincronizării stricte între nivele.

6.2 Distribuţia datelor

Distribuţia datelor are un mare impact asupra numărului de comunicaţii pentru un program paralel. Vom prezenta două tipuri de distribuţii, clasificate în funcţie de numărul de procese care conţin o anumită dată. Dacă un element-dată este atribuit unui singur proces avem distribuţii simple (univoce), iar dacă un element-dată poate fi atribuit mai multor procesoare simultan atunci avem distribuţii multivoce. Distribuţiile le considerăm *statice* – nu se schimbă distribuţia datelor în timpul execuţiei programului. Este necesar ca numărul de elemente distribuite pe un proces să fie, dacă este posibil, egal pentru toate procesele, pentru a asigura o încărcare de calcul echilibrată, a proceselor, deci un timp de aşteptare mic. Distribuţia datelor determină care valori vor trebui comunicate într-un program. Numărului total de comunicaţii care este necesar pentru un anumit program paralel, poate fi evaluat în cele mai multe cazuri, înainte de elaborarea efectivă a programului, în funcţie de distribuţia datelor aleasă. Analiza numărului de comunicaţii permite alegerea unei distribuţii cât mai convenabile din punct de vedere al complexităţii de comunicare, iar distribuţia determină deasemenea construcţia programului paralel.

6.2.1 Distribuţii simple

Distribuţiile simple sunt caracterizate de:

- numărul de elemente-dată care sunt atribuite unui proces,
- distribuţia elementelor-dată pe procese.

Distribuţii unidimensionale

Distribuţiile unidimensionale distribuie date de tip vector pe o mulţime de procese.

Definiţia 6.2 (Distribuţie) *$D = (\delta, A, B)$ se numeşte* distribuţie*, dacă A, B sunt mulţimi finite, A reprezentând mulţimea datelor, B reprezentând mulţimea proceselor şi δ este o funcţie de la A la B.*

Distribuţia unui tablou de date t de lungime n pe p procese ($p \leq n$), poate fi specificată prin $(\delta, \overline{n}, \overline{p})$.

Există 3 modalităţi uzuale de distribuire a datelor de tip tablou:

- distribuţie *identic*, pentru cazul în care $n = p$, prin care data a i-a se atribuie procesului i:
$$(\delta, \overline{p}, \overline{p}),\ \delta.i = i,$$
- distribuţie *liniar*, prin care se atribuie segmente continue de date de lungime aproximativ egală fiecărui proces:
$$(\delta, \overline{n}, \overline{p}),\ \delta.i = i/(n/p),\ \text{dacă } p|n,\ \text{sau în cazul general}$$
$$\delta.i = i/(m+1)\ max\ (i - n\%p)/m,\ \text{unde } m = n/p,$$
- distribuţie *ciclic* prin care se atribuie al i-lea element-dată procesului $i\%p$:
$$(\delta, \overline{n}, \overline{p}), \delta.i = i\%p.$$

Definiţia 6.3 *Două distribuţii* $(\delta_0, \overline{n}, \overline{p})$ *şi* $(\delta_1, \overline{n}, \overline{p})$ *se numesc* echivalente *dacă şi numai dacă*
$$(\exists \pi : \pi\ \textit{permutare a mulţimii}\ \overline{p} : \delta_0 = \pi \circ \delta_1)$$

Notăm cu:
$$O^{\delta}.q = \{i : 0 \leq i < n \wedge \delta.i = q : i\}$$
mulţimea tuturor datelor atribuite procesului q. Mulţimile O^{δ} formează, în acest caz, o partiţie pentru mulţimea $\overline{n}$.

Notăm cu $Ma(\delta)$ numărul maxim de elemente-dată atribuite unui proces de distribuţia δ, şi cu $Mi(\delta)$ numărul minim de elemente-dată atribuite unui proces de distribuţia δ:
$$Ma(\delta) = (max\ q : 0 \leq q < p : |O^{\delta}.q|)$$
$$Mi(\delta) = (min\ q : 0 \leq q < p : |O^{\delta}.q|)$$

Propoziţia 6.1 (Ma(liniar)=Ma(ciclic))
Pentru distribuţiile liniar *şi* ciclic *este adevărată următoarea afirmaţie:*
$$(\forall q : 0 \leq q < p : |O^{liniar}.q| = |O^{ciclic}.q| = (n + p - 1 - q)/p).$$

Demonstraţie:

$|O^{ciclic}.q|$
$\equiv$ {calcul}
$|\{\forall i : 0 \leq i < n \wedge i\%p = q : i\}|$
$\equiv$ {împărţirea domeniului, calcul}
$|\{\forall i : 0 \leq i < (n/p) * p \wedge i\%p = q : i\}| + |\{\forall i : (n/p) * p \leq i < n \wedge i\%p = q : i\}|$
$\equiv$ {calcul}
$n/p + |\{(\forall i : 0 \leq i < (n\%p) \wedge i = q : i\}|$
$\equiv$ {calcul}
$(n + p - 1 - q)/p$

Pentru distribuţia *liniar* avem:

$$O^{liniar}.q = \{i : l.q \leq i < l.(q+1) : i\},$$

unde

$$l = (\lambda q * q(n/p) + min(q\,(n\%p)).$$

Ca urmare $|O^{liniar}.q| = l.(q+1) - l.q = (n+p-1-q)/p$.

Definiţia 6.4 *O distribuţie* (δ, n, p) *se numeşte* w-echilibrată, *dacă* $Ma(\delta) - Mi(\delta) = w$. *Distribuţia se numeşte* omogenă *dacă* $w = 1$ *şi* perfectă *dacă* $w = 0$.

Compunerea distribuţiilor

Prin compunerea a două distribuţii se pot obţine noi distribuţii.

Definiţia 6.5 Compunerea *a două distribuţii ,* $D0 = (\delta_0, \overline{m}, \overline{M})$ *şi* $D1 = (\delta_1, \overline{n}, \overline{N})$ *cu* $M = n$, *se defineşte prin* $D1 \circ D0 = (\delta_1 \circ \delta_0, \overline{m}, \overline{N})$.

Exemplul 6.11 Considerăm distribuţiile $liniar = (\delta^{liniar}, \overline{n}, \overline{m})$ şi $ciclic = (\delta^{ciclic}, \overline{m}, \overline{p})$; compunerea lor este distribuţia $ciclic \circ liniar = (\delta, \overline{n}, \overline{p}), \delta.i = (i/(n/m)\%p)$. Dacă $m = n$ atunci se obţine distribuţia *ciclic*, iar dacă $m = p$ atunci se obţine distribuţia *liniar*.

Observaţie: compunerea distribuţiilor nu păstrează proprietatea de omogenitate.

Exemplul 6.12 Considerăm două distribuţii concrete: $(\delta_0, \overline{m}, \overline{M})$ şi $(\delta_1, \overline{n}, \overline{N})$, cu $m = 12, M = n = 5, N = 3$.

i	0	1	2	3	4	5	6	7	8	9	10	11
$\delta_0.i$	0	0	0	1	1	1	2	2	3	3	4	4

i	0	1	2	3	4
$\delta_1.i$	0	0	1	1	2

$Ma(\delta_1) - Mi(\delta_1) = Ma(\delta_0) - Mi(\delta_0) = 1$ deci δ_0 si δ_1 sunt distribuţii omogene.

i	0	1	2	2	4	5	6	7	8	9	10	11
$(\delta_1 \circ \delta_0).i$	0	0	0	0	0	0	1	1	1	1	2	2

$Ma(\delta_1 \circ \delta_0) - Mi(\delta_1 \circ \delta_0) = 4$, deci $\delta_1 \circ \delta_0$ este neomogenă.

Totuşi, dacă distribuţiile sunt omogene avem următoarele proprietăţi:

- $$Ma(\delta_1 \circ \delta_0) = Ma(\delta_1) * Ma(\delta_0)$$
  $$Mi(\delta_1 \circ \delta_0) = Mi(\delta_1) * Mi(\delta_0)$$
  $$Mi(\delta_1) * Mi(\delta_0) \leq |O^{\delta_1 \circ \delta_0}| \leq Ma(\delta_1) * Ma(\delta_0), \forall 0 \leq q < p,$$

- Distribuţia $D1 \circ D0$ este w-echilibrată cu

$$w = l.n.M * (m/M) + l.m.M * (n/N) + l.n.N * l.m.M,$$
$$\text{unde } l.a.b = \begin{cases} 1, \text{ dacă } a\%b > 0 \\ 0, \ altfel \end{cases}$$

- Dacă ambele distribuţii sunt perfecte atunci şi compunerea lor e perfectă.

Compunerea distribuţiilor are ca aplicaţii practice, de exemplu, programele care folosesc diferite tipuri de distribuţii de date. Introducând un parametru, aşa cum este m în distribuţia $ciclic \circ liniar$, este posibil să se elimine redistribuirile din timpul execuţiei şi să se asigure o echilibrare acceptabilă a încărcării de calcul, pentru fiecare parte individuală.

Distribuţii carteziene

Definiţia 6.6 (Distribuţie cartezianã) *O* distribuţie cartezianã *este definitã de produsul cartezian a douã distribuţii unidimensionale* $D0 = (\delta_0, \overline{m}, \overline{M})$ *şi* $D1 = (\delta_1, \overline{n}, \overline{N})$ *definit prin:*

$$D0 \times D1 = (\delta_0 \times \delta_1, \overline{m} \times \overline{n}, \overline{M} \times \overline{N}) \text{ unde } (\delta_0 \times \delta_1).(i,j) = (\delta_0.i, \delta_1.j).$$

Distribuţia produs cartezian foloseşte o pereche de numere ca identificator de proces. Pentru a obţine numărul de proces efectiv, este necesară o funcţie în plus $\beta : \overline{M} \times \overline{N} \rightarrow \overline{p}, p = M * N$. Funcţia β este o bijecţie.

Distribuţiile carteziene pentru matrice, pot fi obţinute distribuind liniile matricii independent de coloane. Cele mai comune distribuţii bidimensionale sunt carteziene. Mulţimea de elemente-dată atribuite de o distribuţie cartezianã unui proces poate fi definită astfel:

$$O^{\delta_0 \times \delta_1}.(s,t) = O^{\delta_0}.s \times O^{\delta_1}.t, \text{ unde } 0 \leq s < M, 0 \leq t < N.$$

În ceea ce priveşte echilibrarea distribuţiilor carteziene, se obţin rezultate similare cu cele de la compunerea distribuţiilor. De exemplu, dacă considerăm două distribuţii omogene $D0$ şi $D1$ avem

$$Mi(\delta_1) * Mi(\delta_0) \leq |O^{\delta_0 \times \delta_1}| \leq Ma(\delta_1) * Ma(\delta_0).$$

Cele mai folosite distribuţii carteziene sunt:

$$liniar^2 = (\delta_{liniar}, \overline{m}, \overline{M}) \times (\delta_{liniar}, \overline{n}, \overline{N}), \text{ cu } M = N$$
$$linie = liniar^2, \text{ cu } N = 1$$
$$coloana = liniar^2, \text{ cu } M = 1$$
$$ciclic^2 = (\delta_{ciclic}, \overline{m}, \overline{M}) \times (\delta_{ciclic}, \overline{n}, \overline{N}), \text{ cu } M = N$$
$$ciclic - linie = ciclic^2, \text{ cu } N = 1$$
$$ciclic - coloana = ciclic^2, \text{ cu } M = 1$$

Distribuţia $liniar^2$ cu $M = N$ este numită *bloc*, iar distribuţia $ciclic^2$ mai este numită şi *grid*.

Numărarea comunicaţiilor

Un aspect important al distribuţiilor este impactul lor asupra numărului de comunicaţii. Este important să distribuim datele astfel încât să minimizăm numărul de comunicaţii necesare, dar şi astfel încât cât mai multe comunicaţii să poată avea loc în paralel. Dacă se dă o postcondiţie csi o distribuţie a datelor se poate evalua numărul total de comunicaţii necesare. Postcondiţia program se împarte în p postcondiţii locale în conformitate cu distribuţia aleasă. Dacă presupunem că fiecare dată e atribuită unui singur proces, atunci numărul total de postcondiţii care fac referinţă la o anumită dată este o măsură a numărului de comunicaţii a acelei date. Totuşi este posibil ca anumite subexpresii, conţinând anumite date, să apară în postcondiţii diferite; un proces poate calcula acea subexpresie şi să o transmită celorlalte. Datorită faptului că în general se foloseşte rafinarea în paşi succesivi, acest caz este destul de rar. Dacă excludem acest caz, putem obţine o evaluare a impactului diferitelor distribuţii asupra complexităţii programului, înainte de elaborarea acestuia. Notăm cu $NAp.e$ = numărul de postcondiţii locale în care apare expresia e. Rezultă că numărul total de comunicaţii va fi egal cu:

$$NCom = (\sum e :: NAp.e - R.e), \text{ unde}$$
$$R.e = \begin{cases} 1, \text{ dacă } e \text{ apare în postcondiţia procesului care o conţine,} \\ 0, \text{ altfel.} \end{cases}$$

Valoarea $NCom$ este determinată de modul în care postcondiţia globală este împărţită în postcondiţii locale şi de distribuţia folosită. Complexitatea comunicării este mărginită inferior de $(NCom + p - 1)/p$, dacă un proces poate executa doar o comunicaţie la un moment dat. În general, $R.e$ este egal cu 1 [111], dar am definit aici valoarea $R.e$ pentru a asigura o definiţie generală, existând totuşi exemple pentru care $R.e = 0$.

Exemplul 6.13 Se dau două matrice A şi B de dimensiune $m \times o$ şi $o \times n$, $(m, n, o \in \mathbb{N}^*)$. Problema calculării produsului $C = A \times B$, implică postcondiţia $R: \ C = A \times B$. Folosim o distribuţie carteziană $D0 \times D1, D0 = (\delta_0, \overline{m}, \overline{M}), D1 = (\delta_1, \overline{n}, \overline{N})$, pentru matricea C pe $p = M \times N$ procese.

Considerăm şi distribuţiile:

$$D2 = (\delta_2, \overline{o}, \overline{N}), \quad \delta_2 : \overline{o} \to \overline{N},$$
$$D3 = (\delta_3, \overline{o}, \overline{M}), \quad \delta_3 : \overline{o} \to \overline{M},$$

şi presupunem că $M < o, N < o, M < m, N < n$.

Matricea A se distribuie folosind distribuţia $D0 \times D2$, iar pentru matricea B se foloseşte distribuţia $D3 \times D1$.

Fiecare proces se identifică cu perechea $(s, t), 0 \leq s < M \wedge 0 \leq t < N$. Postcondiţia locală $R.s.t$ a procesului (s, t) este

$$R.s.t : (i, j : 0 \leq i < m \wedge 0 \leq j < n \wedge \delta_0.i = s \wedge \delta_1.j = t : C(i, j) = (\sum k : 0 \leq k < o : A(i, k) * B(k, j)))$$

Se observă că:

$$(\forall s, t : 0 \leq s < M \wedge 0 \leq t < N : R.s.t) \Rightarrow R$$

Pentru a evalua numărul total de comunicaţii calculăm $NAp.A(i,k)$ şi $NAp.B(k,j)$.

$$\begin{array}{l} \quad NAp.A(i,k) \\ = \{\text{definiţie}\} \\ \quad |\{\forall s, t : 0 \leq s < M \wedge 0 \leq t < N \wedge \delta_0.i = s \wedge (\exists j :: \delta_1.j = t) : (s,t)\}| \end{array}$$

Cerem ca δ_1 să fie surjectivă, ceea ce înseamnă că toate procesele sunt folosite. Prin urmare:

$$\begin{array}{l} \quad NAp.A(i,k) \\ = \{\text{surjectivitatea distribuţiei } \delta_1\} \\ \quad |\{\forall s, t : 0 \leq s < M \wedge 0 \leq t < N \wedge \delta_0.i = s \wedge \ true : (s,t)\}| \\ = \{\text{calcul}\} \\ \quad N * |\{\forall s : 0 \leq s < M \wedge \delta_0.i = s : s\}| \\ = \{\text{un element data se distribuie doar unui proces}\} \\ \quad N \end{array}$$

Similar, se obţine $NAp.B(k,j) = M$, dacă δ_0 este surjectivă.

$$\begin{array}{l} \quad NCom \\ = \{\text{definiţia } NCom\} \\ \quad (\sum i, k : 0 \leq i < m \wedge 0 \leq k < o : NAp.A(i,k) - 1) + \\ \quad (\sum k, j : 0 \leq k < o \wedge 0 \leq j < n : NAp.B(k,j) - 1) \\ = \{\text{calcul}\} \\ \quad mo(N-1) + on(M-1) \\ = \\ \quad o(m(N-1) + n(M-1)) \end{array}$$

Deci, în acest caz, numărul total de comunicaţii nu depinde de distribuţia aleasă.

Se observă că dacă $M = N = P = 1$ (program secvenţial) atunci $NCom = 0$.

Deoarece m, n, o şi p sunt fixe determinăm M şi N astfel încât $NCom$ să fie minim. M şi N sunt întregi şi aparţin hiperbolei $p = M * N$. Avem relaţia:

$$\frac{NCom}{n * o} = \frac{m}{n}(N-1) + (M-1)$$

Minimizarea lui $NCom$ depinde de m/n; dacă $m = n$ atunci dacă p e pătrat perfect, cea mai bună alegere este $M = N, p = M^2$.

Acest rezultat confirmă rezultatele analizelor referitoare la înmulţirea matricelor [62], mai exact avantajul folosirii distribuţiilor *grid* sau *bloc*, în comparaţie cu distribuţiile *linie* sau *coloana*.

Exemplul 6.14 Pentru aceeaşi problemă se consideră matricele A şi B distribuite diferit. În această variantă pentru distribuirea matricelor folosim următoarele distribuţii:

$$\begin{array}{ll} D0 = (\delta_0, \overline{m}, \overline{M}), & \delta_0 : \overline{m} \to \overline{M}, \\ D2 = (\delta_2, \overline{o}, \overline{N}), & \delta_2 : \overline{o} \to \overline{N}, \\ D3 = (\delta_3, \overline{n}, \overline{M}), & \delta_3 : \overline{n} \to \overline{M}, \\ D1 = (\delta_1, \overline{n}, \overline{N}), & \delta_1 : \overline{n} \to \overline{N}. \end{array}$$

Pentru matricea A se foloseşte distribuţia $D0 \times D2$, iar pentru matricea C distribuţia $D0 \times D1$. Distribuim *transpusa matricei* B folosind distribuţia $D3 \times D2$ ($M < m, M < n, N < n, N < o$).

Postcondiţia locală este aceeaşi:

$$\begin{array}{l} R.s.t : (\forall i, j : 0 \le i < m \wedge \delta_0.i = s \wedge 0 \le j < n \wedge \delta_1.j = t : \\ \qquad C(i,j) = (\sum k : 0 \le k < o : A(i,k) * B(k,j))) \end{array}$$

Putem rescrie postcondiţia locală astfel:

$$\begin{array}{ll} & (\forall i, j : 0 \le i < m \wedge \delta_0.i = s \wedge 0 \le j < n \wedge \delta_1.j = t : \\ & \qquad C(i,j) = (\sum k : 0 \le k < o : A(i,k) * B(k,j))) \\ = & \{\text{calcul}\} \\ & (\forall i, j : 0 \le i < m \wedge \delta_0.i = s \wedge 0 \le j < n \wedge \delta_1.j = t : \\ & \qquad C(i,j) = (\sum v : 0 \le v < N : (\sum k : 0 \le k < o \wedge \delta_2.k = v : A(i,k) * B(k,j)))) \\ = & \{\text{notăm } w(i,j,v) = (\sum k : 0 \le k < o \wedge \delta_2.k = v : A(i,k) * B(k,j))\} \\ & (\forall i, j : 0 \le i < m \wedge \delta_0.i = s \wedge 0 \le j < n \wedge \delta_1.j = t : \\ & \qquad C(i,j) = (\sum v : 0 \le v < N : w(i,j,v))) \end{array}$$

Prin urmare, programul va conţine două etape: prima în care se vor calcula valorile $w.i.j.v$, iar a doua în care se vor însuma aceste valori. Evaluăm $NCom$ pentru fiecare etapă.

Pentru prima etapă, postcondiţia este:

$$\begin{array}{l} R0.s.t : (\forall i, j : 0 \le i < m \wedge \delta_0.i = s \wedge 0 \le j < n : \\ \qquad w(i,j,t) = (\sum k : 0 \le k < o \wedge \delta_2.k = t : A(i,k) * B(k,j))) \end{array}$$

$\forall s, t : 0 \le s < M, 0 \le t < N$. Elementul $A(i,k)$ apare într-o postcondiţie, $NAp.A(i,k) = 1$ şi acea postcondiţie este locală procesului care îl conţine $R.A(i,k) = 1$. Elementul $B(k,j)$ apare în M postcondiţii $NAp.B(k,j) = M$ şi $R.B(k,j) = 1$.

$$\begin{array}{ll} & NAp.B(k,j) \\ = & \{\text{definiţie}\} \\ & |\{\forall s, t : 0 \le s < M \wedge 0 \le t < N \wedge \delta_2.k = t \wedge (\exists i :: \delta_0.i = s) : (s,t)\}| \\ = & \{\delta_0 \text{ e surjectivă}\} \\ & M \end{array}$$

Rezultă că $NCom = on(M-1)$.

Pentru a doua etapă, postcondiţia este:

$$R1.s.t : (\forall i, j : 0 \leq i < m \wedge \delta_0.i = s \wedge 0 \leq j < n \wedge \delta_1.j = t : \\ C(i,j) = (\sum v : 0 \leq v < N : w(i,j,v))$$

Numărul de apariţii al datei $w(i,j,v)$ este 1.

Pentru a calcula $NCom$ corespunzător celei de-a doua etape, trebuie să calculăm mai întâi $R.w(i,j,v)$:

$$R.w(i,j,v) = \begin{cases} 1, \text{ dacă } v = \delta 1.j \\ 0, \text{ dacă } v \neq \delta 1.j \end{cases}$$

Deci numărul total de comunicaţii pentru etapa a doua este $mn(N-1)$, iar pentru cele două etape este

$$NCom = on(M-1) + mn(N-1).$$

Aşadar, cele două variante sunt aproximativ echivalente din punctul de vedere al numărului total de comunicaţii (depinde de valorile concrete pentru m, o, n). Ţinând cont că pentru cea de-a doua variantă, complexitatea computaţională este mai mare, datorită însumării sumelor parţiale, rezultă că prima variantă este mai convenabilă, dacă $m = n = o$.

6.2.2 Distribuţii multivoce

În cazul în care numărul elementelor-dată n este mai mic decât numărul de procese, un element-dată ar trebui atribuit mai multor procese. Deasemenea, în cazul în care un element-dată apare în mai multe calcule, atribuirea lui mai multor procese poate conduce la algoritmi mai eficienţi [136].

Definiţia 6.7 (Distribuţie multivocă) *O* distribuţie multivocă *pentru n date de intrare şi p procese este definită de o aplicaţie multivocă $\theta : \overline{n} \multimap \overline{p}$; $\theta.i$ reprezintă mulţimea proceselor care conţin elementul-dată cu indicele i.*

Caracteristicile unei astfel de distribuţii sunt:

- numărul de procese care conţin acelaşi element-dată,
- distribuţia datelor pe procese.

Analog distribuţiilor simple, se pot defini distribuţii multivoce $\overline{liniar}$ şi $\overline{ciclic}$, în cazul în care $p > n$:

$$\theta^{\overline{liniar}}.i = \{\forall k : 0 \leq k < p/n : i(p/n) + k\}, \text{ dacă } n \mid p \\ \theta^{\overline{ciclic}}.i = \{\forall k : 0 \leq k < p/n : kn + i\}, \text{ dacă } n \mid p.$$

Proprietăţile distribuţiilor simple, legate de încărcarea de calcul a proceselor, rămân valabile şi în cazul distribuţiilor multivoce, cu excepţia faptului că mulţimile $O.q = \{\forall i : 0 \leq i < n \wedge q \in \theta.i : i\}$ nu mai formează o partiţie a mulţimii $\overline{n}$.

O distribuţie multivocă cartezianǎ se poate obţine analog, prin produsul cartezian al mulţimilor imagine (de procese). Fie două distribuţii multivoce:

$$\theta_0 : \overline{m} \multimap \overline{M}, M > m$$
$$\theta_1 : \overline{n} \multimap \overline{N}, N > n$$

Produsul lor cartezian este definit de:

$$\theta_0 \times \theta_1 : \overline{m} \times \overline{n} \multimap \overline{M} \times \overline{N},$$
$$(\theta_0 \times \theta_1).(i,j) = \{\forall s,t : 0 \leq s < M \wedge 0 \leq t < N \wedge s \in \theta_0.i \wedge t \in \theta_1.j : (s,t)\}$$

În multe cazuri este necesar să specificăm procesele care conţin o anumită dată, într-o anumită ordine. De exemplu, dacă dorim să specificăm un proces de comunicaţie de tip broadcast al unei date e, este necesar să specificăm primul proces din mulţimea de procese care conţine data e. Se poate defini o funcţie θInd, care să permită accesarea proceselor dintr-o mulţime $\theta.i$. De exemplu, pentru distribuţiile $\overline{liniar}$ şi $\overline{ciclic}$ definim funcţiile:

$$\theta Ind^{\overline{liniar}}.i.k = i(p/n) + k,$$
$$\theta Ind^{\overline{ciclic}}.i.k = kn + i,$$

unde $i \in \overline{n}$ şi $k \in \overline{|\theta.i|}$.

Exemplul 6.15 (Broadcast) Considerăm p procese şi un tablou $x(i : 0 \leq i < n) : array\ of\ int, n < p$ distribuit proceselor pe baza distribuţiei multivoce θ. Folosind funcţiile ΘInd, o operaţie broadcast a elementului-dată $x(i)$, poate fi definit astfel:

$C.q ::$
$|[a : int;$
$\{a = x(i) \vee \neg(q \in \theta.i)\}$
if $(q = \Theta Ind.i.0) \rightarrow$
par $u : 0 \leq u < p \wedge \neg(u \in \theta.i) :$
$u!a$
rap
$[]\ \neg(q \in \theta.i) \rightarrow$
$\Theta Ind.i.0?a$
fi
$\{a = x(i)\}$
$]|$

Pentru a exprima mai uşor mulţimea de procese care conţine o anumită dată de intrare, distribuţiile multivoce pot fi definite folosind distribuţii simple, care sunt într-un anumit mod multiplicate.

Numărarea comunicaţiilor

Numărul total de comunicaţii necesare într-un program paralel, cu distribuţie multivocă a datelor, poate fi evaluat apriori, analog cu cazul distribuţiilor simple. Fie distribuţia multivocă $\theta_0 : \overline{n} \multimap \overline{p}$, atunci

$$NCom = \left(\sum e :: NAp.e - R.e \right).$$

De această dată mărimea $R.e$ se defineşte astfel:

$$R.e = |\{\forall q : 0 \leq q < p \wedge q \in \theta.e : A.q.e\}|$$

unde

$$A.q.e = \begin{cases} 1, \text{ dacă postcondiţia locală lui } q \text{ referă data } e \\ 0, \text{ altfel.} \end{cases}$$

În general, programele paralele care folosesc distribuţii multivoce, calculează rezultate parţiale care sunt apoi combinate. De aceea, estimarea numărului de comunicaţii este indicat să se realizeze în funcţie de aceste etape. Un exemplu este dat în continuare.

Exemplul 6.16 Considerăm din nou înmulţirea a două matrice A şi B de dimensiune $m \times o$ şi $o \times n$. Problema satisface postcondiţia

$$R : \ C = A \times B.$$

Dacă datele de intrare reprezintă o matrice $m \times n$, numărul de procese p poate fi factorizat astfel încât $p = M * N * Q$, unde $M \leq m$ şi $N \leq n$. O distribuţie carteziană $D0 \times D1$ va fi multiplicată de Q ori. Se face, de fapt, doar o renumerotare a proceselor.

Programul paralel conţine două etape. În prima, toate procesele sunt implicate în calcularea unor sume parţiale, iar în a doua sunt însumate aceste rezultate parţiale.

Considerăm $p = M * N * Q$ şi $p \leq m * n * o$. Matricea A este distribuită pe matricea de procese $M \times Q$ şi multiplicată pe direcţia N. Matricea B este distribuită pe matricea de procese $Q \times N$ şi multiplicată pe direcţia M. Matricea C este distribuită pe matricea de procese $M \times N$ şi multiplicată pe direcţia Q.

Fiecare proces este identificat printr-un triplet $(s, t, r), 0 \leq s < M, 0 \leq t < N, 0 \leq r < Q$. Procesul identificat prin (s, t, r) cu $0 \leq s < M \wedge 0 \leq t < N \wedge 0 \leq r < Q$ conţine următoarele date:

$$\begin{array}{l} A(i,k), \text{ cu } 0 \leq i < m \wedge 0 \leq k < o \wedge \delta_0.i = s \wedge \delta_2.k = r, \\ B(k,j), \text{ cu } 0 \leq k < o \wedge 0 \leq j < n \wedge \delta_2.k = r \wedge \delta_1.j = t, \\ C(i,j), \text{ cu } 0 \leq i < m \wedge 0 \leq j < n \wedge \delta_0.i = s \wedge \delta_1.j = t. \end{array}$$

Postcondiţia parametrizată $R.s.t.r$ a procesului (s, t, r) este:

$$\begin{array}{rcl} R.s.t.r & : & r \neq 0 \vee (\forall i, j : 0 \leq i < z \wedge 0 \leq j < n \wedge \delta_0.i = s \wedge \delta_1.j = t \\ & & : C(i,j) = (\sum rr : 0 \leq rr < Q : sum.i.j.rr)), \end{array}$$

unde $sum.i.j.rr = (\sum k : 0 \leq k < o \wedge \delta_2.k = rr : A(i,k) * B(k,j))$. Calculul valorilor $sum.i.j.rr$ se realizează pe baza postcondiţiei parametrizate:

$$\begin{aligned} R0.s.t.r \quad : \quad & (\forall i,j : 0 \leq i < m \wedge 0 \leq j < n \wedge \delta 0.i = s \wedge \delta 1.j = t \\ & : w(i,j,r) = sum.i.j.r. \end{aligned}$$

Se observă că:

$$(\forall s,t : 0 \leq s < M \wedge 0 \leq t < N : R.s.t.0) \Rightarrow R.$$

Pentru prima etapă numărul de comunicaţii este determinat de $NAp.A(i,k)$ şi $NAp.B(k,j)$.

$$\begin{aligned} & NAp.A(i,k) \\ = & \{\text{definiţie}\} \\ & |\{\forall s,t,r : 0 \leq s < M \wedge 0 \leq t < N \wedge 0 \leq r < Q \wedge \delta 0.i = s \wedge \delta 2.k = r : (s,t,r)\}| \\ = & \{\text{calcul}\} \\ & N, \end{aligned}$$

şi analog $NAp.B(k,j) = M$. Deoarece $R.A(i,k) = N$ şi $R.B(k,j) = M$, prin calcul se ajunge la $NCom = 0$.

Pentru a doua etapă:

$$\begin{aligned} & NAp.(w(i,j,r)) \\ = & \{\text{definiţie}\} \\ & |\{\forall s,t : 0 \leq s < M \wedge 0 \leq t < N \wedge \delta 0.i = s \wedge \delta 1.j = t : (s,t)\}| \\ = & \{\text{calcul}\} \\ & 1, \end{aligned}$$

iar

$$R.w(i,j,r) = \begin{cases} 0, & r \neq 0 \\ 1, & r = 0 \end{cases}$$

deci $NCom = mn(Q-1)$. Cu cât Q este ales mai mic, cu atât numărul comunicaţiilor scade, dar calculul computaţional pe fiecare proces creşte ($p = M * N * Q$).

Se poate observa că $M * N$ comunicaţii se pot executa în paralel. De aceea, se poate considera că complexitatea comunicaţională depinde de $\frac{mn}{MN}(Q-1)$. Dacă se consideră un algoritm bazat pe dublare recursivă, pentru însumarea sumelor parţiale, factorul $Q-1$ se înlocuieşte cu $\log_2 Q$.

Modul de distribuţie al datelor (funcţiile $\delta 1, \delta 2$ şi $\delta 3$) nu influenţează numărul de comunicaţii.

Pentru înmulţirea matrice–matrice, această idee de multiplicare a datelor nu este nouă. Dar, folosirea acestor noi tipuri de distribuţii creşte flexibilitatea, putându-se dezvolta formal un algoritm general care nu depinde de valorile concrete m, n, o şi p.

6.3 Aplicaţii

6.3.1 Operaţii prefix

Deseori în programele paralele trebuie calculată o sumă globală sau un maxim global, sau calcularea tuturor sumelor parţiale. Este de fapt problema calculării prefixului paralel, analizat şi în Secţiunea 4.10. Vom analiza aici o dezvoltare formală din specificaţii, bazată pe metoda descrisă în acest capitol, pentru prefixul paralel.

Considerăm un tablou f cu $p = 2^k (k \geq 0)$ elemente, care este distribuit prin atribuirea elementului $f(q)$ procesului q (distribuţie identică). Notăm $M.a.b = (\odot i : a \leq i < b : f(i))$, unde $\odot$ poate fi orice operaţie asociativă (de exemplu: $\sum, max, min \ldots$).

Specificaţia pentru un program paralel care realizează operaţia prefix pentru tabloul f este următoarea:

$|[k, p : int;$
$\quad f(i : 0 \leq i < p) : array\ of\ int;$
$\quad \{0 \leq k \wedge p = 2^k\}$
$\quad$ **par** $q : 0 \leq q < p :$
$\quad\quad |[m : int;$
$\quad\quad\quad S.q$
$\quad\quad\quad \{R.q : m = M.0.(q+1)\}$
$\quad\quad]|$
$\quad$ **rap**
$]|$

Ca variantă, programul poate calcula valorile parţiale finale $M.q.p$ (postfix).

Începem derivarea prin obţinerea postcondiţiei globale din postcondiţiile locale:

$$R' : (\forall q : 0 \leq q < p : m_q = M.0.(q+1)).$$

Notaţia m_q referă variabila m a procesului q.

Variabila ascunsă $k (p = 2^k)$ sugerează folosirea inducţiei. Prin înlocuirea variabilei k cu t obţinem un invariant global P' :

$$P' : (\forall q : 0 \leq q < 2^t : m_q = M.0.(q-1)) \wedge 0 \leq t \leq k.$$

Presupunem că t este global pentru toate procesele.

Se observă că P' este satisfăcut iniţial dacă se iniţializează t cu 0 şi în procesul 0 se iniţializează m cu $f(0)$. Progresul se realizează prin incrementarea lui t, care poate fi privit ca un ceas global; $P' \wedge t = k \Rightarrow R'$.

De la P' se ajunge la invarianţii parametrizaţi $P.q$:

$$\begin{array}{ll} P.q : & P0.q \wedge P1.q \\ P0.q : & 0 \leq t \leq k \\ P1.q : & 0 \leq q < 2^t \Rightarrow m = M.0.(q+1)). \end{array}$$

Deci din postcondiţia locală , trecând prin postcondiţia globală, am obţinut invariantul local $P.q$. Se poate obţine deasemenea, invariantul local şi direct din postcondiţia locală, introducând variabila locală t.

Alegerea modului de abordare se face în funcţie de problema concretă. În general, obţinerea unui invariant local se face considerând de la început o anumită distribuţie a datelor.

Pentru a deriva programul, considerăm $P1.q(t := t + 1)$

$$\begin{array}{ll} & P1.q(t := t + 1) \\ \equiv & \{\text{definiţia lui } P1.q(t := t + 1)\} \\ & 0 \leq q < 2^{t+1} \Rightarrow m = M.0.(q + 1) \\ \equiv & \{\text{împărţirea domeniului}\} \\ & P1.q \wedge 2^t \leq q < 2^{t+1} \Rightarrow m = M.0.(q + 1) \end{array}$$

Pentru un proces q, pentru care $2^t \leq q < 2^{t+1}$ trebuie calculat $M.0.(q + 1)$ şi avem următoarea derivare:

$$\begin{array}{ll} & m = M.0.(q + 1) \\ \equiv & \{ \text{ definiţia lui } M\} \\ & m = (\odot i : 0 \leq i < q + 1 : f(i)) \\ \equiv & \{2^t \leq q < 2^{t+1}, \text{împărţirea domeniului}\} \\ & m = (\odot i : 0 \leq i < q - 2^t + 1 : f(i)) \odot (\odot i : q - 2^t + 1 \leq i < q + 1 : f(i)) \\ \equiv & \{ \text{ definiţia lui } m\} \\ & m = M.0.(q - 2^t + 1) \odot M.(q - 2^t + 1).(q + 1) \end{array}$$

S-a folosit proprietatea $M.a.c = M.a.b \odot M.b.c$, pentru $0 \leq a < b < c \leq p$, care rezultă din asociativitatea operatorului $\odot$.

Valoarea $M.0.(q-2^t+1)$ este cunoscută în procesul $q-2^t$, din paşii anteriori; valoarea $M.(q-2^t+1).(q+1)$ e necunoscută. Aceasta sugerează să întărim invariantul cu un nou invariant $P2.q$ în care valoarea $M.(q-2^t+1).(q+1)$ e înregistrată în variabila m pentru toate procesele $q \geq 2^t$.

Deci

$$\begin{array}{ll} P.q: & P0.q \wedge P1.q \wedge P2.q \\ P2.q: & 2^t \leq q < p \Rightarrow m = M.(q - 2^t + 1).(q + 1). \end{array}$$

Dacă iniţializăm $m = M.(q-2^0+1).(q+1) = f(q)$ pentru $q > 0$ şi $m = M.0.1 = f(0)$, pentru $q = 0$, $P2$ e adevărat iniţial.

Progresul pentru $P2$:

$$\begin{array}{ll} & P2.q(t := t + 1) \\ \equiv & \{\text{definiţia lui } P2.q(t := t + 1)\} \\ & 2^{t+1} \leq q < p \Rightarrow m = M.(q - 2^{t+1} + 1).(q + 1) \\ \equiv & \{\text{împărţirea domeniului}\} \\ & 2^{t+1} \leq q < p \Rightarrow m = M.(q - 2^{t+1} + 1)(q - 2^t + 1) \odot M.(q - 2^t + 1).(q + 1) \end{array}$$

Termenul $M.(q - 2^{t+1} + 1)(q - 2^t + 1)$ este cunoscut în procesul $q - 2^t$.

Notăm cu $\overline{M}.q.t = M.(q - 2^t + 1).(q + 1)$ dacă $q \geq 2^t$ şi $\overline{M}.q.t = M.0.(q + 1)$ dacă $q < 2^t$.

În concluzie, vor fi trei tipuri de procese:

1. procesele q cu $q < 2^t$ au invarianţii $P1$ şi $P2$ adevăraţi,
2. procesele q cu $2^t \leq q < 2^{t+1}$ trebuie să restaureze $P1$, folosind valoarea din procesul $q - 2^t$,
3. procesele q cu $2^{t+1} \leq q < 2^k$ trebuie să restaureze $P2$ folosind valoarea din procesul $q - 2^t$.

Deci la iteraţia t, procesul $q, q \geq 2^t$ trebuie să se angajeze în recepţie de la procesul $q - 2^t$ şi în transmisie către procesul $q + 2^t$, dacă acesta există.

$S.q ::$
$|[t, aux : int;$
$\quad t := 0; m := f(q); \{m = M.q.(q + 1)\}$
$\quad \{P.q\}$
$\quad ; \mathbf{do}(t \neq k) \rightarrow$
$\quad\quad \mathbf{if}(q < 2^t) \rightarrow \{m = M.0.(q + 1)\}$
$\quad\quad\quad (q + 2^t)!m$
$\quad\quad [](2^t \leq q < 2^k - 2^t) \rightarrow \{m = \overline{M}.q.t\}$
$\quad\quad\quad \mathbf{par}\ (q - 2^t)?aux,\ (q + 2^t)!m\ \mathbf{rap}$
$\quad\quad\quad \{aux = \overline{M}.(q - 2^t).t\}$
$\quad\quad\quad ; m := aux \odot m$
$\quad\quad [](2^k - 2^t \leq q < 2^k) \rightarrow$
$\quad\quad\quad (q - 2^t)?aux$
$\quad\quad\quad \{aux = \overline{M}.(q - 2^t).t \wedge m = \overline{M}.q.t\}$
$\quad\quad\quad ; m := aux \odot m$
$\quad\quad \mathbf{fi}$
$\quad\quad ; t := t + 1$
$\quad \{P.q\}$
$\quad \mathbf{od}$
$]|$

Figura 6.1 reprezintă vizualizarea structurii comunicaţiei pentru un exemplu concret cu $p = 2^3$.

Complexitatea-timp este $T.p.p = O(\log_2 p)$, pentru că la fiecare iterare au loc cel mult două comunicaţii şi o operaţie $\odot$.

Se observă că s-a ajuns la o altă variantă de calculare a operaţiei prefix, diferită de cele prezentate în Secţiunea 4.10.

Se poate face implementare cu succes pe o reţea hipercub.

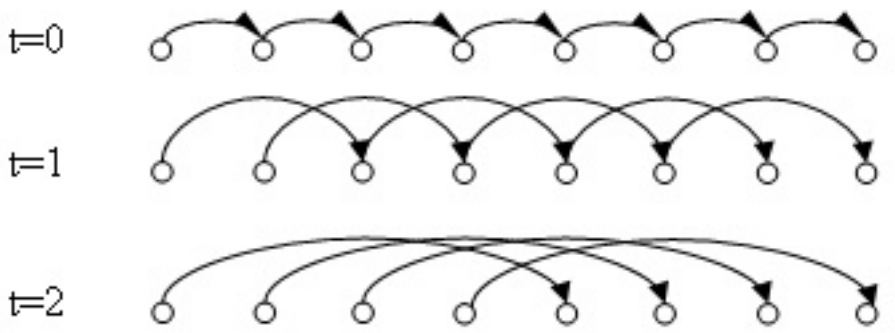

Figura 6.1: Calculul operaţiei prefix – structura comunicaţiei pentru $p = 8$, pe etape.

6.3.2 Înmulţire matriceală

În această secţiune prezentăm două variante de programe pentru înmulţirea matriceală. În secţiunile anterioare am analizat numărul total de comunicaţii în cele două cazuri: distribuţii simple şi distribuţii multivoce. Se evidenţiază aici importanţa distribuţiei datelor, aceasta determinând programul care se construieşte.

Datele sunt matricea A de ordin $(m \times o)$ şi matricea B de ordin $(o \times n)$, iar rezultatul $C = A \times B$.

Cazul distribuţiilor simple

Considerăm distribuţiile, corespunzătoare exemplului 6.13:

$$\begin{aligned} D0 &= (\delta_0, \overline{m}, \overline{M}), \quad \delta_0 : \overline{m} \to \overline{M}, \\ D2 &= (\delta_2, \overline{o}, \overline{N}), \quad \delta_2 : \overline{o} \to \overline{N}, \\ D3 &= (\delta_3, \overline{o}, \overline{M}), \quad \delta_3 : \overline{o} \to \overline{M}, \\ D1 &= (\delta_1, \overline{n}, \overline{N}), \quad \delta_1 : \overline{n} \to \overline{N}. \end{aligned}$$

Pentru matricea A folosim distribuţia $D0 \times D2$, pentru matricea B distribuţia $D3 \times D1$, iar pentru matricea C distribuţia $D0 \times D1$.

Postcondiţia locală fiecărui proces (s, t) este:

$$\begin{aligned} R.s.t : \ & (\forall i, j : 0 \le i < m \wedge 0 \le j < n \wedge \delta_0.i = s \wedge \delta_1.j = t : \\ & C(i,j) = (\sum k : 0 \le k < o : A(i,k) * B(k,j))). \end{aligned}$$

Pentru a putea obţine un invariant concretizăm alegerea distribuţiilor. Din analiza numărului total de comunicaţii, s-a observat că putem alege orice distribuţie. Alegem distribuţii *ciclic* şi considerăm că $N|o$:

$$\begin{aligned} \delta_0.i &= i\%M, \\ \delta_2.k &= k\%N, \\ \delta_3.k &= k\%M, \\ \delta_1.j &= j\%N. \end{aligned}$$

Invariantul local se obţine introducând o variabilă locală ol:

$$\begin{array}{l} P.s.t: \ P0.s.t \wedge P1.s.t \\ P0.s.t: \ 0 \leq ol < N \wedge ol\%N = 0 \\ P1.s.t: \ (\forall i,j : 0 \leq i < m \wedge 0 \leq j < n \wedge i\%M = s \wedge j\%N = t : \\ \quad C(i,j) = (\sum k : 0 \leq k < ol : A(i,k) * B(k,j))) \end{array}$$

Notăm cu $S.i.j.x.y = (\sum k : x \leq k < y : A(i,k) * B(k,j))$. Dacă iniţializăm $ol := 0$ şi $C(i,j) := 0$, atunci invariantul este adevărat iniţial. Progresul se realizează prin incrementarea lui ol cu N. Adică, la fiecare pas se vor adauga elementelor $C(i,j)$ termenii sumei cu indicii $k, ol \leq k < ol + N$:

$$\begin{array}{ll} & P1.s.t(ol := ol + N) \\ = & \\ & (\forall i,j : 0 \leq i < m \wedge 0 \leq j < n \wedge i\%M = s \wedge j\%N = t : \\ & \qquad C(i,j) = S.i.j.0.(ol + N)) \\ = & \\ & (\forall i,j : 0 \leq i < m \wedge 0 \leq j < n \wedge i\%M = s \wedge j\%N = t : \\ & \qquad C(i,j) = S.i.j.0.ol + S.i.j.ol.(ol + N). \end{array}$$

La fiecare pas trebuie restaurat invariantul $P1.s.t$.

Procesele parametrizate care se execută în paralel şi care constituie programul paralel sunt:

$S.s.t ::$
$||[ol : int;$
  $ol := 0;$
  for all $i,j : 0 \leq i < m \wedge 0 \leq j < n \wedge i\%M = s \wedge j\%N = t :$
    $C(i,j) := 0$
  lla rof
  $\{P.s.t\}$
  do $ol < o \ \rightarrow$
    $ReaceP1.s.t$
    $; ol := ol + N$
    $\{P.s.t\}$
  od
  $\{R.s.t\}$
$]||$

Pentru a reface invariantul $P1$ este necesar să se comunice datele necesare calculării sumelor $S.i.j.ol.(ol + N)$. Procesul (s,t) trebuie să primească de la procesele $(s,v), v \neq t$ valorile $A(i,k), ol \leq k < ol + N$, pentru toţi indicii i locali. Procesele $(k\%M, t), ol \leq k < ol + N$ transmit pe coloană valorile $B(k,j)$, pentru toţi indicii j locali. Pentru a reţine datele care se transmit se folosesc două tablouri locale x şi y. Deci procesul parametrizat $ReaceP1.s.t$ va repeta pentru fiecare ol cu $ol\%N = 0$ un proces de comunicaţie $C0.s.t$ şi un proces de calcul $S0.s.t$.

$RefaceP1.s.t ::$
$|[x(i,k : 0 \leq i < m \wedge 0 \leq k < N) : array\ of\ real;$
$\quad y(k,j : 0 \leq k < N \wedge 0 \leq j < n) : array\ of\ real;$
$\quad C0.s.t\{(\forall i,j,k : 0 \leq i < m \wedge i\%M = s \wedge 0 \leq j < n \wedge j\%N = t \wedge 0 \leq k < N :$
$\qquad x(i,k) = A(i,k+ol) \wedge y(k,j) = B(k+ol,j))\}$
$\quad ; S0.s.t\{(\forall i,j : 0 \leq i < m \wedge i\%M = s \wedge 0 \leq j < n \wedge j\%N = t :$
$\qquad C(i,j) = (\sum k : 0 \leq k < ol + N : A(i,k) * B(k,j)))\}$
$]|$

Pentru a evalua complexitatea comunicaţiilor ţinem cont de comunicaţiile care se pot executa în paralel şi considerăm că un broadcast al unei valori necesită o unitate de timp (reţea completă):

$$T_c.M.N.m.n.o = \frac{o}{N} * (N\frac{m}{M} + N\frac{n}{N}) \simeq o(m/M + n/N).$$

Procesul de calcul este definit de:

$S0.s.t ::$
$|[$ **for all** $i : 0 \leq i < m \wedge i\ \%\ M = s :$
$\quad$ **for all** $j : 0 \leq j < n \wedge j\ \%\ N = t :$
$\quad\quad$ **for all** $k : 0 \leq k < N :$
$\quad\quad\quad C(i,j) := C(i,j) + x(i,k) * y(k,j)$
$\quad\quad$ **lla rof**
$\quad$ **lla rof**
$\quad$ **lla rof**
$]|$

Ca urmare complexitatea de calcul este:

$$T_f.M.N.m.n.o = 2\frac{o}{N} * \frac{m}{M} * \frac{n}{N} * N \simeq 2(m * o * n)/(M * N)$$

Cazul distribuţiilor multivoce

Reluăm cazul tratat în Exemplul 6.16.

Considerăm toate distribuţiile carteziene ca fiind distribuţii *grid* care sunt definite în modul următor:

$A :$	$B :$	$C :$
$\delta_0 : \overline{m} \to \overline{M}, \delta_0.i = i\%M$	$\delta_2 : \overline{o} \to \overline{Q},$	$\delta_0 : \overline{m} \to \overline{M},$
$\delta_2 : \overline{o} \to \overline{Q}, \delta_2.k = k\%Q$	$\delta_1 : \overline{n} \to \overline{N}, \delta_1.j = j\%N$	$\delta_1 : \overline{n} \to \overline{N},$
$\delta_0 \times \delta_2,$	$\delta_2 \times \delta_1,$	$\delta_0 \times \delta_1.$

Postcondiţia locală procesului (s,t,r) este:

$$R.s.t.r: \ r \neq 0 \vee (\forall i,j : 0 \leq i < m \wedge 0 \leq j < n \wedge \delta_0.i = s \wedge \delta_1.j = t : \\ C(i,j) = (\sum k : 0 \leq k < o : A(i,k) * B(k,j)))$$

Ca urmare, programul paralel are următoarea structură:

$|[A(i,j : 0 \leq i < m \wedge 0 \leq j < o) : array\ of\ real;$
$B(i,j : 0 \leq i < o \wedge 0 \leq j < n) : array\ of\ real;$
$C(i,j : 0 \leq i < m \wedge 0 \leq j < n) : array\ of\ real;$
par $s,t,r : 0 \leq s < M \wedge 0 \leq t < N \wedge 0 \leq r < Q :$
$|[$
$S.s.t.r\{r \neq 0 \vee (\forall i,j : 0 \leq i < m \wedge 0 \leq j < n \wedge \delta_0.i = s \wedge \delta_1.j = t :$
$C(i,j) = (\sum k : 0 \leq k < o : A(i,k) * B(k,j)))\}$
$]|$
rap
$]|$

Postcondiţia poate fi rescrisă în modul următor:

$$R.s.t.r: \ r \neq 0 \vee (\forall i,j : 0 \leq i < m \wedge 0 \leq j < n \wedge \delta_0.i = s \wedge \delta_1.j = t : \\ : C(i,j) = (\sum rr : 0 \leq rr < Q : sum.i.j.rr)).$$

unde $sum.i.j.r = (\sum k : 0 \leq k < o \wedge \delta_2.k = r : A(i,k) * B(k,j))$.

În consecinţă avem un proces de calcul $S0.s.t.r$ care calculează sumele $sum.i.j.r$. Postcondiţia locală pentru acest proces este:

$$R0.s.t.r: \ (\forall i,j : 0 \leq i < m \wedge 0 \leq j < n \wedge \delta_0.i = s \wedge \delta_1.j = t : \\ w(i,j,r) = sum.i.j.r).$$

Procesul $S0.s.t.r$ conţine 3 instrucţiuni **for all** şi lucrează doar cu date locale.

Pentru a doua etapă se poate folosi câte un algoritm de tip arbore binar, pentru calculul valorii finale pentru fiecare $C(i,j)$. Dacă reţeaua de comunicaţie permite implementarea convenabilă a acestuia, atunci pentru această etapă se ajunge la o complexitate $O(\log_2 Q(mn)/(MN))$.

În acest caz complexitatea computaţională şi respectiv cea de comunicaţie totale sunt:

$$T_f.M.N.Q.m.n.o \simeq (m*n*o)/(M*N*Q) + \log_2 Q(m*n)/(M*N)$$
$$T_c.M.N.Q.m.n.o \simeq \log_2 Q(m*n)/(M*N).$$

Se pot analiza următoarele cazuri particulare:

1. $M = 1$, matricea A se distribuie doar pe coloane, dar e multiplicată pe fiecare coloană, iar matricea rezultat se distribuie pe linii.
2. $Q = 1$, matricea A se distribuie pe linii, iar matricea B pe coloane. Nu e necesară nici o transmisie.
3. $N = 1$, matricea B se distribuie pe linii şi matricea rezultat C pe coloane.

Alegerea valorilor M, N, Q, se face astfel încât să se minimizeze complexitatea, în funcţie de valorile concrete m, n, o.

6.3.3 Polinomul de interpolare Lagrange

În această secţiune se descrie construcţia unor algoritmi paraleli pentru calculul valorii într-un punct a polinomului de interpolare Lagrange, folosind metoda descrisă în acest capitol. Sunt construiţi doi algoritmi, pornind de la tipuri diferite de distribuţii: simple şi multivoce. Pentru analiza proceselor de comunicaţie considerăm reţele de interconectare complete.

Problema

Fie $[a,b] \subset \mathbb{R}, x(i) \in [a,b], 0 \leq i < m$, astfel încât $x(i) \neq x(j)$ pentru $i \neq j$ şi $f : [a,b] \to \mathbb{R}$.

Polinomul de interpolare Lagrange e definit de formula:

$$(L_{(m-1)}f)(x) = \left(\sum i : 0 \leq i < m : l_i(x) * f.x(i)\right),$$

unde $l_i, 0 \leq i < m$ sunt polinoamele fundamentale de interpolare Lagrange:

$$l_i(x) = \frac{(x-x(0))\ldots(x-x(i-1))(x-x(i+1))\ldots(x-x(m-1))}{(x(i)-x(0))\ldots(x(i)-x(i-1))(x(i)-x(i+1))\ldots(x(i)-x(m-1))}$$

$$= \frac{u(x)}{(x-x(i))} * \frac{1}{(x(i)-x(0))\ldots(x(i)-x(i-1))(x(i)-x(i+1))\ldots(x(i)-x(m-1))}$$

Postcondiţia globală pentru calculul valorii polinomului Lagrange într-un punct dat x este:

$$R: \ lx = (L_{m-1}f)(x).$$

Varianta 1 – distribuţii simple

Alegem distribuţia $\delta : \overline{m} \to \overline{p}$, pentru datele $x(i), f(i), 0 \leq i < m$. Valoarea x e distribuită tuturor proceselor (sau putem considera că x e distribuită procesului 0 şi apoi comunicată tuturor celorlalte procese, printr-un broadcast).

Folosind rafinarea în paşi succesivi, determinăm următoarele etape în dezvoltarea programului:

1. calcularea valorii $u(x)$,
2. calcularea valorii polinoamelor fundamentale $l_i(x)$,
3. calcularea valorii $(L_m f)(x)$.

Postcondiţiile locale pentru aceste etape sunt:

$$R0.q: \ ux = u(x) \wedge (\forall i : 0 \leq i < m \wedge \delta.i = q : xx(i) = x - x(i))$$
$$R1.q: \ (\forall i : 0 \leq i < m \wedge \delta.i = q : l(i) = l_i(x))$$
$$R2.q: \ lx = (L_{m-1}f)(x).$$

```
|[p, m : int; x, ux : real;
  l, xx, x(i : 0 ≤ i < m) : array of real;
  par q : 0 ≤ q < p :
    |[{Q.q : (∀i : i ∈ O.q : xx(i) = x − x(i)) ∧ ux = u.x)}
        S.q
      {R1.q : l(i) = l_i.x}
    |]
  rap
|]
```

```
S.q ::
|[ pr(i : 0 ≤ i < m) : array of real;
  a(i : 0 ≤ i < m) : array of real;
  for all i : i ∈ O.q :
    a(i) := x(i);
    pr(i) := 1
  lla rof
  {(∀i : i ∈ O.q : a(i) = x(i))}
  k := 0;   {P1.q(k := 0)}
  do (k ≠ m) →
    RefaceP1.1.q {P1.1.q(k := k + p)}
    ; k := k + p   {P1.q(k := k + p)}
  od
  for all i : i ∈ O.q :
    l(i) := (ux/xx(i))/pr(i)
  lla rof   {R1.q}
]|
```

```
RefaceP1.1.q ::
|[
  C0.q
{(∀i : 0 ≤ i < m ∧ i < k + p : a(i) = x(i))}
  ; S0.q
]|
```

```
C0.q ::
|[
  par u : 0 ≤ u < p ∧ u ≠ q :
    u!a(k + q) {transmisie}
  rap
  ; for all u : 0 ≤ u < p ∧ u ≠ q :
    u?a(k + u) {recepţie}
  lla rof
]|
```

```
S0.q ::
|[  for all i : i ∈ O.q :
      for all u : 0 ≤ u < p :
       if (i ≠ k + u) →
         pr(i) := pr(i) ∗ (a(i) − a(k + u))
       fi
      lla rof
    lla rof
]|
```

Figura 6.2: Interpolare Lagrange cu distribuţii simple – calcularea polinoamelor fundamentale $l_i(x)$.

Prima şi ultima reprezintă calcularea unui produs şi a unei sume şi deci se poate folosi un algoritm de tip arbore.

Postcondiţia pentru cea de-a doua etapă poate fi rescrisă în următoarea formă:

$$R1.q: \ (\forall i: 0 \leq i < m \wedge \delta.i = q : l(i) = \frac{ux}{x - x(i)} * \frac{1}{prod.i.m})$$

unde $prod.i.k = (\Pi j : 0 \leq j < k \wedge j \neq i : x(i) - x(j))$.

Folosind tehnica de numărare a comunicaţiilor putem decide că numărul comunicaţiilor nu este influenţat de funcţia de distribuţie a datelor ($NCom = m * (p - 1)$). Prin urmare, deoarece putem alege orice distribuţie, alegem distribuţia *ciclic*. Pentru simplificarea exprimării calculului presupunem că $m\%p = 0$. Vom folosi şi notaţia $O.q = \{\forall i : 0 \leq i < m \wedge i\%p = q : i\}$.

Pentru derivarea programului este necesar să definim invarianţi parametrizaţi. Introducem pentru aceasta, o variabilă $k : 0 \leq k < m \wedge k\%p = 0$. Aşadar invarianţii sunt:

$$P1.q : P1.0.q \wedge P1.1.q$$
$$P1.0.q : \ 0 \leq k < m \wedge k\%p = 0$$
$$P1.1.q : \ (\forall i : i \in O.q : pr(i) = prod.i.k)$$

Dacă iniţializăm variabila k cu 0 şi $pr(i)$ cu 1 atunci invarianţii sunt iniţial adevăraţi.

Progresul se realizează prin incrementarea variabilei k cu p:

$P1.1.q(k := k + p)$
$= \{\text{substituţie}\}$
$(\forall i : i \in O.q : pr(i) = prod.i.(k + p))$
$= \{\text{descompunerea domeniului}\}$
$(\forall i : i \in O.q : pr(i) = prod.i.k * (\prod j : k \leq j < k + p \wedge j \neq i : x(i) - x(j))$

Algoritmul va conţine compoziţia în paralel a mai multor procese $S.q, 0 \leq q < p$ (Figura 6.2).

Complexitatea

Procesul $RefaceP1.q$ este apelat de $\frac{m}{p}$ ori şi conţine un proces de comunicaţie şi un proces de calcul. În procesul de comunicaţie fiecare proces execută un broadcast prin care trimite o valoare celorlalte procese. Dacă considerăm că un broadcast se poate executa într-o unitate de timp, atunci $T_c.p.m = \frac{m}{p} * p = m$.

Deci, complexitatea celei de-a doua etape este:

$$\begin{aligned} T.p.m &= \tfrac{m}{p} * [p * \alpha + 2p * \tfrac{m}{p}] + 2\tfrac{m}{p} \\ &= m * \alpha + 2\tfrac{m^2}{p} + 2\tfrac{m}{p}. \end{aligned}$$

Varianta 2 – distribuţii multivoce

Fie $p = M * M$ şi identificăm fiecare proces printr-o pereche $(s, t), 0 \leq s, t < M$.

Vom folosi o distribuţie *ciclic*: $\delta : \overline{M} \to \overline{m}$ şi M permutări $\pi_t : \overline{M} \to \overline{M}, 0 \leq t < M$, definite de $\pi_t.i = (i + t)\%M$.

Distribuţia multivocă este definită de relaţia:

$$x(i) \in O.s.t \Leftrightarrow \delta.i = \pi_t.s \Leftrightarrow i\%M = (s + t)\%M$$

Pentru $m = 9$ şi $M = 3$ distribuţia datelor este arătată în Figura 6.3.

s\t	0	1	2
0	x_0, x_3, x_6	x_1, x_4, x_7	x_2, x_5, x_8
1	x_1, x_4, x_7	x_2, x_5, x_8	x_0, x_3, x_6
2	x_2, x_5, x_8	x_0, x_3, x_6	x_1, x_4, x_7

Figura 6.3: Distribuţia datelor pentru $m = 9$ şi $M = 3$.

Considerăm din nou cele trei etape, definite de următoarele postcondiţii locale:

$$\begin{aligned} &R0.s.t: \ (ux = u.x \wedge (\forall i : \quad i \in O.s.t : xx(i) = x - x(i))) \\ &R1.s.t: \ t \neq 0 \vee (\forall i : i \in O.s.t : l(i) = l_i(x)) \\ &R2.s.t: \ (lx = (L_{m-1}f).x) \end{aligned}$$

Pentru prima şi ultima etapă se poate folosi un algoritm de tip arbore, cu puţine modificări faţă de cel clasic. Diferenţa constă în faptul că se foloseşte o numerotare diferită a proceselor.

De exemplu, pentru calcularea lui ux putem folosi următoarea derivare:

$$\begin{aligned} &(\prod i : 0 \leq i < m : xx(i)) \\ &= \{\forall i, \exists!(s,t) \text{ astfel încât } i\%M = (s+t)\%M \wedge i\%M^2 = s * M + (s+t)\%M\} \\ &(\prod s, t : 0 \leq s, t < M : \\ &\quad (\prod i : 0 \leq i < m \wedge i\%M = (s+t)\%M \wedge i\%M^2 = s * M + (s+t)\%M : xx(i))) \\ &= \{O.s.t = (\forall i : 0 \leq i < m \wedge i\%M = (s+t)\%M : i)\} \\ &(\prod s, t : 0 \leq s, t < M : \\ &\quad (\prod i : i \in O.s.t \wedge i\%M^2 = s * M + (s+t)\%M : xx(i))) \end{aligned}$$

Postcondiţiile pentru produsele parţiale sunt:

$$R01.s.t : ux.s.t = (\prod i : i \in O.s.t \wedge i\%M^2 = s * M + (s+t)\%M : xx(i))$$

Deci, putem concluziona că

$$ux = (\prod i : 0 \leq i < m : xx(i)) = (\prod s, t : 0 \leq s, t < M : ux.s.t)$$

Invariantul este construit folosind o variabilă k, care este iniţializată la început cu $s*M+(s+t)\%M$; progresul se realizează prin incrementarea variabilei k cu M^2. Valoarea finală se obţine prin înmulţirea produselor parţiale:

$$R02.s.t : ux = (\prod s, t : 0 \leq s, t < M : ux.s.t).$$

Aşadar, complexitatea este aceeaşi cu cea de la varianta 1.

Această derivare se poate obţine şi pornind de la de definiţia funcţiei ΘInd corespunzătoare: $\Theta Ind.i.k = (k, (i-k)\%M)$; in acest caz, postcondiţiile parţiale corespunzătoare sunt:

$$R01.s.t : ux.s.t = (\prod i : 0 \leq i < m \wedge \Theta Ind.i.((i/M)\%M) = (s,t) : xx(i)),$$

care sunt echivalente cu cele scrise anterior.

În continuare vom detalia mai mult cea de-a doua etapă. Ca şi în cazul înmulţirii a două matrice, vom considera două subetape: una pentru calcule **par**ţiale şi una pentru combinarea acestor calcule **par**ţiale.

Fiecare linie $(s, .)$ calculează valorile $l(i), \forall i : 0 \leq i < m \wedge \delta.i = s$.

Postcondiţia $R1.s.t$ poate fi rescrisă în următoarea formă:

$$R1.s.t : t \neq 0 \vee (\forall i : i \in O.s.t : \quad l(i) = ux/xx(i) * 1/prod.i.m)$$

unde $prod.i.m = (\prod j : 0 \leq j < m \wedge i \neq j : (x(i) - x(j)))$.

Pentru a calcula produsele $prod.i.m$ putem să le împărţim în M subproduse. Fiecare subprodus corespunde mulţimii de elemente atribuite unui proces.

Deci, putem rescrie produse $prod.i.m$ astfel:

$$\begin{array}{l} prod.i.m \\ = \{\text{descompunerea domeniului}\} \\ \quad (\prod t : 0 \leq t < M : (\prod j : j \in O.s.t \wedge i \neq j : (x(i) - x(j)))) \\ = \{(j \in O.s.t \Leftrightarrow 0 \leq j < m \wedge j\%M = (s+t)\%M)\} \\ \quad (\prod t : 0 \leq t < M : (\prod j : 0 \leq j < m \wedge \delta.j = \pi_t.(\delta.i) \wedge i \neq j : (x(i) - x(j)))) \\ = \{parprod.i.t \stackrel{not}{=} (\prod j : 0 \leq j < m \wedge \delta.j = \pi_t.(\delta.i) \wedge i \neq j : (x(i) - x(j)))\} \\ \quad (\prod t : 0 \leq t < M : parprod.i.t) \end{array}$$

Valorile $prod.i.m$ se obţin în final folosind un algoritm de tip arbore pe fiecare linie.

Invarianţii pentru calcularea produselor parţiale sunt definiţi prin introducerea variabilei k, care va fi incrementată cu M:

$$\begin{array}{l} P1.s.t : P1.0.s.t \wedge P1.1.s.t \\ P1.0.s.t : \ 0 \leq k < m \wedge k\%M = 0 \\ P1.1.s.t : \ (\forall i : 0 \leq i < k \wedge \delta.i = s : ppr(i) = parprod.i.t). \end{array}$$

Procesele parametrizate pentru calcularea produselor parţiale $S.s.t, 0 \leq s, t < M$ sunt descrise în Figura 6.4.

Complexitatea

Pentru această variantă complexitatea celei de-a doua etape este:

$$\begin{array}{rl} T.p.m & = \frac{m}{M}[\alpha + 2\frac{m}{M} + \log_2 M(\alpha+1)] + 2\frac{m}{M} \\ & = 2\frac{m^2}{p} + \frac{m}{M}(\log_2 M + 2) + \alpha\frac{(1+\log_2 M)}{M}m. \end{array}$$

Complexitatea comunicaţiei este mai mică decât cea a primului algoritm.

$S.s.t ::$
$|[\ ppr(i : 0 \leq i < m) : array\ of\ real;$
$\quad a(i : 0 \leq i < m) : array\ of\ real;$
$\quad$ **for all** $i : i \in O.s.t :$
$\quad\quad a(i) := x(i);$
$\quad$ **lla rof**
$\quad \{(\forall i : i \in O.s.t : a(i) = x(i))\}$
$\quad k := 0; \quad \{P1.q(k := 0)\}$
$\quad$ **do** $(k \neq m) \rightarrow$
$\quad\quad RefaceP1.1.s.t\ \{P1.1.s.t(k := k + M)\}$
$\quad\quad ; k := k + M\ \{P1.s.t(k := k + M)\}$
$\quad$ **od**
$]|$

$RefaceP1.1.s.t ::$
$|[$
$\quad C0.s.t\ \{a(s + k) = x(s + k)\}$
$\quad ; S0.s.t$
$\quad \{ppr(s + k) = (\prod i : i \in O.s.t \wedge i \neq s + k : a(s + k) - a(i))\}$
$]|$

$C0.s.t ::$
$|[$
$\quad$ **if** $(t = 0) \rightarrow$
$\quad\quad$ **par** $v : 0 < v < M :$
$\quad\quad\quad (s, v)!a(k + s)$
$\quad\quad$ **rap**
$\quad [](t \neq 0) \rightarrow$
$\quad\quad (s, 0)?a(k + s)$
$\quad$ **fi**
$]|$

$S0.s.t ::$
$|[\quad ppr(s + k) := 1;$
$\quad$ **for all** $j : j \in O.s.t :$
$\quad\quad$ **if** $(j \neq s + k) \rightarrow$
$\quad\quad\quad ppr(s + k) := ppr(s + k) * (a(s + k) - a(j))$
$\quad\quad$ **fi**
$\quad$ **lla rof**
$]|$

Figura 6.4: Interpolare Lagrange cu distribuţii multivoce – procesele $S.s.t$.

Se pot trage următoarele concluzii:

- Ambele variante de algoritmi sunt construite pornind de la specificaţii, folosind reguli de derivare corecte.
- Tipuri diferite de distribuţii conduc la algoritmi diferiţi.
- Cel de-al doilea algoritm poate fi folosit în ambele cazuri: $p \leq m$, sau $p > m$ şi are complexitatea de comunicaţie mai mică.

  Dacă $p > m$, avantajul folosirii celui de-al doilea algoritm este evident.

  Dacă $p \leq m$, comparaţia dintre complexităţile celor două variante duce la următoarea concluzie:

  $$\alpha > 3.5 \text{ adevărat în multe cazuri}$$
  $$\Rightarrow T_{distributie-multivoca} < T_{distributie-simpla}, \forall M \geq 4.$$

  Dacă M este mai mare, atunci inegalitatea este adevărată şi pentru cazurile în care α este mai mic.
- Dacă se consideră implementări pe reţele de tip hipercub sau latice şi nu o reţea de interconectare ideală, atunci complexitatea variantei a doua este şi mai bună.

Sumar

Metoda de programare paralelă propusă, seamănă mult cu o metodă de derivare corectă din specificaţii pentru programare secvenţială. Se bazează pe reguli stricte care forţează structura unui program paralel:

- Un program paralel constă din p instanţe ale unui singur proces parametrizat S.
- S este mai departe rafinat, folosind tehnici standard ale programării secvenţiale, în secvenţe de programe secvenţiale ordinare şi procese de comunicaţie, fiecare fiind la rândul lui un proces parametrizat.
- Instanţele unui proces parametrizat de comunicare formează un nivel închis la comunicaţii (comunicaţiile au loc doar între instanţele aceluiaşi proces parametrizat).

Ca o consecinţă a acestei structuri putem considera un program paralel ca fiind descompus pe nivele. În nivelele de calcul, operaţiile sunt distribuite între cele p procese şi fiecare proces realizează calcule pe propriul set de date locale. În nivelele de comunicaţie, procesele interacţionează prin transmitere de mesaje. Descompunerea pe nivele facilitează demonstrarea corectitudinii. Fiecare nivel este construit folosind invarianţi parametrizaţi şi poate fi demonstrată corectitudinea lui aplicând reguli de demonstrare clasice şi specifice. Un nivel de comunicaţie are specificaţii separate, de obicei, cu o funcţionalitate simplă. Se pot face implementări alternative ale proceselor de comunicaţie, care se bazează pe diferite reţele de comunicaţie. Este de preferat să se lucreze cu nivele de

comunicaţie mici şi evitarea lor pe cât posibil. Nivelele de calcul este bine să fie cât mai mari şi bine echilibrate, pentru a reduce pe cât posibil timpii de aşteptare.

Eficienţa unui program paralel este mult determinată de distribuţia datelor folosite, care la rândul ei determină încărcarea de calcul şi numărul de comunicaţii.

Folosirea distribuţiilor multivoce este absolut necesară pentru cazul dezvoltării de programe paralele pentru sisteme cu granulaţie fină, când numărul de procesoare este foarte mare. Totuşi, şi în celelalte cazuri, mai ales datorită faptului că în general se construiesc mai multe procese decât numărul de procesoare efective (pentru ascunderea întârzierilor), folosirea distribuţiilor multivoce poate conduce la programe paralele generale eficiente.

Capitolul 7

Formalismul Bird-Meertens – BMF

Proprietăţile programării funcţionale fac ca aceasta să aducă avantaje importante pentru programarea paralelă, făcând posibilă dezvoltarea programelor prin transformări riguroase şi exploatând mecanisme de abstractizare a datelor şi a controlului fluxului de execuţie. Problema practică care se pune este de a găsi o modalitate de a folosi aceste proprietăţi pentru a asigura eficienţa şi predictibilitatea execuţiei programelor pe arhitecturi paralele.

Formalismul Bird-Meertens (BMF)[20] a fost iniţial creat pentru dezvoltarea programelor secvenţiale. În timp, BMF a devenit tot mai popular pentru construcţia programelor paralele. În BMF, funcţii de nivel înalt (funcţionale) captează conceptele generale ale programării paralele, într-un mod independent de arhitectură. Aceaste funcţionale pot fi compuse pentru a obţine algoritmi. Funcţionalele BMF au ca şi parametrii operatori elementari şi funcţii, astfel încât o expresie BMF reprezintă o clasă de programe asupra căreia se poate raţiona independent, sau ţinând cont de proprietăţile particulare ale funcţiilor concrete. Stilul acesta de programare se numeşte *generic* sau *bazat pe şabloane*(“skeletons”). În ultima perioadă, acest domeniu a fost investigat, dezvoltat şi adaptat programării paralele.

Prezentăm în acest capitol o abordare de dezvoltare a programelor paralele numită SAT (Stages and Transformations), specificată de S. Gorlatch [70], bazată pe BMF, care combină abstractizarea şi performanţa într-un mod sistematic. Această metodă are la bază:

- **Etape** care structurează specificaţia abstractă a programelor paralele folosind funcţionale BMF şi conceptul de schemă şi deci ascunde detaliile de programare de nivel jos.

- **Transformări** care captează procesul de construcţie al programului prin tranziţii care păstrează corectitudinea, fie între diferite specificaţii abstracte, fie între specificaţie şi implementări.

Abordarea SAT include algoritmi care lucrează cu structuri de date construite recursiv, cum sunt listele şi arborii. Paralelismul algoritmic este abstractizat prin scheme

de nivel înalt în contextul funcţional al BMF. Schema de bază care este folosită este *omeomorfismul*.

În continuare, este dată o prezentare succintă a notaţiei BMF, pentru structura de date de bază lista nevidă, cu operatorul de concatenare ($\diamond$) ca şi constructor. (Listele se notează printr-o înşiruire de elemente încadrate de paranteze drepte.) Pentru a nu încărca prezentarea, definirea funcţionalelor se va face informal, presupunându-se că expresiile funcţionalelor sunt corect tipizate.

Cea mai simplă şi în acelaşi timp cea care conţine cel mai înalt grad de paralelism este funcţionala *map*, care aplică funcţia unară f, fiecărui element al listei:

$$map(f).[x_1, x_2, \ldots, x_n] = [f.x_1, f.x_2, \ldots, f.x_n].$$

Calcularea lui f pe diferite elemente ale listei poate fi făcută independent, dacă sunt disponibile destule procesoare. Elementele listei pot fi la rândul lor liste şi f poate fi o funcţie complexă sau o compunere de funcţii.

O variantă a funcţionalei *map* este *dmap* (*distributed map*), care este o funcţională care aplică o listă $[h_1, h_2, \ldots, h_n]$ de n funcţii, pe o listă $[l_1, l_2, \ldots, l_n]$ de n liste de argumente:

$$dmap([h_1, h_2, \ldots, h_n]).[l_1, l_2, \ldots, l_n] = [map(h_1).l_1, map(h_1).l_1, \ldots, map(h_n).l_n]$$

Altă variantă de funcţională *map* este *omap*:

$$omap(\odot).[a_1, \ldots, a_n].[b_1, \ldots, b_n] = [a_1 \odot b_1, \ldots, a_n \odot b_n]$$

care aplică operatorul binar $\odot$ elementelor corespunzătoare din cele două liste argument.

În afara paralelismului lui *map*, BMF permite descrierea paralelismului de tip arbore. Acesta se exprimă prin funcţionala *red* (de la reducere) cu un operator binar asociativ $\oplus$:

$$red(\oplus).[x_1, x_2, \ldots, x_n] = x_1 \oplus x_2 \oplus \ldots \oplus x_n.$$

Reducerea poate fi calculată pe un arbore binar echilibrat cu operatorul $\oplus$ în noduri. Timpul calculului paralel depinde de adâncimea arborelui, care este $\log_2 n$, pentru o listă cu lungimea n.

Pe parcurs se vor introduce şi alte funcţionale. Expresiile BMF se pot obţine prin compunerea funcţionalelor BMF $(f \circ g).x$. Compunerea funcţionalelor este asociativă şi reprezintă execuţia secvenţială a celor două funcţionale.

Pentru expresiile condiţionale cum este `dacă` p `atunci` b `altfel` c, se foloseşte notaţia $p \rightarrow b; c$, cu urmatoarele proprietăţi:

$$p \rightarrow h; h = h \tag{7.1}$$

$$(p \rightarrow b; h) \circ t = p \circ t \rightarrow b \circ t; h \circ t \tag{7.2}$$

Exemplul 7.1 (**Rezolvare sistem liniar prin metoda Jacobi**) Pentru rezolvarea unui sistem liniar $A \times X = B$, unde A este o matrice de dimensiune $n \times n$ cu proprietatea $a_{ii} \neq 0, \forall i = 1, \ldots, n$ şi X, B sunt vectori de dimensiune n, prin metoda iterativă Jacobi,

se porneşte de la o aproximaţie iniţială $X^0 = (x_1^0, \ldots, x_n^0)$ şi apoi se evaluează recursiv aproximaţiile $X^1, \ldots, X^n, n > 1$. Un pas de aproximare presupune următoarele calcule:

$$x_i^m = (b_i - \sum_{j=1, j\neq i}^{n} a_{i,j} \cdot x_j^{m-1})/a_{i,i}, \quad i = 1, \ldots, n, m > 1.$$

Considerăm lista $x = [x_1^0, \ldots, x_n^0, 1]$ şi lista de liste $a = [[a_{1,1}, \ldots, a_{1,n}, -b_1], \ldots, [a_{n,1}, \ldots, a_{n,n}, -b_n]]$. Definim funcţia *Jacobi* care returnează o listă care conţine valorile $[x_1^1, \ldots, x_n^1]$ şi care reprezintă aplicarea unui pas Jacobi.

$Jacobi.a.x = dmap([ja_1.x, \ldots, ja_n.x]).a$
$ja_i.x.y = (-red(+).(prod_i.x.y))/\pi_i.y$
$prod_i.x.y = dmap([f_1.i, \ldots, f_n.i]).(lists.(omap(\cdot).x.y))$
$f_j.i.[z] = i \neq j \rightarrow z; 0, \quad j = 1, \ldots, n$
$lists.[z_1, \ldots, z_n] = [[z_1], \ldots, [z_n]]$

S-a folosit funcţia proiecţie π_i, care returnează a i-a componentă a unei liste.

Funcţia *lists* s-a introdus doar pentru a respecta definiţia funcţionalei *dmap* care are ca argument o listă de liste. În acest caz, fiecare sublistă conţine un singur element.

Prin urmare un pas al iteraţiei Jacobi este o compunere de funcţionale *map* şi *red* şi avem o descriere funcţională a unui program paralel.

Expresiile BMF – şi prin urmare şi programele specificate de ele – pot fi manipulate prin aplicarea unor reguli ale formalismului, care păstrează semantica.

Această descriere abstractă a programelor paralele se poate lega de o descriere care să includă şi performanţa programului pe o anumită arhitectură, folosind conceptul de *model de programare*.

Modelul formal tradiţional de paralelism, PRAM, permite descrierea şi compararea algoritmilor paraleli, făcând abstracţie de sincronicitate, localizarea datelor şi capacitatea de comunicare. Într-un model reţea, un program este văzut ca o colecţie de procese comunicante, fiecare cu o memorie locală. Folosind terminologia Occam sau CSP (Communicating Sequential Processes) [89], acesta poate fi numit model "PAR-SEQ" (PARallel composition of SEQuential processes). Acesta corespunde direct clasei SPMD şi exprimă bine eficienţa programelor, dar implică multe detalii de nivel jos.

Soluţia este de a folosi modelul PAR-SEQ doar pentru reprezentarea programului ţintă. Procesul de construcţie al programului exploatează modelul dual SEQ-PAR. Acest model îşi are originile în paralelismul de tip SIMD şi este extins la clasa programelor SPMD. Un program paralel în construcţie este privit ca o secvenţă de etape paralele. Fiecare etapă încapsulează paralelism de diferite tipuri şi implică potenţial toate procesoarele. Se foloseşte din nou conceptul de "nivel închis la comunicaţii": nu sunt permise comunicaţii între etape diferite. BMF permite specificaţia directă a unui program cu un singur fir de execuţie, care la rândul lui poate fi extins la un program SPMD, datorită închiderii la comunicaţie a etapelor.

Acest model simplifică semnificativ structura programelor: controlul global devine secvenţial şi părţile paralele ale programului devin mai mici şi mai uşor de înţeles.

Formalismul BMF este un mediu care permite transformări ale programelor. Un exemplu simplu de transformare este legea fuziunii *map*:

$$map(f \circ g) = map(f) \circ map(g) \tag{7.3}$$

Dacă compunerea secvenţială a celor doi paşi paraleli din partea dreaptă este implementată prin bariera de sincronizare, atunci partea stângă e mai eficientă şi deci preferabilă. Pentru a estima impactul unei anumite transformări asupra performanţei se poate folosi un calcul de cost.

7.1 Omeomorfisme pe liste

Principiul omeomorfismului este foarte cunoscut şi folosit în diferite ramuri ale matematicii; intuitiv, o funcţie omeomorfă conservă structura domeniului său în codomeniu.

Definiţia 7.1 *O funcţie h definită pe o mulţime de liste este un* omeomorfism *dacă şi numai dacă există un operator binar $\otimes$ astfel încât, oricare ar fi listele x şi y:*

$$h.(x \diamond y) = h.x \otimes h.y \tag{7.4}$$

Adică, valoarea unui omeomorfism pe o listă poate fi calculată prin aplicarea operatorului de combinare $\otimes$ asupra componentelor acelei liste. Datorită faptului că valorile $h.x$ şi $h.y$ se pot calcula independent, ele se pot calcula în paralel. Operatorul $\otimes$ este în mod necesar asociativ, datorită asociativităţii operatorului de concatenare.

Omeomorfismele exprimă foarte bine paradigma *divide&impera*.

Exemplul 7.2 (**Prefix**) Funcţia $prefix$ furnizează pentru un operator binar $\odot$ şi o listă, lista tuturor "sumelor prefix". De exemplu, pentru o listă cu 3 elemente:

$$prefix(\odot).[a, b, c] = [a, a \odot b, a \odot b \odot c].$$

Funcţia $prefix$ este un omeomorfism cu operatorul de combinare $\otimes$ precizat în continuare:

$$S_1 \otimes S_2 = S_1 \diamond (map((last.S_1)\odot).S_2)$$
$$prefix(\odot).(x \diamond y) = prefix(\odot).x \otimes prefix(\odot).y,$$

unde $last.S$ returnează ultimul element din S.
(Algoritmul corespunde algoritmului "upper-lower" prezentat în Capitolul 4, Secţiunea 4.10, dar în acest caz avem şi asigurarea corectitudinii.)

Teorema 7.1 *(***de caracterizare***) O funcţie h este un omeomorfism dacă şi numai dacă poate fi factorizată ca o compunere:*

$$h = red(\otimes) \circ map(f), \tag{7.5}$$

unde oricare ar fi un element a, $f.a$=$h.[a]$.

Fiecare omeomorfism este unic determinat de f şi $\otimes$. Astfel, toate omeomorfismele pot fi privite ca instanţe ale unei singure *şablon omeomorfism*. În consecinţă, aceast şablon poate fi calculat într-un mod uniform de un program cu două nivele, compus din funcţionalele BMF *map* şi *red*.

Pentru a folosi omeomorfismele în procesul de construcţie al programelor paralele, trebuie rezolvate următoarele probleme:

1. *Extragere:* verificarea unei funcţii dacă poate fi o instanţă a schemei omeomorfism, adică găsirea specificaţiei concrete pentru funcţia f şi pentru operatorul $\otimes$.

2. *Ajustare:* dacă o funcţie nu e omeomorfism, se încearcă includerea ei într-un n-uplu de funcţii care constituie un omeomorfism.

3. *Compunere:* găsirea unei modalităţi eficiente de a compune mai multe omeomorfisme ca nivele într-un program abstract mai mare.

4. *Implementare:* găsirea unei modalităţi de implementare eficiente, care ţine cont de proprietăţile funcţiilor concrete

7.1.1 Extragere

Pentru o funcţie h este necesar să se găsească funcţia f, care furnizează imaginea lui h pentru o listă cu un singur element şi operatorul de combinare $\otimes$. Funcţia f este în general uşor de găsit, dar găsirea operatorului $\otimes$ nu este în general trivială.

Metoda CS

Se consideră constructorii *cons* şi *snoc* tipici programării secvenţiale funcţionale. Constructorul $cons(\triangleright)$ adaugă un element la începutul unei liste, iar constructorul $snoc(\triangleleft)$ adaugă un element la sfârşitul unei liste.

Definiţia 7.2 *Funcţia h se numeşte* de stânga *(st) dacă şi numai dacă există un operator binar $\oplus$ astfel încât $h.(a \triangleright y) = a \oplus h.x$, oricare ar fi elementul a şi lista x. Dual, o funcţie h este* de dreapta *(dr) dacă şi numai dacă există un operator $\ominus$ astfel încât $h.(x \triangleleft a) = h.x \ominus a$.*

Datorită faptului că operatorii $\oplus, \ominus$ pot fi complecşi (pot conţine şi alternative) şi nu sunt neapărat asociativi, o funcţie poate fi numai *st* sau *dr* sau ambele.

Importanţa pentru o funcţie de a fi *st* şi *dr* este evidentiaţă de următoarea teoremă, care combină prin cele două implicaţii ale sale, două teoreme clasice în BMF (a doua şi a treia), ambele definite de Bird [20]. Prima implicaţie este simplu de demonstrat, iar a doua a fost demonstrată de Meertens şi prezentată sistematic de Gibbons[65].

Teorema 7.2 *O funcţie pe liste este un omeomorfism dacă şi numai dacă este şi (st) şi (dr).*

Prin această teoremă se asigură faptul că pentru o funcţie care este şi (st) şi (dr), existenţa operatorului $\otimes$ este asigurată, dar nu se dă o modalitate de obţinere a lui. Pentru aceasta se impun restricţii în plus pentru funcţiile (st) şi (dr).

Definiţia 7.3 *O funcţie h se numeşte* omeomorfă la stânga *(os) dacă şi numai dacă există $\otimes$ astfel încât oricare ar fi lista x şi elementul a: $h.(a \triangleright x) = h.[a] \otimes h.x$. Analog, pentru funcţii* omeomorfe la dreapta *(od)* $h.(x \triangleleft a) = h.x \otimes h.[a]$.

Introducem funcţia f astfel încât $f.a = h.([a])$. Pentru o funcţie dată f şi un operator $\otimes$, există un singur omeomorfism la stânga, (respectiv la dreapta), notat cu $os(f, \otimes)$ (respectiv $od(f, \otimes)$).

Evident, orice funcţie $os(od)$ este de asemenea şi $st(dr)$, dar nu şi invers. De exemplu, funcţia g:

$$g.[a] = |a|$$
$$g.(a \triangleright x) = if\ a \leq g.x\ \rightarrow\ |a + g.x|\ ;\ |a - g.x|$$

este de stânga, dar nu este omomorfism de stânga, deoarece $g.(a \triangleright x)$ nu poate fi exprimat doar cu ajutorul lui $|a|$ şi $g.x$.

Se observă că, deoarece operaţia $\otimes$ din (7.5) este în mod necesar asociativă, un $\otimes$-omeomorfism este complet determinat de acţiunea sa pe liste cu un singur element, deci putem scrie $om(f, \otimes)$ pentru unica funcţie $\otimes$-omeomorfism, pentru care $h.[a] = f.a$ oricare ar fi a.

Teorema 7.3 *Dacă o funcţie e omeomorfism atunci este şi (os) şi (od), cu acelaşi operator de combinare. Dacă funcţia este (os) sau este (od) şi operatorul de combinare este asociativ atunci funcţia este omeomorfă cu operatorul de combinare specificat de (os) sau de (od).*

O demonstraţie a acestei teoreme poate fi găsită în [73].

Teorema sugerează o modalitate de construcţie a operatorului: se construieşte o definiţie a funcţiei omeomorfă la stânga, pe liste *cons* şi o definiţie omeomorfă la dreapta, pe liste *snoc* şi se încearcă să se "potrivească" cele două definiţii astfel încât să conţină acelaşi operator. Dacă operatorul găsit este asociativ, atunci este cel căutat.

Metoda CS:

1. Furnizarea a două definiţii secvenţiale pentru o funcţie dată: o definiţie *cons* şi o definiţie *snoc*.

2. Determinarea unei generalizări CS, aplicată definiţiilor *cons* şi *snoc*.

3. Dacă asociativitatea operatorului $\otimes$ poate fi demonstrată inductiv, atunci din Teorema 7.3 rezultă că $\otimes$ este operatorul căutat.

Exemplul 7.3 (**extragere omeomorfism** pentru **prefix**) Pentru listele cu un element $prefix.[a] = [a]$, şi deci $f = [.]$. Pentru $prefix$, definiţiile *cons* şi *snoc* sunt:

$$[a] \otimes prefix(\odot).y = a \triangleright (map(a\odot).(prefix(\odot).y))$$
$$prefix(\odot).x \otimes [b] = (prefix(\odot).x \triangleleft (last.(prefix(\odot).x) \odot b)$$

Generalizarea CS conduce la $u \otimes v = u \diamond map(last.(u)\odot).v$. Se poate demonstra uşor asociativitatea operatorului $\otimes$ şi astfel s-a demonstrat că $prefix$ este un omeomorfism cu $f = [.]$ şi $\otimes$ definit ca mai sus.

7.1.2 Aproape-omeomorfisme

Omeomorfismele pe liste se pot folosi cu mare succes în construcţia programelor paralele, dar din păcate sunt multe funcţii necesare şi interesante care nu sunt omeomorfisme. Un astfel de exemplu este funcţia mss(**"maximum segment sum"**) care caută maximul dintre sumele formate din elementele segmentelor continue dintr-o listă de numere. De exemplu, $mss.[3, -4, 2, -1, 6, -3] = 7$, unde rezultatul s-a obţinut prin adunarea elementelor segmentului $[2, -1, 6]$. Funcţia mss nu este un omeomorfism, deoarece nu e suficient să cunoaştem $mss.xs$ şi $mss.ys$ pentru a putea calcula $mss.(xs \diamond ys)$.

Un *aproape-omeomorfism* este o funcţie, care împreună cu un număr de funcţii auxiliare, formează o funcţie de tip n-uplu, care este un omeomorfism. Ceea ce este dificil este de a găsi funcţiile auxiliare care împreună cu funcţia dată formează un omeomorfism.

Poate fi folosită o abordare similară cu metoda CS. Metoda CS începe cu definirea funcţiei mss pentru liste *cons*. Notăm cu $\uparrow$ operatorul care returnează maximul a două argumente. Pentru orice element a şi lista x, $mss.(a \triangleright x)$ ar fi egală cu $mss.[a] \uparrow mss.x$; dar astfel nu se ţine cont de segmentele iniţiale care se pot construi. De aceea, se introduce funcţia mis("maximum initial segment") şi obţinem următoarea definiţie a funcţiei mss:

$$mss.(a \triangleright x) = mss.[a] \uparrow mss.x \uparrow (a + mis.x)$$
$$mis(a \triangleright x) = mis.[a] \uparrow (a + mis.x).$$

În mod analog, se defineşte funcţia mss pentru liste *snoc* şi pentru aceasta se defineşte funcţia mcs("maximum concluding segment"):

$$mss.(x \triangleleft a) = mss.x \uparrow mss.[a] \uparrow (mcs.x + a)$$
$$mcs.(x \triangleleft a) = (mcs.x + a) \uparrow mcs.[a]$$

Avem deocamdată funcţia (mss, mis, mcs), dar funcţia mis nu poate fi definită pentru listele *snoc* doar pe baza acestor funcţii şi similar funcţia mcs nu poate fi definită pentru liste *snoc*. Se introduce, de aceea, o altă funcţie ts("total sum") care determină suma totală a unei liste de întregi. Rezultă, aşadar, un 4-uplu (mss, mis, mcs, ts) pentru care avem următoarele definiţii *cons* şi *snoc*:

$$
\begin{aligned}
&mss.(a \triangleright x) = mss.[a] \uparrow mss.x \uparrow (a + mis.x)\\
&mis.(a \triangleright x) = mis.[a] \uparrow (a + mis.x)\\
&mcs.(a \triangleright x) = (a + ts.x) \uparrow mcs.x\\
&ts.(a \triangleright x) = ts.[a] + ts.x\\
&mss.(x \triangleleft a) = mss.x \uparrow mss.[a] \uparrow (mcs.x + a)\\
&mis.(x \triangleleft a) = mis.x \uparrow (ts.x + a)\\
&mcs.(x \triangleleft a) = (mcs.x + a) \uparrow mcs.[a]\\
&ts.(x \triangleleft a) = ts.x + ts.[a]
\end{aligned}
$$

Funcţia f determină rezultatul aplicării 4-uplului de funcţii pe liste cu un singur element:

$$f.a =< mss, mis, mcs, ts > .[a] = (a \uparrow 0, a \uparrow 0, a \uparrow 0, a).$$

Urmează să găsim părţile comune din definiţiile *cons* şi *snoc* pentru a găsi definiţia generală pentru $\diamond$. Pentru funcţia ts se observă că este şi de stânga şi de dreapta şi deci definiţiile se potrivesc perfect. Pentru celelalte funcţii trebuie înlocuit a. Cea mai directă modalitate de înlocuire este cu $ts.[a]$, care este egal cu a pentru listele cu un element. Dar în acest caz definiţiile pentru mss nu se potrivesc.

O altă modalitate este de a înlocui, pentru mss, pe a cu $mcs.[a]$ în definiţia *cons* şi cu $mis.[a]$ în definiţia *snoc*:

$$
\begin{aligned}
&(a + mis.x) \uparrow mss.x = (mcs.[a] + mis.x) \uparrow mss.x\\
&mss.x \uparrow (a + mis.x) = mss.x \uparrow (mcs[a] + mis.x).
\end{aligned}
$$

Pentru definiţiile mis şi mcs se înlocuieşte a cu $ts.[a]$, acolo unde nu a fost deja înlocuit.

Ca urmare se obţin următoarele definiţii:

$$
\begin{aligned}
&mss.(a \triangleright x) = mss.[a] \uparrow mss.x \uparrow (mcs.[a] + mis.x)\\
&mss.(x \triangleleft a) = mss.x \uparrow mss.[a] \uparrow (mcs.x + mis.[a])\\
&\\
&mis.(a \triangleright x) = mis.[a] \uparrow (ts.[a] + mis.x)\\
&mis.(x \triangleleft a) = mis.x \uparrow (ts.x + mis.[a])\\
&\\
&mcs.(a \triangleright x) = (mcs.[a] + ts.x) \uparrow mcs.x\\
&mcs.(x \triangleleft a) = (mcs.x + ts.[a]) \uparrow mcs.[a]\\
&\\
&ts.(a \triangleright x) = ts.[a] + ts.x\\
&ts.(x \triangleleft a) = ts.x + ts.[a]
\end{aligned}
$$

Se ajunge la următoarea definiţie a operatorului $\otimes$:

$$
\begin{aligned}
&< mss.x, mis.x, mcs.x, ts.x > \otimes < mss.y, mis.y, mcs.y, ts.y >=\\
&\quad ((mss.x) \uparrow (mss.y) \uparrow (mcs.x + mis.y), mis.x \uparrow (ts.x + mis.y)),\\
&\quad (mcs.x + ts.y) \uparrow mcs.y, (ts.x + ts.y)).
\end{aligned}
\tag{7.6}
$$

Asociativitatea operatorului poate fi demonstrată prin calcul direct şi rezultă că funcţia definită prin 4-uplul de funcţii este omeomorfism.

Funcţia mss se calculează după formula:

$$mss = \pi_1 \circ red(\otimes) \circ map(f),$$

unde π_1 este o proiecţie care returnează prima componenta din 4-uplu.

Această reprezentare omeomorfă este paralelizabilă în modul standard pe o structură de procesoare de tip arbore. Deoarece şi funcţia f şi operatorul $\otimes$ necesită un timp constant de calcul, complexitatea timp totală este $O(\log_2 n)$. Numărul de procesoare poate fi redus la $O(n/\log_2 n)$ simulându-se nivelele joase ale arborelui, în mod secvenţial, conform teoremei lui Brent.

7.2 Implementare

Metoda standard de implementare a omeomorfismelor este de a folosi reprezentarea:

$$h = red(\oplus) \circ map(f).$$

Funcţia map se execută în paralel într-un timp constant, iar reducerea se face într-un timp $O(\log_2 n)$, dacă dimensiunea listei este n. Această implementare este optimă numai dacă presupunem că există suficiente procesoare, sau mai exact numărul de procesoare creşte liniar cu dimensiunea datelor.

O abordare mai practică este a considera un număr limitat de procesoare p (paralelism limitat). Introducem tipul $[\alpha]_p$ al listelor cu lungimea p, şi indexăm funcţiile definite pe acest tip de liste cu p (de exemplu map_p). Partiţionarea unei liste arbitrare în p subliste, numite *blocuri*, este realizată cu ajutorul *functiei distribuţie*, $dist^{(p)} : [\alpha] \to [[\alpha]]_p$. Următoarea egalitate este evidentă: $red(\diamond) \circ dist^{(p)} = id$.

Teorema 7.4 (Promovare) *Pentru un $\otimes$-omeomorfism h:*

$$h \circ red(\diamond) = red(\otimes) \circ map(h). \tag{7.7}$$

Datorită egalităţii (7.7), abstractizarea standard a omeomorfismului h pe p procesoare, se transcrie astfel:

$$h = red(\otimes) \circ map_p(h) \circ dist^{(p)}. \tag{7.8}$$

Pentru a ilustra folosirea schemei (7.8) reluăm exemplul MSS studiat. Deoarece 4-uplul $< mss, mis, mcs, ts >$ defineşte un omeomorfism cu operatorul $\otimes$ definit de (7.6), poate fi aplicată teorema de promovare şi rezultă:

$$mss = \pi_1 \circ red(\otimes) \circ map_p(< mss, mis, mcs, ts >) \circ dist^{(p)} \tag{7.9}$$

Cele trei etape sunt obţinute direct din abstractizarea (7.9), care este urmată de $dist^{(p)}$.

Prima etapă $map_p(< mss, mis, mcs, ts >)$ este implementată prin apelul unui program secvenţial, în fiecare procesor pentru blocul-listă local, care are dimensiunea n/p; ca urmare complexitatea acestei etape este $O(n/p)$. Reducerea $red(\otimes)$ necesită un timp $O(\log_2 p)$, iar ultima etapă – proiecţia se realizează într-un timp constant. Deci complexitatea totală este $O(n/p + \log_2 p)$.

Putem analiza trecerea de la paralelism nelimitat (fără limitarea numărului de procesoare) la paralelism limitat şi separat pentru funcţionalele *map* şi *red*, independent, nu doar pentru omeomorfisme.

Pentru funcţionala *map* avem următoarea egalitate:

$$map\ f\ = flat \circ map_p\ (map\ f) \circ dist^{(p)} \tag{7.10}$$

unde funcţia $flat : [[\alpha]]_p \rightarrow [\alpha]$ transformă o listă de subliste într-o listă folosind concatenarea ($flat = red(\diamond)$).

Pentru reducere, transformarea este:

$$red(\oplus)\ = red_p(\oplus) \circ map_p\ (red(\oplus)) \circ dist^{(p)} \tag{7.11}$$

Vom considera că doar funcţiile cu indicele p vor fi distribuite pe procesoare. Celelalte vor fi calculate secvenţial.

7.2.1 Sortare prin numărare

Ideea algoritmului de sortare prin numărare este următoarea: se determină rangul fiecărui element din secvenţa nesortată şi se plasează apoi fiecare element pe poziţia corespunzătoare rangului său. Rangul unui element este egal cu numărul de elemente mai mici decât el [96].

Sortarea prin numărare nu este un algoritm de sortare secvenţială foarte eficient, pentru că complexitatea-timp pentru cazul secvenţial este $O(n^2)$ (n este lungimea secvenţei). Dar acestă idee conduce la algoritmi paraleli eficienţi.

Folosind acest exemplu simplu, vom ilustra faptul că proiectarea abstractă bazată pe formalismul BMF poate să analizeze diferite cazuri care pot apare la faza de implementare. Şi astfel se evidenţiază faptul că abstractizarea nu exclude performanţa [137].

Proiectarea BMF

Bazat pe definiţia metodei se ajunge la următorul program BMF pentru calcularea reangului fiecărui element:

$$
\begin{aligned}
&rank : [\alpha] \to [\alpha] \\
&rank.l = map\ (count.l).l \\
\\
&count : [\alpha] \times \alpha \to [\alpha] \\
&count.l.x = (red(+) \circ map\ (f.x)).l \\
\\
&f : \alpha \times \alpha \to \alpha \\
&f.x.y = \begin{cases} 1, \text{ if } x \geq y \\ 0, \text{ if } x < y \end{cases}
\end{aligned}
\tag{7.12}
$$

Am considerat α ca fiind un tip pentru care există o relaţie de ordine între instanţele sale, iar $[\alpha]$ este tipul listelor construite peste acest tip.

Putem face o transformare foarte simplă:

$$(red(+) \circ map\ (f.x)).l = red(+).(map\ (f\ x).l) \tag{7.13}$$

care presupune doar că aplicarea funcţiei $f.x$ se face în acelaşi pas cu reducerea şi nu într-un pas separat secvenţial.

Pentru calculul funcţionalei *map* ar fi nevoie de un număr de procesoare egal cu lungimea listei de intrare – n. Fiecare aplicare a funcţiei $count.l$ reprezintă o reducere care poate fi calculată cu o complexitate-timp egală cu $O(\log_2 n)$, complexitatea-procesor fiind $O(n)$. Deci pentru întreg programul complexitatea-timp rămâne $O(\log_2 n)$.

Adaptarea la paralelism limitat

Pentru a adapta programul la paralelism limitat impunem ca numărul de procesoare să fie egal cu p şi vom folosi funcţia $dist^{(p)}$.

Se poate remarca faptul că algoritmul conţine două faze: una reprezentată de funcţia *map* şi cealaltă reprezentată de funcţia *count*, fiecare având lista l ca şi argument. Funcţia $dist^{(p)}$ divide lista argument în p subliste echilibrate ca lungime. Prin urmare, o putem aplica fie pentru calculul funcţiei *map*, fie pentru calculul funcţiei *count*, fie pentru amândouă.

Va trebui, de asemenea, să analizăm două cazuri:

1. $p \leq n$,
2. $n < p \leq n^2$

Cazul $p \leq n$

Dacă numărul de procese este egal sau mai mic decât lungimea secvenţei, atunci funcţia $dist^{(p)}$ poate fi aplicată doar o singură dată: pentru funcţia *map*, sau pentru funcţia *count*.

```
for i = 0, p − 1 in parallel do
  for k = 0, n/p − 1 do
    global_read(A[i ∗ n/p + k], ak);
    r = 0;
    for j = 0, n − 1 do {se numără câte elemente sunt mai mici decât ak}
      global_read(A[j], aj);
       if (aj < ak)  r = r + 1; end if
    end for
    global_write(r, R[i ∗ n/p + k]);
  end for
end for
```

Figura 7.1: Sortare prin numărare – programul SM pentru $p \leq n$.

În primul caz, obţinem următorul program BMF, folosind regula ecuaţională (7.10):

$$\begin{aligned} &rank.l = (flat \circ map_p\ (map(count)) \circ dist^{(p)}).l \\ &count.l.x = red(+).(map\ (f.x).l) \end{aligned} \tag{7.14}$$

Aceasta înseamnă că fiecare procesor calculează secvenţial rangul pentru n/p elemente.

Dacă aplicăm funcţia *dist* funcţiei *count* obţinem următorul program BMF:

$$\begin{aligned} &rank.l = map(count.l).l \\ &count.l.x = (red_p(+) \circ map_p\ (red(+)) \circ map\ (f.x) \circ dist^{(p)}).l \end{aligned} \tag{7.15}$$

S-a folosit de această dată regula (7.11). Acest program calculează secvenţial rangurile pentru fiecare element, dar fiecare rang în parte se calculează în paralel folosind p procese. Pentru calcularea rangului elementului x, procesele compară valoarea lui x cu toate cele n/p elemente locale lor şi astfel se calculează mai întâi ragurile locale; apoi rangurile locale se adună.

Aceste două cazuri reflectă modurile în care algoritmul se poate descompune în componente, care pot fi calculate folosind p procese. Dacă considerăm că funcţiile cu indexul p vor fi calculate în paralel cu p procese şi funcţiile fără indecşi vor fi calculate secvenţial, obţinem următoarele complexităţi: (n^2/p) pentru primul caz şi $(n^2/p + n \log p)$ pentru cel de-al doilea. Evident primul este mai bun.

Implementări pentru $p \leq n$

Pe arhictecturi cu memorie partajata (SM) lista l poate fi partajată de toate procesele şi prin urmare, în mod natural vom alege prima variantă care are complexitate-timp mai bună. Putem transforma programul BMF corespunzător într-un program de tip PRAM descris în Figura 7.1.

Fiecare proces face n^2/p citiri şi n/p scrieri din/în memoria partajată. Dacă se consideră cazul unei arhitecturi CREW complexitatea este $(n^2/p + \alpha(n^2/p + n/p))$, unde α este unitatea de timp pentru accesul la memoria partajată.

```
Rank(mypid) :
   for i = 0, n − 1 do
     Bcast(A[i]); { emiţătorul este procesul care conţine A[i] }
     rl = ComputeLocalRank(A[i]);
     Reduce(rl, mypid);
   end for
```

Figura 7.2: Sortare prin numărare – programul DM pentru $p \leq n$.

Pe o arhitectură cu memorie distribuită (DM) cea de-a doua variantă este mai bună deoarece lista este distribuită pe procesoare. La un pas, un element este difuzat prin broadcast tuturor procesoarelor, care îi vor calcula rangul - sunt deci n paşi. Rangurile locale sunt însumate folosind un calcul de tip arbore binar, care reprezintă operaţia de reducere. Figura 7.2 descrie într-un limbaj de tip pseudocod un astfel de program.

Complexitatea-timp este dată de expresia $n(n/p + \log p(1 + \beta) + b\beta)$, unde β este unitatea de timp pentru o comunicaţie a unei valori, iar constanta b reflectă timpul necesar pentru broadcast.

Arhitecturile pipeline pot fi de asemenea folosite pentru implementarea acestui algoritm. Dacă sunt n procesoare, fiecare procesor are o valoare locală $a[i]$. Elementele listei sunt "pompate" pe rând şi rangul fiecărui element "pompat" este actualizat în fiecare procesor prin compararea cu valoarea locală. Rangul curent trebuie să fie de asemenea "pompat". Dacă $p < n$ atunci fiecare procesor va avea mai multe valori locale şi deci va face mai multe comparaţii la un pas. Complexitatea-timp, pentru această implementare, este: $(n + p - 1)(n/p + \beta)$, unde β este unitatea de timp necesară unei comunicaţii între două procesoare vecine.

Cazul $n < p \leq n^2$

Dacă avem mai mult de n procese şi $p = q * r$, putem folosi funcţia *dist* atât pentru calcularea funcţionalei *map* cât şi pentru *count*. Şi astfel ajungem la următorul program BMF:

$$\begin{aligned} rank.l &= (flat \circ map_q\ (map\ (count.l)) \circ dist^{(q)}).l \\ count.l.x &= (red_r(+) \circ map_r\ (red(+)) \circ map\ (f.x) \circ dist^{(r)}).l \end{aligned} \tag{7.16}$$

Complexitatea-timp în acest caz este $(n/q)(n/r + \log r)$.

Pe o arhitectură SM programul diferă de programul prezentat în Figura 7.1 prin faptul că fiecare element al rezultatului funcţiei *count* este calculat folosindu-se un calcul de tip arbore binar. Fiecare procesor face $n^2/p + (n/q)\log r$ citiri şi $(n/q)\log r$ scrieri din/în memoria partajată.

Pentru o arhitectură DM procesoarele pot fi aranjate ca o reţea grid(plasă) de dimensiune $q \times r$, iar distribuţia datelor de pe prima linie poate fi replicată pe celelelte linii de procesoare. Fiecare linie calculează ragurile unei subliste de n/q elemente. Complexitatea-timp este $n^2/p + n/q(\log r(1 + \beta) + b\beta)$.

O arhitectură pipeline cu mai mult de n procesoare nu este folositoare pentru această problemă.

Sortarea prin numărare, deşi este un algoritm secvenţial ineficient conduce la algoritmi paraleli eficienţi; deci relevă faptul că nu neapărat cea mai bună variantă secvenţială conduce la cei mai buni algoritmi paraleli. Sortarea paralelă prin numărare a fost implementată în special pentru arhitecturile de tip SM [173]. Am evidenţiat aici, într-un mod formalizat, bazat pe BMF, că şi arhitecturile DM şi pipeline pot fi considerate pentru o implementarea eficientă.

Prin acest exemplu, se evidenţiază faptul că programele BMF pot fi transformate formal pentru paralelism limitat, iar variantele obţinute pot fi analizate mai departe pentru alegerea celei mai bune, în funcţie de arhitectura ţintă.

7.3 Tipuri de date categoriale

BMF a fost dezvoltat iniţial doar pentru structuri de date de tip listă. Ulterior, a fost extins pentru tipuri de date categoriale [155], care sunt extensii ale tipurilor abstracte de date.

Definiţia 7.4 (Categorie) *O* categorie *este o structură cu obiectele $A, B, \ldots$, săgeţile $f, g, \ldots$ şi cu o operaţie binară asociativă $\cdot$ (compunere) pe săgeţi, astfel încât:*

$$\frac{f : A \to B \;\; g : B \to C}{(g \cdot f) : A \to C}$$

care înseamnă că oricare ar fi săgeţile $f : A \to B$ şi $g : B \to C$ există o săgeată $(g \cdot f) : A \to C$ şi

$$\frac{}{id_A : A \to A}$$

astfel încât pentru fiecare săgeată $f : A \to B$,

$$f \cdot id_A = f$$
$$id_b \cdot f = f$$

Considerăm o categorie $Type$, ale cărei obiecte sunt mulţimi, iar săgeţile sunt funcţii totale. Impunem următoarele proprietăţi pentru această categorie:

- Are un obiect iniţial **0**. Există o săgeată unică de la acest obiect la oricare alt obiect al categoriei.
- Are un obiect final **1**. Există o săgeată unică de la oricare alt obiect al categoriei la acest obiect.
- Are produse, astfel încât pentru toate perechile de obiecte A şi B, există un obiect $A \times B$ şi proiecţiile de la el la A şi B.

- Are coproduse, astfel încât pentru toate perechile de obiecte A şi B, există un obiect $A + B$ şi injecţiile de la A şi B la el (un element aparţine mulţimii $A + B$ dacă aparţine mulţimii A sau mulţimii B).
- pentru fiecare pereche de săgeţi $f : A \to B$ şi $g : B \to C$ există o săgeată, numită joncţiune, $f \triangledown g : A + B \to C$.

Un functor este un omeomorfism pe categorii.

Definiţia 7.5 (Functor) *Un* functor $F : \mathcal{C} \to \mathcal{D}$ *mapează obiectele categoriei* C *pe obiectele categoriei* D *şi săgeţile categoriei* $\mathcal{C}$ *spre săgeţile categoriei* $\mathcal{D}$*, astfel încât păstrează compunerile şi identităţile:*

$$\begin{array}{rcl} F(f : A \to B) & = & F(f) : F(A) \to F(B) \\ F(g \cdot f) & = & F(f) \cdot F(g) \\ F(id_A) & = & id_{F(A)} \end{array}$$

Tipurile de date categoriale ($A*$) sunt construite formal prin precizarea unor constructori definiţi pentru o mulţime de bază (A). De exemplu, pentru liste omogene (posibil vide), aceşti constructori sunt:

$$\begin{array}{lcl} [] & : & \mathbf{1} \to A* \\ [\,\cdot\,] & : & A \to A* \\ \diamond & : & A* \times A* \to A* \end{array}$$

Aceşti trei constructori corespund unei singure săgeţi: $[] \triangledown \cdot \triangledown \diamond : \mathbf{1} + A + A* \times A* \to A*$, a categoriei de bază ($Type$). Pentru a arăta că obiectul $A*$ există, se defineşte un functor $T : A* \to \mathbf{1} + A + A* \times A*$ care este definit de $T = K_1 + K_A + | \times |$, unde K_X este functorul constant X care mapează fiecare obiect spre X şi fiecare săgeată spre id_X. Functorul T este un functor polinomial şi într-o astfel de categorie are un punct fix; $A*$ se defineşte ca fiind partea obiect a punctului fix al functorului T [158].

Dintr-o anumită perspectivă, tipurile de date categoriale captează un stil al programării funcţionale de ordin doi, în care programele sunt construite din forme restrictive ale unor funcţii de ordin doi. Din altă perspectivă, ele sunt extensii de obiecte fără stare, sau tipuri abstracte de date. Tipurile categoriale de date încapsulează fluxul de control şi reprezentarea datelor. Prin încapsularea fluxului de control, programatorul nu trebuie să specifice cum sunt aranjate operaţiile de calcul şi de comunicare pe tipurile de date. Aceasta va cădea în sarcina implementatorilor. În acest fel se realizează o separare a noţiunilor, la un nivel corect.

Programele bazate pe tipuri categoriale de date sunt compuneri de operaţii monolitice, care calculează omeomorfisme pe obiectele tipurilor de date. Aceste operaţii ascund detaliile interne de calcul. În cazul paralel, aceasta înseamnă că se ascunde modul în care aceste calcule sunt descompuse în procese, modul în care acestea sunt mapate pe procesoare şi modul în care acestea comunică şi se sincronizează. Prin acestea, ele permit dezvoltarea de programe *independente de arhitectură*.

Dezvoltarea de programe are o metodologie formată din două părţi. Prima se bazează pe faptul că structura calculului oricărui omeomorfism este aceeaşi, astfel încât programatorii se pot concentra asupra părţilor specifice fiecărui omeomorfism în parte. A doua constă în faptul că construcţia unui tip de date furnizează un set de ecuaţii care pot fi folosite pentru transformarea programului. Această abordare încurajează dezvoltarea derivaţională, prin care specificaţiile sunt transformate în programe. Se poate considera că această abordare este restrictivă, pentru că se bazează pe omeomorfisme. Dar, totuşi toate funcţiile injective sunt omeomorfisme şi prin aceasta se includ foarte multe funcţii. Pe de altă parte, anumite funcţii pot fi adaptate, astfel încât să fie reprezentate ca un omeomorfism urmat de o proiecţie (aproape-omeomorfisme).

Pot fi construite diferite tipuri de date categoriale, bazate pe: liste, arbori, mulţimi, tablori, grafuri. Prezentăm în continuare tipul arbore binar omogen.

7.3.1 Tipul arbore binar omogen

Obiectele acestui tip sunt arbori binari cu o valoare dintr-un anumit tip de bază A pe fiecare nod (frunze sau noduri terminale) – prin urmare sunt omogeni – şi un nod are doi descendenţi sau nici unul.

Acest tip are doi constructori, unul transformă o valoare de tip A într-un arbore care constă dintr-un singur nod, iar cel de-al doilea transformă doi arbori şi o valoare de tip A într-un arbore mai mare care îi conţine.

$$\begin{array}{lcl} Leaf & : & A \to A_t \\ Join & : & A_t \times A \times A_t \to A_t \end{array}$$

Omeomorfismele pe aceşti arbori sunt funcţii definite pe o mulţime A_t, cu valori într-o mulţime P cu operaţiile:

$$\begin{array}{lcl} p_1 & : & A \to P \\ p_2 & : & P \times A \times P \to P \end{array}$$

De fapt, există o corespondenţă biunivocă între asemenea structuri (mulţime + operaţii) şi omeomorfisme.

Astfel, dacă se doreşte găsirea unui omeomorfism h de la arborii A_t la o anumită mulţime P, se poate porni direct de la ce trebuie să facă acel omeomorfism, sau se poate porni de la operaţiile din P care trebuie să corespundă acţiunii lui h. Aceasta este în general o problemă mai simplă.

Toate omeomorfismele pe arbori pot fi calculate folosind aceeaşi schemă recursivă, parametrizată cu funcţiile p_1 şi p_2:

$$\begin{array}{l} eval\_hom(p_1, p_2, t) \\ if\ (t = Leaf(a)) \ \to p_1(a) \\ \ []\ (t = Join(t1, a, t2)) \ \to \\ \qquad p_2(eval\_hom(p_1, p_2, t1), \\ \qquad\quad a, \\ \qquad\quad eval\_hom(p_1, p_2, t2)) \end{array}$$

Exemplul 7.4 (**înălţimea**) Înălţimea unui arbore binar poate fi calculată printr-un omeomorfism, pentru care:

$$p_1 = K_0 : A \to \mathbb{N}$$
$$p_2 = \oplus : \mathbb{N} \times A \times \mathbb{N} \to \mathbb{N}$$

unde $\oplus(n, a, m) =\uparrow (n, m) + 1$ şi K_0 este funcţia constant nulă.

Exemplul 7.5 (**map**) Funcţionalele de tip $map(f)$ sunt omeomorfisme de la o mulţime de arbori A_t la o altă mulţime de arbori B_t, care aplică funcţia f pe fiecare nod al arborelui. Ele se pot exprima formal prin precizarea funcţiilor p_1 şi p_2:

$$\begin{aligned} p_1 &= (Leaf \cdot f) : A \to B_t \\ p_2 &= (Join \cdot id \times f \times id) : B_t \times A \times B_t \to B_t \end{aligned}$$

În acestă abordare, complexitatea-timp de evaluare este proporţională cu adâncimea recursivităţii, care este înălţimea arborelui. Evident, această operaţie merită să fie considerată ca un caz special cu o implementare mai bună, în care funcţia f este aplicată pe fiecare nod simultan.

Exemplul 7.6 (**red**) Pentru funcţionala $red(\oplus)$ avem:

$$\begin{aligned} p_1 &= id : A \to A \\ p_2 &= \oplus : A \times A \times A \to A \end{aligned}$$

unde s-a folosit varianta ternară a operatorului asociativ $\oplus$.

Sumar

Modelul funcţional, prezentat în acest capitol, are în vedere două aspecte importante ale programării paralele: abstractizarea şi performanţa.

Abordarea construirii programelor folosind funcţionalele:

- scuteşte programatorul de a lua decizii care ţin de detaliile de nivel jos al unei aplicaţii particulare;
- furnizează implementări standard care măresc încrederea în corectitudinea programelor ţintă;
- oferă predicţia performanţei;
- furnizează metode care păstrează corectitudinea, pentru compunere şi rafinare.

Prin abstractizare se trece de la particular la general, ceea ce dă noi dimensiuni principiilor programării paralele.

Posibilele pierderi în ceea ce priveşte performanţa sunt compensate de portabilitate şi de uşurinţa programării.

Capitolul 8

Structuri de date pentru specificarea paralelismului

Algebrele de date distribuite sunt o noţiune abstractă utilizată pentru descrierea programelor paralele. Ideea de bază este aceea că structura de date este împărţită în subobiecte care pot fi alocate mai multor procesoare.

Pentru că se încearcă găsirea unor scheme de paralelizare cât mai simple şi mai eficiente se poate pune problema găsirii unor metode de specificare specializate pe anumite clase de probleme.

Algoritmii recursivi intervin în rezolvarea unei mari varietăţi de probleme, constituindu-se astfel într-o clasă importantă de probleme. *PowerList*, *ParList* şi *PList* sunt structuri de date liniare, ce pot fi folosite cu succes în descrierea funcţională simplă a programelor paralele care sunt de natura Divide&Impera. Folosind tehnici formale se pot deriva descrieri succinte pentru programele paralele, plecând de la specificaţii. Acest formalism a fost introdus prima dată de către J. Misra[121] şi dezvoltat apoi de către J. Kornerup[99]. Folosind acest formalism este posibil să se lucreze la un nivel înalt de abstractizare, prin eliminarea notaţiilor de index. Se poate spune că unul dintre motivele pentru care programarea paralelă este considerată dificilă astăzi este folosirea intensivă a notaţiilor de index în limbajele de programare paralelă. Aceasta duce şi la o demonstrare a corectitudinii programelor paralele foarte anevoioasă.

Algoritmii sunt specificaţi cu ajutorul funcţiilor definite recursiv pe aceste structuri. Funcţiile pot fi transformate prin derivări corecte bazate pe axiomele algebrelor corespunzătoare, pentru a se ajunge la algoritmi mai eficienţi.

Secţiunile 2, 3 şi 4, ale acestui capitol, prezintă cele trei structuri de date împreună cu teoria asociată lor, precum şi exemple care ilustrează avantajele şi diferenţele folosirii lor. Secţiunea 5 prezintă specificarea algoritmului de calcul al transformatei Fourier rapide în trei cazuri distincte, funcţie de gradul polinomului care este transformat, folosindu-se cele trei structuri de date prezentate. În ultima secţiune se prezintă o extindere a celor trei structuri în structuri n-dimensionale *PowerArray*, *ParArray* şi *PArray*, facilitând astfel specificarea algoritmilor care lucrează cu date structurate pe mai multe dimensiuni. Un exemplu elocvent este dat prin specificarea unui algoritm general pentru calcularea

formulelor recurente liniare, oarecare. Acesta este folosit la calcularea unor polinoamele ortogonale.

Complexitatea programelor paralele specificate prin acest formalism este discutată doar la un nivel abstract (fără a se ţine seama de arhitectura pe care vor fi implementate). Este analizată în schimb implementarea algoritmilor specificaţi prin *PowerList* pe arhitecturi de tip hipercub, care se dovedeşte a fi foarte simplă şi eficientă.

Tipuri folosite

Notăm cu *Type* mulţimea tuturor tipurilor folosite.

Tipurile simple folosite sunt următoarele:

Nume tip	Ce defineşte
Nat	Numere naturale
Pos	Numere naturale pozitive
Real	Numere reale
Com	Numere complexe
Bool	Valori booleene

Tipul unei funcţii este precizat prin numele funcţiei, domeniu şi codomeniu.

Tipurile structurilor *PowerList*, *ParList* şi *PList* sunt formalizate prin introducerea unei funcţii – *constructor de tip* pentru fiecare structură. Aceste funcţii au două argumente, un tip şi o lungime şi returnează tipul tuturor instanţelor structurii cu elemente de tipul specificat şi lungimea egală cu lungimea dată.

8.1 Structuri de date *PowerList*

În acestă secţiune vom prezenta structura de date *PowerList* şi algebra definită pe mulţimea structurilor de date de acest tip. Demonstrarea proprietăţilor acestor structuri şi definirea funcţiilor pe aceste structuri se face pe baza inducţiei structurale definite pe mulţimea structurilor *PowerList*. Este analizată maparea algoritmilor specificaţi cu ajutorul structurilor *PowerList* pe arhitecturi de tip hipercub. În final, sunt prezentaţi algoritmi paraleli pentru calcularea valorii unui polinom şi pentru integrare numerică Romberg şi Simpson.

8.1.1 Definiţii

O structură *PowerList* este o structură liniară ale cărei elemente sunt de acelaşi tip. Lungimea unei structuri *PowerList* este o putere a lui 2. O structură *PowerList* de lungime 1 se numeşte *singleton* şi este reprezentată astfel: $[a]$, unde a este unicul element al listei.

Constructorul de tip asociat acestei structuri de date este:

$$PowerList : Type \times Nat \rightarrow Type$$

care are ca argumente un tip (X) şi un număr natural (k) şi returnează tipul tuturor structurilor $PowerList$ cu elemente de tipul X şi cu lungimea egală cu 2^k. De exemplu, $PowerList.Nat.2$ este tipul tuturor structurilor $PowerList$ cu lungime 2^2 care conţin elemente numere naturale.

Două structuri $PowerList$ care au aceeaşi lungime şi elemente de acelaşi tip se numesc *similare* (aparţin unui tip $PowerList.X.n$).

Două structuri $PowerList$ similare p, q, pot fi combinate într-o listă de lungime dublă în două moduri diferite:

- $p \mid q$ *(tie)* este lista care conţine elementele din p urmate de elementele din q,

- $p \# q$ *(zip)* este lista formată din elementele din p şi q luate alternativ.

Ca urmare, constructorii de structuri $PowerList$ sunt următorii:

$$\begin{array}{rcl} [.] & : & X \to PowerList.X.0 \\ . \mid . & : & PowerList.X.n \times PowerList.X.n \to PowerList.X.(n+1) \\ .\#. & : & PowerList.X.n \times PowerList.X.n \to PowerList.X.(n+1) \end{array}$$

Funcţiile:

$$length : PowerList.X.n \to Nat \text{ şi}$$
$$loglen : PowerList.X.n \to Nat$$

sunt definite prin următoarele relaţii

$$(\forall p : p \in PowerList.X.n : length.p = 2^n)$$
$$(\forall p : p \in PowerList.X.n : loglen.p = n).$$

Considerăm următoarele **axiome ale algebrei** $PowerList$:

1. $p \in PowerList.X.n \wedge n > 0 \Rightarrow (\exists\, !u, v : u, v \in PowerList.X.(n-1) : p = u|v)$
2. $p \in PowerList.X.n \wedge n > 0 \Rightarrow (\exists\, !u, v : u, v \in PowerList.X.(n-1) : p = u\#v)$
3. $[a] = [b] \equiv a = b$
4. $p|q = u|v \equiv p = u \wedge q = v$
5. $p\#q = u\#v \equiv p = u \wedge q = v$
6. $[a] \mid [b] = [a] \# [b]$
7. $(p|q) \# (u|v) = (p\#u) \mid (q\#v)$

unde p, q, u, v sunt structuri $PowerList$.

Conform ultimei axiome putem spune că operaţiile *zip* şi *tie* "*comută*".

Un element particular al unei structuri $PowerList$ nu poate fi accesat direct. Singura cale de acces la elementele unei structuri $PowerList$ este împărţirea listei folosind operaţiile *tie* şi *zip* ca şi *deconstructori,* folosind axiomele 1 şi 2.

8.1.2 Principiul inducţiei pentru *PowerList*

Funcţiile peste structuri *PowerList* se definesc cu ajutorul inducţiei structurale. Folosind inducţia structurală putem demonstra diferite proprietăţi ale structurilor *PowerList* şi a funcţiilor definite pe acestea.

Fie $\Pi : PowerList.X.n \to Bool, X \in Type, n \in Nat$, un predicat al cărui adevăr trebuie stabilit pentru toate structurile *PowerList* cu elemente de tipul X. Putem să demonstrăm că predicatul Π este adevărat folosind următorul principiu al inducţiei:

$$
\begin{array}{l}
\quad (\forall x : x \in X : \Pi.[x]) \\
\wedge \\
\quad (\ (\forall p, q, n : p, q \in PowerList.X.n \wedge n \in Nat : \Pi.p \wedge \Pi.q \Rightarrow \Pi.(p|q)) \\
\qquad \vee (\forall p, q, n : p, q \in PowerList.X.n \wedge n \in Nat : \Pi.p \wedge \Pi.q \Rightarrow \Pi.(p\#q))\ ) \\
\Rightarrow \\
\quad (\forall p, n : p \in PowerList.X.n \wedge n \in Nat : \Pi.p)
\end{array}
$$

8.1.3 Operatori, relaţii şi funcţii

Fie $\otimes : X \times X \to X$ un operator binar definit pe tipul scalar X.

Extindem operatorul prin definirea lui pe structuri *PowerList*:

$$\otimes : PowerList.X.n \times PowerList.X.n \to PowerList.X.n$$

definit prin relaţiile:

$$
\begin{array}{l}
[a] \otimes [b] = [a \otimes b] \\
(p|q) \otimes (u|v) = (p \otimes u) \,|\, (q \otimes v) \quad sau \\
(p\#q) \otimes (u\#v) = (p \otimes u) \,\#\, (q \otimes v)
\end{array}
$$

Ultimele două egalităţi nu sunt independente, fiecare putând constitui ipoteza de plecare în demonstrarea celeilalte, prin inducţie structurală.

Ele exprimă faptul că $|$ şi $\otimes$ (respectiv $\#$ şi $\otimes$) "comută".

Relaţiile peste tipurile scalare sunt extinse în acelaşi mod ca şi operatorii binari.

Fie Δ o relaţie peste tipul X, definită prin funcţia $\Delta : X \times X \to Bool$ şi fie $p, q, u, v \in PowerList.X.n$ şi $x, y \in X$. Definim relaţia extinsă prin:

$$\Delta : PowerList.X.n \times PowerList.X.n \to Bool$$

$$
\begin{array}{l}
[x]\,\Delta\,[y] \equiv [x \Delta y] \\
(p\#q)\,\Delta\,(u\#v) \equiv (p \Delta u) \wedge (q \Delta v)
\end{array}
$$

Deci două structuri *PowerList* similare sunt în relaţia Δ dacă şi numai dacă elementele de pe aceeaşi poziţie sunt în relaţie. Se poate folosi şi operatorul *tie* pentru definirea relaţiei:

$$(p|q)\,\Delta\,[u|v] \equiv (p \Delta u) \wedge (q \Delta v)$$

Prezentăm în continuare câteva exemple de funcţii definite pe structuri *PowerList*:

1. $sum : PowerList.X.n \to X$

   calculează suma elementelor unei structuri $PowerList$. Presupunem că $\oplus : X \times X \to X$ este un operator aditiv asociativ, definit pe X:

   $$sum.[a] = a$$
   $$sum.(p|q) = sum.p \oplus sum.q$$

   Reţeaua de calcul a funcţiei *sum* are forma unui arbore binar total echilibrat.

2. Funcţia *sum* este o particularizare a funcţiei de nivel înalt de reducere

   $reduce : (X \times X \to X) \times PowerList.X.n \to X$ definită prin:

   $$reduce.\otimes.[a] = a$$
   $$reduce.\otimes.(p|q) = reduce.\otimes.p \otimes reduce.\otimes.q$$

   unde $\otimes : X \times X \to X$ este un operator asociativ.

   Funcţia *sum* se obţine prin instanţierea $sum = reduce.\oplus$.

3. Un alt exemplu de funcţională este funcţia

   $$map : (X \to Z) \times PowerList.X.n \to PowerList.Z.n$$

   care primeşte ca argumente o funcţie şi o structură $PowerList$ şi aplică funcţia fiecărui element din structură. Este definită prin:

   $$map.f.[a] = [f.a]$$
   $$map.f.(p\#q) = map.f.p \,\#\, map.f.q$$

   Un exemplu de aplicare a funcţionalei *map* este determinarea listei formate din elementele în valoarea absolută a unei liste de întregi, prin aplicarea funcţiei *abs:*

   $$map.abs.[17, -3, 8, -5, 6] = [17, 3, 8, 5, 6]$$

   (Am folosit pentru specificarea explicită a unei structuri $PowerList$ parantezele drepte.)

   Observaţie: Funcţia *abs* poate fi extinsă astfel încât să aibă argumente de tip listă, aşa cum am arătat la extinderea operatorilor binari şi relaţiilor.

4. Funcţia $rev : PowerList.X.n \to PowerList.X.n$ este o funcţie de permutare care returnează o listă formată din elementele liste argument în ordine inversă.

   $$rev.[a] = [a]$$
   $$rev.(p|q) = rev.q|rev.p \quad \text{sau}$$
   $$rev.(p\#q) = rev.q\#rev.p$$

Pentru a ilustra modul de demonstrare a proprietăţilor folosind principiul inducţiei structurale, demonstrăm următoarea propoziţie:

Propoziţia 8.1

$$rev.(map.f.p) = map.f.(rev.p) \qquad \forall p \in PowerList.X.k, \forall k \in Nat \wedge X \in Type$$

Demonstraţie: *Cazul de bază:*

$$\begin{array}{ll} & rev.(map.f.[a]) \\ = & \quad \{\text{definiţia } map\} \\ & rev.([f.a]) \\ = & \quad \{\text{definiţia } rev\} \\ & [f.a] \\ = & \quad \{\text{definiţia } map\} \\ & map.f.[a] \\ = & \quad \{\text{definiţia } rev\} \\ & map.f.(rev.[a]) \end{array}$$

Pasul inductiv:

Presupunem că proprietatea e adevărată pentru toate structurile $PowerList.X.k, k > 0$ şi demonstrăm proprietatea pentru structurile $PowerList.X.(k+1)$.

$$\begin{array}{ll} & rev.(map.f.(p\#q)) \\ = & \quad \{\text{definiţia } map\} \\ & rev.(map.f.p\#map.f.q) \\ = & \quad \{\text{definiţia } rev\} \\ & rev.(map.f.q)\#rev.(map.f.p) \\ = & \quad \{\text{ipoteza inducţiei}\} \\ & map.f.(rev.q)\#map.f.(rev.p) \\ = & \quad \{\text{definiţia } map\} \\ & map.f.(rev.q\#rev.p) \\ = & \quad \{\text{definiţia } rev\} \\ & map.f.(rev.(p\#q)) \end{array}$$

8.1.4 Complexitatea funcţiilor definite pe $PowerList$

Analizăm complexitatea-timp a programelor paralele descrise cu ajutorul structurilor $PowerList$, doar la nivel abstract fără să includem şi costurile comunicaţiilor care depind de implementarea pe o anumită arhitectură. Algoritmii descrişi cu ajutorul structurilor $PowerList$ sunt de natură Divide&Impera şi determină o partiţionare a datelor în două părţi de dimensiuni egale. Aceasta conduce la o complexitate în funcţie de $n, n = loglen.p$ (unde p este lista de intrare). Putem deci să calculăm complexitatea-timp în funcţie de n şi nu de $length.p$. Calculăm mărimea $P.n$, care reprezintă complexitatea unui pas de combinare a rezultatelor obţinute prin divizare; în final complexitatea-timp se aproximează prin formula $T.n = (\sum i : 0 \leq i < n : P.i)$.

Exemplul 8.1 Complexitatea funcţiei $sum : PowerList.X.n \rightarrow X$ folosind 2^{n-1} procesoare este dată de $P.i = c = O(1), \forall i$, dacă $T^{\oplus} = c$ (unde c este o constantă). Complexitatea-timp a celui mai bun algoritm secvenţial pentru calcularea sumei a 2^n numere este $T_s.n = 2^n - 1$, deci acceleraţia este $S.n = (2^n - 1)/c * n$, iar costul este $C.n = O(n * 2^n)$; rezultă deci că algoritmul specificat de funcţia *sum* este eficient.

8.1.5 Maparea pe hipercuburi

Deşi în analiza complexităţii nu am implicat şi costurile ce depind de arhitectura ţintă, totuşi există o arhitectură care se dovedeşte a fi foarte eficientă în implementarea algoritmilor specificaţi prin acest formalism.

Ca şi structurile *PowerList*, hipercuburile au mărimi care sunt puteri ale lui 2. De asemenea două hipercuburi de aceeşi mărime pot fi combinate rezultând un nou hipercub de mărime dublă. Există numeroase arhitecturi de supercalculatoare comerciale bazate pe hipercuburi. (Descrierea arhitecturii hipercub a fost dată în primul capitol.)

Dacă etichetăm elementele unei structuri *PowerList* cu un string de biţi de lungime $n(n = loglen.p)$, care reprezintă poziţia elementului respectiv în listă, atunci putem mapa fiecare element unui nod dintr-un hipercub de mărime 2^n. Folosind această mapare, operaţiile *tie* şi *zip* (deci şi funcţiile care sunt definite cu ajutorul lor) pot fi implementate eficient prin descompunea hipercubului respectiv (H_n) în cele două hipercuburi asociate (H_{n-1}).

8.1.6 Aplicaţii

Prima aplicaţie este calcularea valorii unui polinom cu gradul egal cu $2^n - 1$, iar următoarele două sunt aplicaţii de integrare numerică folosind formule repetate.

Valoarea unui polinom

Un polinom cu coeficienţii $p_j, 0 \leq j < 2^n$, unde $n \geq 0$, poate fi reprezentat printr-o *PowerList* al cărui al j-lea element este p_j. Valoarea polinomului într-un punct x este $\sum_{0 \leq j < 2^n} p_j * x^j$. Pentru $n > 0$ această expresie se poate scrie:

$$\sum_{0 \leq j < 2^{n-1}} p_{2j} * x^{2j} + \sum_{0 \leq j < 2^{n-1}} p_{2j+1} * x^{2j+1} = \sum_{0 \leq j < 2^{n-1}} p_{2j} * (x^2)^j + x * \sum_{0 \leq j < 2^{n-1}} p_{2j+1} * (x^2)^j.$$

Funcţia de evaluare *vp* foloseşte această strategie de calcul şi se va folosi operatorul #. Putem considera o mulţime de puncte (o lista w), în care să se calculeze valoarea polinomului:

$vp : PowerList.X.n \times PowerList.X.m \to PowerList.X.m$
$vp.[a].[w] = [a]$
$vp.p.(u|v) = vp.p.u|vp.p.v$
$vp.(p\#q).w = vp.p.w^2 \oplus (w \odot (vp.q.w^2))$

Operatorul $\odot$ este operatorul înmulţire extins pe structuri $PowerList$, iar $w^2 = w \odot w$. S-a folosit şi operatorul aditiv extins $\oplus$.

Ordinea de aplicare a operatorului $|$ pentru w, sau a operatorului $\#$ pentru p este irelevantă.

Complexitatea-timp depinde de m şi n şi este $T.n.m = O(n * m)$, iar constanta de multiplicitate depinde de timpul necesar unei operaţii elementare $+, *$.

Integrare numerică folosind formula repetată Romberg

Pentru o funcţie integrabilă $f : [a, b] \to \mathbb{R}$, integrala $I = \int_a^b f(x)\,dx$ poate fi aproximată folosind formula lui Romberg:

$$Q_{T_0} = \tfrac{b-a}{2}(f(a) + f(b))$$
$$Q_{T_k}(f) = \tfrac{1}{2} Q_{T_{k-1}}(f) + \tfrac{h}{2^k} \sum_{j=1}^{2^k-1} f\left(a + \tfrac{2j-1}{2^k} h\right),$$

unde $h = b - a$ şi $k = 1, 2, \ldots$.

Şirul $Q_{T_k}(f)$ converge la valoarea I a integralei funcţiei f pe intervalul $[a, b]$.

Considerăm ca date iniţiale lista valorilor funcţiei în punctele diviziunii:
$\left[a, a + \frac{h}{2^n}, ..., a + \frac{2^n-1}{2^n} h\right]$, deci $p = \left[f(a), f\left(a + \frac{h}{2^n}\right), ..., f\left(a + \frac{2^n-1}{2^n} h\right)\right]$.

Observăm că elementele de pe poziţiile pare intervin în calculul valorii $Q_{T_{k-1}}(f)$, iar cele de pe poziţiile impare intervin în calculul termenului al doilea al sumei care ne dă valoarea $Q_{T_k}(f)$. În concluzie, definirea funcţiei care calculează $Q_{T_k}(f)$ va folosi în definiţia structurală operatorul $\#$. Funcţia va avea tipul $Romb : Real \times Real \times PowerList.Real.n \to Real$, unde primul argument este $hk = \frac{h}{2^n}$ pasul diviziunii, al doilea este valoarea funcţiei în punctul b, iar al treilea este lista valorilor funcţiei. Definim funcţia $Romb$ prin următoarele relaţii:

$Romb.hk.fb.[x] = \frac{1}{2} * hk * x + \frac{1}{2} * hk * fb$
$Romb.hk.fb.(p\#q) = \frac{1}{2} * Romb.(2 * hk).fb.p + hk * sum.q$

Dacă presupunem că funcţia $sum.q$ se va calcula în paralel cu calcularea funcţiei $Romb$ pe sublista p atunci $T.n = O(n)$.

Formula repetată de integrare numerică Simpson

Aceată formulă se obţine din formula lui Romberg pe baza relaţiei:

$$Q_{S_k}(f) = \frac{1}{3}\left[4Q_{T_{k+1}}(f) - Q_{T_k}(f)\right], k = 0, 1, \ldots,$$

unde $Q_{S_k}(f)$ este formula repetată de ordin k a lui Simpson.

Definiţia funcţiei $Sims$ este:

$$Sims : Real \times Real \times PowerList.Real.(n+1) \to Real,$$

primul argument este $hk = \frac{h}{2^{n+1}}$, al doilea este valoarea funcţiei în punctul b, iar al treilea este lista valorilor funcţiei, în punctele diviziunii $\left[a, a + \frac{h}{2^{n+1}}, ..., a + \frac{2^{n+1}-1}{2^{n+1}}h\right]$, deci $p = \left[f(a), f\left(a + \frac{h}{2^{n+1}}\right), ..., f\left(a + \frac{2^{n+1}-1}{2^{n+1}}h\right)\right], loglen.p = n + 1.$

$Sims.hk.fb.[x\ y] = \frac{1}{3}Romb.hk.fb.[x] + \frac{2}{3}hk * y$
$Sims.hk.fb.(p\#q) = \frac{1}{3} * Romb.(2 * hk).fb.(p) + \frac{4}{3} * hk * sum.q$

Funcţia $Sims$ este definită folosind funcţia $Romb$. Se poate căuta o exprimare recursivă directă a funcţiei $Sims$ folosind inducţia structurală:

Cazul de bază:

$$\begin{aligned} & Sims.hk.fb.[x\ y] \\ = & \{\text{definiţia } Sims\} \\ & \tfrac{1}{3}Romb.hk.fb.[x] + \tfrac{2}{3}hk * y \\ = & \{\text{definiţia } Romb,\ \text{calcul}\} \\ & \tfrac{hk}{6}[x + 4 * y + fb] \end{aligned}$$

Pasul inductiv:

$$\begin{aligned} & Sims.hk.fb.((p\#q)\#(u\#v)) \\ = & \{\ \text{definiţia funcţiei } Sims\} \\ & \tfrac{1}{3}Romb.(2hk).fb.(p\#q) + \tfrac{4}{3}hk * sum.(u\#v) \\ = & \{\ \text{definiţia funcţiei } Romb\} \\ & (\tfrac{1}{6}Romb.(4hk).fb.p + \tfrac{1}{3}(2hk) * sum.q) + \tfrac{4}{3}hk * sum.(u\#v) \\ = & \{\ \text{definiţia funcţiei } Sims\} \\ & \tfrac{1}{2}Sims.(2hk).fb.(p\#q) + \tfrac{4}{3}hk * sum.(u\#v) - \tfrac{2}{3}hk * sum.q \end{aligned}$$

Prin urmare noua definiţie a funcţiei $Sims$ este:

$Sims.hk.fb.[x\ y] = \frac{hk}{6}[x + 4 * y + fb]$
$Sims.hk.fb.(p\#q) = \frac{1}{2} * Sims.(2hk).fb.(p) + \frac{4}{3} * hk * sum.q - \frac{2}{3} * sum2.p$

$sum2.[x, y] = y$
$sum2.(p\#q) = sum.q$

Din analiza celor două expresii ale funcţiei *Sims* se observă că este mai eficientă folosirea funcţiei *Romb* pentru calcularea funcţiei *Sims*. Explicaţia se datorează faptului că cea de-a doua expresie calculează un termen în plus faţă de prima şi de asemenea foloseşte lista p pentru calculul a două valori; dacă considerăm datele distribuite pe mai multe procesoare, rezultă că încărcarea de calcul a procesoarelor care conţin date din p este mult mai mare.

Complexitatea este $T.n = O(n)$.

8.2 Structuri de date *ParList*

Notaţia *PowerList* poate fi extinsă la liste de lungime arbitrară. Noua structură poartă numele *ParList*, care este o prescurtare de la "Parallel List". Funcţiile peste structurile *ParList* sunt definite tot pe baza inducţiei structurale, dar de data acesta printr-un caz de bază şi diferite cazuri inductive.

8.2.1 Definiţii

O structură *ParList* este o listă nevidă ale cărei elemente au acelaşi tip şi sunt ori scalari cu acelaşi tip de bază sau (recursiv) structuri *ParListe* similare. Două structuri *ParList* sunt *similare* dacă au aceeaşi lungime şi elementele lor sunt similare; doi scalari sunt similari dacă au acelaşi tip.

Clasificăm structurile *ParList* în funcţie de lungime; cea mai mică *ParList* are lungimea 1 şi se numeşte *singleton* - $[x]$.

O structură *ParList* v care nu e singleton poate fi descompusă într-un singur element şi o structură *ParList* a cărei lungime este mai mică cu o unitate faţă de lungimea lui v, folosind operatorul $\triangleright$ (*cons*) sau operatorul $\triangleleft$ (*snoc*):

$$v = a \triangleright p \wedge v = q \triangleleft b,$$

unde a, b şi elementele listelor p şi q sunt similare, iar listele p şi q sunt similare. a este primul element din v şi b este ultimul element din v. Acestă definiţie corespunde cu cea din teoria listelor liniare secvenţiale din limbajele funcţionale.

O structură $ParList$ p de lungime pară poate fi descompusă folosind operatorii $\#$ *(zip)* şi $|$ *(tie):*

$$p = u\#v \wedge p = r|s,$$

unde *u, v, r, s* sunt structuri $ParList$ similare, cu proprietăţile:

- u conţine elementele de pe poziţiile pare ale lui p,
- v conţine elementele de pe poziţiile impare ale lui p,
- r conţine prima jumătate de elemente ale lui p,
- s conţine a doua jumătate de elemente ale lui p.

Constructorul de tip asociat structurilor de date $ParList$ este definit astfel:

$$ParList : Type \times Pos \to Type$$

unde primul argument reprezintă tipul de bază al structurii $ParList$ construite, iar al doilea argument lungimea listei.

Constructorii structurilor $ParList$ sunt următorii:

$$\begin{array}{rcl} [.] & : & X \to ParList.X.1 \\ .\triangleright. & : & X \times ParList.X.n \to ParList.X.(n+1) \\ .\triangleleft. & : & ParList.X.n \times X \to ParList.X.(n+1) \\ .\mid. & : & ParList.X.n \times ParList.X.n \to ParList.X.(2*n) \\ .\#. & : & ParList.X.n \times ParList.X.n \to ParList.X.(2*n) \end{array}$$

Tipul $ParList.X$ are ca valori toate structurile $ParList$ cu elemente de tipul X. El poate fi partiţionat în următoarele subtipuri:

$$\begin{array}{lcl} Singleton.X & = & ParList.X.1 \\ EvenParList.X & = & \{\cup k : k \in Pos : ParList.X.(2*k)\} \\ OddParList.X & = & \{\cup k : k \in Pos : ParList.X.(2*k+1)\} \end{array}$$

Se observă că $PowerList$ este un subtip al lui $ParList$, corespunzător listelor a căror lungime este o putere a lui 2.

Funcţia lungime $length : ParList.X.n \to Pos$ este definită prin:

$$(\forall p : p \in ParList.X.n : length.p = n)\,.$$

Axiomele teoriei $PowerList$ se extind pentru a ajunge la o axiomatizare a algebrei $ParList$.

Axiomele $ParList$:

1. $[a] = [b] \equiv a = b$
2. $p|q = u|v \equiv p = u \wedge q = v$
3. $p\#q = u\#v \equiv p = u \wedge q = v$
4. $a \triangleright p = b \triangleright q \equiv a = b \wedge p = q$
5. $p \triangleleft a = q \triangleleft b \equiv a = b \wedge p = q$
6. $t \in ParList.X.1 \Rightarrow (\exists! a : a \in X : t = [a])$
7. $t \in ParList.X.(2*n) \Rightarrow (\exists! u, v : u, v \in ParList.X.n : t = u|v)$
8. $t \in ParList.X.(2*n) \Rightarrow (\exists! u, v : u, v \in ParList.X.n : t = u\#v)$
9. $t \in ParList.X.(n+1) \Rightarrow (\exists! a, v : a \in X \wedge v \in ParList.X.n : t = a \triangleright v)$
10. $t \in ParList.X.(n+1) \Rightarrow (\exists! a, v : a \in X \wedge v \in ParList.X.n : t = v \triangleleft a)$

Următoarele două axiome sunt reţinute din teoria $PowerList$:

11. $[a] \# [b] = [a] \mid [b]$
12. $(p|q) \# (u|v) = (p\#u) \mid (q\#v)$

Algebra $ParList$ mai conţine şi următoarele axiome:

13. $a \triangleright [b] = [a] \mid [b]$
14. $[a] \triangleleft b = [a] \mid [b]$
15. $a \triangleright (p \triangleleft b) = (a \triangleright p) \triangleleft b$
16. $a \triangleright p|q \triangleleft b = a \triangleright (p|q) \triangleleft b$
17. $a \triangleright p\#q \triangleleft b = a \triangleright (p\#q) \triangleleft b$
18. $a \triangleright (b \triangleright (p\#q)) = (a \triangleright p)\#(b \triangleright q)$

În teoria $PowerList$ rolul operatorilor # şi | putea fi interschimbat; în cazul $ParList$ acest lucru nu este posibil.

8.2.2 Un principiu al inducţiei pentru $ParList$

O structură $ParList$ cu elemente de tipul X poate fi descompusă unic într-o secvenţă ordonată a elementelor sale; acest lucru este posibil prin construirea unui *arbore de construcţie* pentru p. Folosim arborii de construcţie ca bază formală pentru principiul inducţiei pentru $ParList$ şi în definiţia funcţiilor peste $ParList$. Arborele de construcţie se poate forma după diferite reguli. O regulă posibilă ar fi folosirea operatorilor $\triangleleft$ şi $\triangleright$ doar pentru a construi elemente de tipul $OddParList$.X. Astfel, un arbore de construcţie pentru o structură $ParList$ p, va fi un arbore binar care are un număr de noduri terminale egal cu lungimea listei p (şi nodurile terminale sunt etichetate cu elementele listei), iar fiecare nod neterminal al său este etichetat cu una dintre operaţiile: $|, \#, \triangleleft, \triangleright$ şi cu o structură care se obţine prin aplicarea operatorului respectiv asupra structurilor din nodurile fii. Rădăcina va fi etichetată cu lista p. Un asemenea arbore de construcţie poate fi realizat datorită axiomelor $6 - 10$.

Fie $\Pi : ParList.X.n \rightarrow Bool$ predicatul al cărui adevăr dorim să fie stabilit pentru toate $ParList$ peste tipul X. Pentru a demonstra că predicatul $\Pi.p$ este adevărat, construim arborele de construcţie pentru p. Dacă putem dovedi că Π este adevărat pentru un nod dacă este adevărat pentru toate nodurile frunză ale subarborelui nodului p, atunci putem concluziona că Π este adevărat pentru rădăcina arborelui.

Corespunzător acestei reguli de construcţie avem următorul principiu de inducţiei pentru structuri $ParList$:

$$
\begin{array}{l}
\quad (\forall x : x \in X : \Pi.\,[x]) \\
\wedge\ (\quad (\forall p, q, n : p, q \in ParList.X.n \wedge n \in Pos : \Pi.p \wedge \Pi.q \Rightarrow \Pi.\,(p|q)) \\
\quad \vee (\forall p, q, n : p, q \in ParList.X.n \wedge n \in Pos : \Pi.p \wedge \Pi.q \Rightarrow \Pi.\,(p\#q)) \quad) \\
\wedge\ (\quad (\forall p, x, n : p \in EvenParList.X.n \wedge x \in X \wedge n \in Pos : \Pi.p \Rightarrow \Pi.\,(x \triangleright p)) \\
\quad \vee (\forall p, x, n : p \in EvenParList.X.n \wedge x \in X \wedge n \in Pos : \Pi.p \Rightarrow \Pi.\,(p \triangleleft x)) \quad) \\
\Rightarrow \\
\quad (\forall p, n : p \in ParList.X.n \wedge n \in Pos : \Pi.p)
\end{array}
$$

Conform acestui principiu, o demonstrare va conţine trei părţi: cazul de bază, pasul inductiv par şi pasul inductiv impar.

Este posibilă o generalizare prin folosirea unui operator de concatenare $\diamondsuit : ParList.X.n \times ParList.X.m \to ParList.X.(n+m)$ care este o generalizare a operatorului $|$ *(tie).* (Formalismul Bird-Meertens se bazează pe acest operator.)

Ilustrăm principiul inducţiei prin demonstrarea următoarei propoziţii:

Propoziţia 8.2 *Funcţia* $rev : ParList.X.n \to ParList.X.n$ *definită prin:*

$$\begin{aligned} &rev.[a] = [a] \\ &rev.(a \triangleright p) = rev.p \triangleleft a \\ &rev.(p \triangleleft a) = a \triangleright rev.p \\ &rev.(p\#q) = rev.q\#rev.p \end{aligned}$$

are proprietatea de idempotenţă:

$$rev.(rev.p) = p$$

Demonstraţie

Cazul de bază:

$$\begin{aligned} & rev.(rev.[a]) \\ =\ & \{\text{definiţia } rev\} \\ & rev.[a] \\ =\ & \{\text{definiţia } rev\} \\ & [a] \end{aligned}$$

Pasul inductiv par:

Presupunem relaţia adevărată pentru orice listă cu n elemente şi demonstrăm relaţia pentru listele cu $2n$ elemente.

$$\begin{aligned} & rev.(rev.(p\#q)) \\ =\ & \{\text{definiţia } rev\} \\ & rev.(rev.q\#rev.p) \\ =\ & \{\text{definiţia } rev\} \\ & rev.(rev.p)\,\#rev.(rev.q) \\ =\ & \{\text{ipoteza inducţiei}\} \\ & p\#q \end{aligned}$$

Pasul inductiv impar:

Presupunem relaţia adevărată pentru orice listă cu $2n$ elemente şi demonstrăm relaţia pentru listele cu $2n+1$ elemente.

$$\begin{aligned} & rev.(rev.(a \triangleright p\#q)) \\ =\ & \{\text{definiţia } rev\} \\ & rev.(rev.(p\#q) \triangleleft a) \\ =\ & \{\text{definiţia } rev\} \\ & a \triangleright rev.(rev.(p\#q)) \\ =\ & \{\text{ipoteza inducţiei }\} \\ & a \triangleright (p\#q) \end{aligned}$$

8.2.3 Operatori, relaţii şi funcţii

Fie $\otimes : X \times X \to X$ un operator binar definit pe tipul scalar X. Extindem operatorul prin definirea lui pe structuri $ParList$;

$\otimes : ParList.X.n \times ParList.X.n \to ParList.X.n$ este definit prin relaţiile:

$$
\begin{array}{l}
[a] \otimes [b] = [a \otimes b] \\
(p|q) \otimes (u|v) = (p \otimes u) \,|\, (q \otimes v) \quad \text{sau} \\
(p\#q) \otimes (u\#v) = (p \otimes u) \,\#\, (q \otimes v) \\
(a \triangleright p) \otimes (b \triangleright q) = (a \otimes b) \triangleright (p \otimes q) \quad \text{sau} \\
(p \triangleleft a) \otimes (q \triangleleft b) = (p \otimes q) \triangleleft (a \otimes b)
\end{array}
$$

Relaţiile peste tipurile scalare sunt extinse în acelaşi mod ca şi operatorii binari.

Fie Δ o relaţie peste tipul X, definită prin funcţia booleană $\Delta : X \times X \to Bool$ şi fie $p, q, u, v \in PowerList.X.n$ şi $x, y \in X$, definim relaţia extinsă prin:

$$
\begin{array}{l}
\Delta : ParList.X.n \times ParListX.n \to Bool \\
[x] \,\Delta\, [y] \equiv [x \Delta y] \\
(p\#q) \,\Delta\, (u\#v) \equiv (p \Delta u) \wedge (q \Delta v) \\
(a \triangleright p) \,\Delta\, (b \triangleright q) \equiv (a \Delta b) \wedge (p \Delta q)
\end{array}
$$

Deci două structuri $ParList$ sunt în relaţia Δ dacă şi numai dacă elementele de pe aceeaşi poziţie sunt în relaţie.

Funcţiile *map* şi *reduce* se definesc pe structuri $ParList$, în mod analog cu definirea lor pe structurile $PowerList$:

1. Funcţia
$$reduce : (X \times X \to X) \times ParList.X.n \to X$$
definită prin:
$$
\begin{array}{l}
reduce.\otimes.[a] = a \\
reduce.\otimes.(p|q) = reduce.\otimes.p \otimes reduce.\otimes.q \\
reduce.\otimes.(a \triangleright p) = a \otimes reduce.\otimes.p
\end{array}
$$
unde $\otimes : X \times X \to X$ este un operator asociativ.

   Funcţia *sum* se obţine prin instanţierea $sum = reduce.\oplus$.

2. Funcţia
$$
\begin{array}{l}
map : (X \to Z) \times ParList.X.n \to ParList.Z.n \\
map.f.[a] = [f.a] \\
map.f.(p\#q) = map.f.p \,\#\, map.f.q \\
map.f.(a \triangleright p) = f.a \triangleright map.f.p
\end{array}
$$
primeşte ca argumente o funcţie şi o structură $PowerList$ şi aplică funcţia fiecărui element din structură.

8.2.4 Aplicaţii

Pentru exemplificarea teoriei *Parlist*, prezentăm următoarele aplicaţii: sumă broadcast, calcularea diferenţelor divizate şi pe baza diferenţelor divizate, polinomul de interpolare Newton.

Deşi în cazul teoriei *ParList* paralelismul este introdus mai ales prin folosirea operatorilor $|$ şi $\#$, sunt cazuri în care doar folosirea operatorilor $\triangleright$ şi $\triangleleft$ conduce la paralelizări eficiente. Din această cauză definim şi un principiu al inducţiei alternativ bazat doar pe aceşti operatori:

$$\begin{array}{l} \quad (\forall x : x \in X : \Pi.\,[x]) \\ \wedge\ (\quad (\forall p, x, n : p \in ParList.X.n \wedge x \in X \wedge n \in Pos : \Pi.p \Rightarrow \Pi.\,(x \triangleright p)) \\ \quad \vee (\forall p, x, n : p \in ParList.X.n \wedge x \in X \wedge n \in Pos : \Pi.p \Rightarrow \Pi.\,(p \triangleleft x)) \quad) \\ \Rightarrow \\ \quad (\forall p, n : p \in ParList.X.n \wedge n \in Pos : \Pi.p) \end{array}$$

Funcţiile care sunt definite doar cu aceşti operatori se vor baza pe acest principiu al inducţiei.

Sumă broadcast

Funcţia $b\_sum : ParList.Y.n \to ParList.Y.n$ returnează o listă în care fiecare element este suma tuturor elementelor din lista argument. Y este un tip cu proprietatea că $(Y, +)$ este un semigrup.

Este necesar să definim funcţia $\langle a+\rangle : ParList.Y.n \to ParList.Y.n$ care returnează o listă obţinută prin adăugarea la fiecare element din lista argument a valorii a. Această funcţie, ar putea fi definită cu ajutorul funcţiei *map*: $< a+ > .p = map.(a+).p$.

$$\begin{array}{l} b\_sum.\,[a] = [a] \\ b\_sum.\,(a \triangleright p) = (a + first.t) \triangleright \langle a+\rangle .t, \quad \text{unde} \quad t = b\_sum.p \\ b\_sum.\,(p \# q) = t \# t, \qquad \text{unde} \quad t = b\_sum.\,(p + q) \end{array}$$

$$\begin{array}{l} \langle a+\rangle .\,[b] = [a + b] \\ \langle a+\rangle .\,(b \triangleright p) = (a + b) \triangleright \langle a+\rangle .p \\ \langle a+\rangle .\,(p|q) = \langle a+\rangle .p \mid \langle a+\rangle .q \end{array}$$

$$\begin{array}{l} first : ParList.Y.n \to Y \\ first.\,[a] = a \\ first.\,(p|q) = first.p \\ first.\,(a \triangleright p) = a \\ first.\,(p \triangleleft a) = first.p \end{array}$$

Fiecare descompunere necesită un pas paralel. Cel mai devaforabil caz apare atunci când $n = 2^k - 1, k > 0$; fiind necesară folosirea de $k - 1$ ori a deconstructorului $\triangleright$ şi de

$k-1$ ori deconstructorului #. Deci, în total $2k-2$ paşi paraleli, în comparaţie cu k paşi paraleli în cazul $n = 2^k$. O soluţie ar fi să se completeze lista iniţială cu elemente egale cu elementul unitate al semigrupului $(Y, +)$, dacă acesta există, pentru a se forma o *PowerList*.

Diferenţe divizate

Fie $X = \{x_i | x_i \in \mathbb{R}, i = 0, 1, ..., m\}$ o mulţime de puncte şi o funcţie $f : X \to \mathbb{R}$.

Dacă $r \in \mathbb{N}, r < m$ atunci mărimea

$$(Df)(x_r) = \frac{f(x_{r+1}) - f(x_r)}{x_{r+1} - x_r}$$

se numeşte *diferenţă divizată* a funcţiei f relativ la punctele x_r şi x_{r+1}.

Dacă $r, k \in N, r < m \wedge 1 \leq k \leq m - r$ atunci mărimea

$$(D^k f)(x_r) = \frac{(D^{k-1} f)(x_{r+1}) - (D^{k-1} f)(x_r)}{x_{r+k} - x_r}$$

se numeşte *diferenţă divizată de ordinul k* a funcţiei f pe punctul x_r şi se notează cu $(D^k f)(x_0) = [x_0, ..., x_k; f]$.

Funcţia $dif : ParList.Real.(m+1) \times ParList.Real.(m+1) \to Real$ calculează diferenţa divizată de ordin m. Argumentele sunt două liste *ParList* p şi q care conţin punctele diviziunii şi respectiv valorile funcţiei în punctele diviziunii: $p = [x_0, x_1, \ldots, x_m]$ şi $q = [f(x_0), f(x_1), \ldots, f(x_m)]$.

$dif.[a].[x] = x$
$dif.\,[a, b]\,[x\ y] = \frac{y-x}{b-a}$
$dif.\,(a \triangleright p \triangleleft b)\,.\,(x \triangleright q \triangleleft y) = (dif.\,(p \triangleleft b)\,.\,(q \triangleleft y) - dif.\,(a \triangleright p)\,.\,(x \triangleright q))\,/\,(b - a)$

Complexitatea-timp este $T.m = T.(m-1)+2 = O(m)$. Faptul că $dif.\,(p \triangleleft b)\,.\,(q \triangleleft y)$ şi $dif.(a \triangleright p).(x \triangleright q)$ se calculează în paralel, nu impune ca ele să fie calculate şi independent – calculele comune celor două pot fi realizate o singură dată (Figura 8.1).

Complexitatea-timp în cazul secvenţial este $T_s.m = O(m^2)$, deci acceleraţia este $S.n = O(m)$.

Polinomul de interpolare Newton

Fie o mulţime de puncte X şi o funcţie $f : [a, b] \to \mathbb{R}$.

Polinomul de interpolare Newton, corespunzător funcţiei $f : [a, b] \to \mathbb{R}$,
$(X = \{x_i | x_i \in \mathbb{R}, i = 0, 1, ..., m\}, X \subset [a, b])$
este definit prin formula:

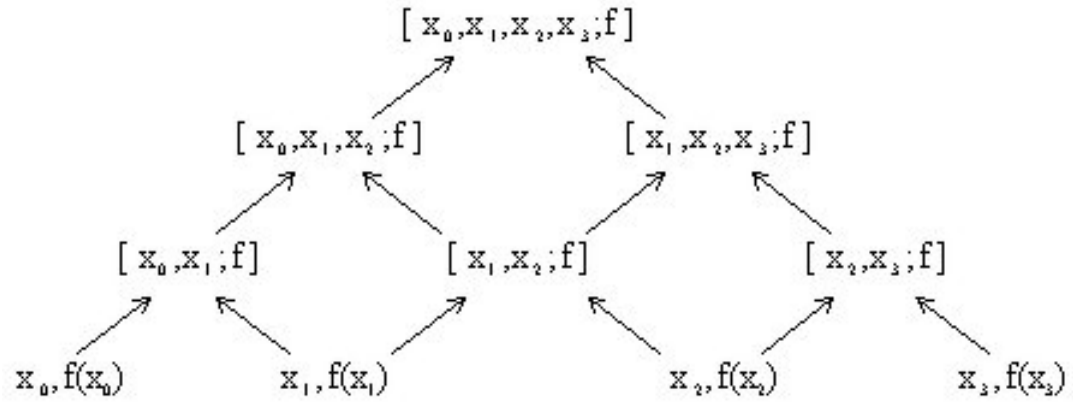

Figura 8.1: Reţeaua de calcul pentru $[x_0, x_1, x_2, x_3; f]$

$$(N_k f)(x) = (N_{k-1} f)(x) + (x - x_0) \ldots (x - x_{k-1}) [x_0, ..., x_k; f]$$
$$(N_0 f)(x) = f(x_0).$$

Folosind structuri *ParList* definim funcţia:

$$Newton : ParList.Real.(m+1) \times ParList.Real.(m+1) \times Real \to Real$$

Parametrii funcţiei sunt două liste *ParList* p şi q care conţin punctele diviziunii şi respectiv valorile funcţiei în punctele diviziunii: $p = [x_0, x_1, \ldots, x_m]$ şi
$q = [f(x_0), f(x_1), \ldots, f(x_m)]$ şi punctul $x \in [a, b]$ în care se aproximează funcţia f.

Funcţia *Newton* se defineşte prin:

$$Newton.[a].[b].x = b$$
$$Newton.(p \lhd a).(q \lhd b).x = Newton.p.q.x + prod.pp * dif.(p \lhd a).(q \lhd b),$$

---

unde $pp = \langle x- \rangle .p$, $\langle x- \rangle .p$ este funcţia definită la aplicaţia suma broadcast, iar funcţia *prod* este aplicaţia funcţiei *reduce* cu operatorul produs.

Analiza expresiei funcţiei *Newton* conduce la concluzia că funcţia *prod* trebuie să fie calculată la fiecare etapă, ceea ce ar fi ineficient. Expresia funcţiei poate fi transformată.

Notăm $DP.p.q.x = prod.(< x- > .p) * dif.p.q$ şi căutam o definiţie recursivă pentru DP:

Cazul de bază:

$$\begin{array}{ll} & DP.[a,b].[c,d].x \\ = & \{\text{calcul}\} \\ & (x-a)(x-a)(d-c)/(b-a) \end{array}$$

Pasul inductiv:

$$
\begin{array}{ll}
 & DP.\,(a \triangleright p \triangleleft b)\,.\,(c \triangleright q \triangleleft d) \\
= & \{\text{substituţie, definiţia funcţiei } dif\} \\
 & prod.(\langle x{-}\rangle\,.(a \triangleright p \triangleleft b)) * ((dif.\,(p \triangleleft b)\,.\,(q \triangleleft d) - dif.\,(a \triangleright p)\,.\,(c \triangleright q))\,/\,(b-a)) \\
= & \{\text{calcul}\} \\
 & DP.(p \triangleleft b).(q \triangleleft d).x * (x-a)/(b-a) - DP.(a \triangleright p).(c \triangleright q).x * (x-b)/(b-a)
\end{array}
$$

Prin urmare, definiţia completă a funcţiei DP este:

$$
\begin{array}{l}
DP.[a,b].[c,d].x = (x-a)(x-a)(d-c)/(b-a) \\
DP.\,(a \triangleright p \triangleleft b)\,.\,(c \triangleright q \triangleleft d) = \\
\qquad\qquad (DP.(p \triangleleft b).(q \triangleleft d).x * (x-a) - DP.(a \triangleright p).(c \triangleright q).x * (x-b))/(b-a)
\end{array}
$$

Folosind funcţia DP redefinim funcţia $Newton$:

$$
\begin{array}{l}
Newton.\,[a]\,.\,[b]\,.x = b \\
Newton.\,(p \triangleleft a)\,.\,(q \triangleleft b)\,.x = Newton.p.q.x + DP.\,(p \triangleleft a)\,.\,(q \triangleleft b)\,.x/(x-a),
\end{array}
$$

Complexitatea funcţiei DP este $O(m)$ şi prin urmare, dacă funcţiile $Newton.p.q.x$ şi $DP.\,(p \triangleleft a)\,.\,(q \triangleleft b)\,.x$ pot fi calculate în paralel atunci complexitatea funcţiei $Newton$ este $O(m)$. Complexitatea-timp în cazul secvenţial este $O(m^2)$.

8.3 Structuri de date $PList$

Structurile $PList$ sunt o generalizare a structurilor de date $PowerList$. Ele sunt construite cu ajutorul operatorilor n-ari $\#$ şi $|$; pentru n pozitiv operatorul $|$ n-ar concatenează n structuri $PList$ similare. Dacă notaţia $PowerList$ este limitată la baza 2, notaţia $PList$ este mult mai generală, ea permiţând folosirea în specificaţii a unor baze compuse şi facilitează demonstrarea algebrică a acestor specificaţii.

În acestă secţiune, vom folosi pentru notarea cuantificării ordonate, parantezele drepte; expresia $\left[\#i : i \in \bar{n} : p.i\right]$ exprimă aplicarea operatorului n-ar $\#$ asupra structurilor $PList$ $p.i$ în ordine. Domeniul $i \in \bar{n}$ exprimă faptul că termenii expresiei sunt scrişi de la 0 la $n-1$ în ordine.

8.3.1 Definiţii

O structură $PList$ este o structură de date liniară ale cărei elemente sunt toate de acelaşi tip, fie scalari cu acelaşi tip de bază, fie (recursiv) structuri $PList$ care sunt similare.

Definim lungimea unei structuri $PList$ prin:

$$
\begin{array}{l}
length : PList.X.n \to Pos \\
(\forall p : p \in PList.X.n : length.p = n)\,.
\end{array}
$$

Două structuri *PListe* sunt *similare* dacă au aceeaşi lungime şi elementele constitutive sunt similare; doi scalari sunt similari dacă aparţin aceluiaşi tip de bază.

Cea mai simplă structură $PList$ este numită *singleton* şi are un singur element; un singleton care-l conţine pe x se notează $[x]$.

Fie $p.i$, unde $0 \leq i < n \wedge n \in Pos$ un grup de n structuri $PList$, similare, cu lungimea m. Definim structurile u, v astfel:

- $u = [|i : i \in \bar{n} : p.i]$,
  u se obţine prin concatenarea elementelor listelor $p.i$ în ordine, astfel încât al j-lea element din $p.i$ apare ca elementul $i * m + j$ în u.

- $v = [\#i : i \in \bar{n} : p.i]$,
  conţine interclasarea elementelor listelor $p.i$ în ordine, astfel încât al j-lea element din $p.i$ apare ca elementul $i + m * j$ în v.

Formal, constructorii structurii $PList$ au următoarele tipuri:

$$\begin{array}{lcl} [.] & : & X \to PList.X.1 \\ \left[|\ i : i \in \dot{n} : .\ \right] & : & (PList.X.m)^n \to PList.X.(n * m) \\ \left[\#\ i : i \in \dot{n} : .\ \right] & : & (PList.X.m)^n \to PList.X.(n * m) \end{array}$$

Axiomele algebrei $PList$

Fie $p.i.j \in PList.X.k$, pentru $0 \leq i < n$ şi $0 \leq j < m$ (unde $n, m \in Pos$) şi fie $x.i, a, b \in X$. Următoarele axiome definesc algebra *PList:*

1. $(\forall t : t \in PList.X.1 : (\exists! a : a \in X : t = [a]))$
2. $\Big(\forall t : t \in PList.X.(k * n) : \big(\exists! u : u \in (PList.X.n)^k : t = [\#i : i \in \bar{n} : u.i]\big)\Big)$
3. $\Big(\forall t : t \in PList.X.(k * n) : \big(\exists! u : u \in (PList.X.n)^k : t = [|i : i \in \bar{n} : u.i]\big)\Big)$
4. $[a] = [b] \equiv a = b$
5. $[\#i : i \in \bar{n} : u.i] = [\#i : i \in \bar{n} : v.i] \equiv (\forall i : 0 \leq i < n : u.i = v.i)$
6. $[|i : i \in \bar{n} : u.i] = [|i : i \in \bar{n} : v.i] \equiv (\forall i : 0 \leq i < n : u.i = v.i)$
7. $[\#i : i \in \bar{n} : [x.i]] = [|i : i \in \bar{n} : [x.i]]$
8. $[\#i : i \in \bar{n} : [|j : j \in \bar{m} : p.i.j]] = [|j : j \in \bar{n} : [\#i : i \in \bar{m} : p.i.j]]$

Liste liniare

Folosim funcţia de tip:

$$List : Type \to Type$$

pentru a construi tipul listelor liniare peste un tip de date. Lista vidă este notată cu []. Pentru un element $x \in X$ şi o listă $l \in List.X$ putem forma lista $x \triangleright l$ care are primul element x şi următoarele elemente din lista l şi analog $l \triangleleft x$ care are ultimul element x şi primele elemente din lista l. Pentru lista care conţine un element folosim notaţia $[x]$. Pentru că vom folosi în special $List.Pos$, introducem notaţia $PosList = List.Pos$.

Putem defini funcţii pe aceste structuri de date. Funcţia

$$prod : PosList \to Pos$$

definită prin:

$$prod.[] = 1$$
$$prod.(x \triangleright l) = x * prod.l$$

calculează produsul elementelor conţinute într-o *PosList.*

8.3.2 Un principiu al inducţiei pentru $PList$

Funcţiile peste $PList$ sunt definite prin inducţie structurală peste două structuri $PosList$ şi $PList$, perechile admise fiind specificate printr-o funcţie specială numită *valid.*

Prima listă $PosList$ specifică aritătile, iar a doua, lista $PList$, este argumentul propriu zis. Legătura dintre ele este făcută de către funcţia *valid* care defineşte de fapt, un predicat ce specifică unde este definită funcţia:

$$valid : ((PosList \times Plist) \to X) \times PosList \times PList \to Bool$$

Se poate stabili un principiu al inducţiei fie bazat pe structurile $PList$, fie pe listele $PosList$. Vom defini un principiu bazat pe $PosList$.

Fie

$$\Pi : PosList \times PList.X.n \to Bool$$

predicatul al cărui adevăr vrem să-l stabilim pentru toate perechile din $PosList \times PList$ peste X, unde aplicarea funcţiei este definită.

Pentru a demonstra că predicatul Π este adevărat putem folosi următorul principiu al inducţiei:

$$\begin{array}{ll} & (\forall p : p \in PList.X.n \wedge valid.[].p : \Pi.[].p) \\ \wedge(& (\forall x, p, q, l : x \in Pos \wedge p \in PList.X.n \wedge q \in PList.X.m \wedge l \in PosList : \\ & \quad (valid.\Pi.l.p \Rightarrow \Pi.l.p) \Rightarrow (valid.\Pi.(x \triangleright l).q \Rightarrow \Pi.(x \triangleright l).q)\ ) \\ & \vee (\forall x, p, q, l : x \in Pos \wedge p \in PList.X.n \wedge q \in PList.X.m \wedge l \in PosList : \\ & \quad (valid.\Pi.l.p \Rightarrow \Pi.l.p) \Rightarrow (valid.\Pi.(l \triangleleft x).q \Rightarrow \Pi.(l \triangleleft x).q)\ ) \quad) \\ \Rightarrow & \quad (\forall p, l : p \in PList.X.n \wedge l \in PosList : valid.\Pi.l.p \Rightarrow \Pi.l.p) \end{array}$$

Operatorii binari şi relaţiile pot fi extinse la structuri $PList$, folosindu-se acest principiu (în mod similar cu modul de extindere pentru celelalte structuri).

Funcţii definite pe structuri $PList$

1. Suma elementelor unei *PList:*

$$sum : PosList \times PList.X.n \to X$$
$$valid.sum.l.p \equiv prod.l = length.p$$
$$sum.[].[a] = a$$
$$sum.(x \triangleright l).[|i : i \in \bar{x} : p.i] = (+i : 0 \leq i < x : sum.l.(p.i))$$

2. Funcţia *map* definită pe structuri *PList:*

$$map : (X \to Y) \times PosList \times PList.X.n \to PList.Y.n$$
$$valid.map.l.p \equiv prod.l = length.p$$
$$map.f.[].[x] = [f.x]$$
$$map.f.(x \triangleright l).[|i : i \in \overline{x} : p.i] = [|i : i \in \overline{x} : map.f.l.(p.i)]$$

Un caz special al funcţiei *map* se obţine dacă funcţia f care se distribuie, realizează înmulţirea fiecărui element al listei cu un element dat (z):

$$< z* > .l.p = map.(z*).l.p.$$

3. Funcţia de permutare *inv.*

Funcţia *inv* permută un element a cărui poziţie poate fi scrisă ca un şir de cifre într-o notaţie cu baza compusă, pe o poziţie care poate fi scrisă ca inversa şirului iniţial:

$$valid.inv.l.p \equiv prod.l = length.p$$
$$inv.\,[]\,.\,[a] = [a]$$
$$inv.\,(x \triangleright l)\,.\,[|i : i \in \bar{x} : p.i] = [\#i : i \in \bar{x} : inv.l.\,(p.i)]$$

Propoziţia 8.3

$$inv.\,(l \triangleleft x)\,.\,[\#i : i \in \bar{x} : p.i] = [|i : i \in \bar{x} : inv.l.\,(p.i)]$$

Demonstraţie: Folosim principiul inducţiei definit anterior.
Cazul de bază:

$$\begin{aligned}
& inv.\,[x]\,.\,[\#i : i \in \bar{x} : [a.i]] \\
= & \{\text{axioma 7}\} \\
& inv.\,[x]\,.\,[|i : i \in \bar{x} : [a.i]] \\
= & \{\text{definiţia } inv\} \\
& [\#i : i \in \bar{x} : inv.\,[]\,.\,[a.i]] \\
= & \{\text{definiţia } inv\} \\
& [\#i : i \in \bar{x} : [a.i]] \\
= & \{\text{axioma 7}\} \\
& [|i : i \in \bar{x} : [a.i]] \\
= & \{\text{definiţia } inv\} \\
& [|i : i \in \bar{x} : inv.\,[]\,.\,[a.i]]
\end{aligned}$$

Pasul inductiv:

$$
\begin{array}{ll}
 & inv.((y \triangleright l) \triangleleft x) \,.\, \big[\#i : i \in \bar{x} : \big[|j : j \in \bar{y} : p.i.j\big]\big] \\
= & \{\text{axioma 8}\} \\
 & inv.((y \triangleright l) \triangleleft x) \,.\, \big[|j : j \in \bar{y} : [\#i : i \in \bar{x} : p.i.j]\big] \\
= & \{\text{definiţia } inv\} \\
 & \big[\#j : j \in \bar{y} : inv.(l \triangleleft x) \,.\, [\#i : i \in \bar{x} : p.i.j]\big] \\
= & \ \{\text{ipoteza inducţiei}\} \\
 & \big[\#j : j \in \bar{y} : [|i : i \in \bar{x} : inv.l.p.i.j]\big] \\
= & \{\text{axioma 8}\} \\
 & [|i : i \in \bar{x} : [\#j : j \in \bar{x} : inv.l.p.i.j]] \\
= & \{\text{definiţia } inv\} \\
 & \big[|i : i \in \bar{x} : inv.(y \triangleright l) \,.\, \big[|j : j \in \bar{y} : p.i.j\big]\big]
\end{array}
$$

8.3.3 Aplicaţii

Secţiunea de aplicaţii pentru $PList$ conţine definirea unei funcţii pentru calcularea valorilor unui polinom într-o mulţime de puncte date, folosind structurile $PList$; este prezentată de asemenea o aplicaţie pentru integrare numerică, folosind de această dată formula repetată a dreptunghiului definită recursiv.

Valoarea unui polinom

Extindem funcţia vp de calculare a valorii unui polinom, definită pentru structuri $PowerList$ la structuri $PList$.

Funcţia este definită pe două structuri $PList$ corespunzătoare coeficienţilor polinomului (p) şi punctelor (a) în care se calculează valoarile acestuia. Pentru fiecare dintre acestea avem şi un parametru de tip $PosList$:

$$vp : PosList \times PList.X.n \times PosList \times PList.X.m \to PList.X.m$$

$$
\begin{array}{l}
valid.vp.lx.p.ly.a \equiv prod.lx = length.p \wedge prod.ly = length.a \\
vp.[].[p].[].[a] = p \\
vp.l.p.(y \triangleright ly).[|i : i \in \overline{y} : a.i] = [|i : i \in \overline{y} : vp.l.p.ly.(a.i)] \\
vp.(x \triangleright l).[\#i : i \in \overline{x} : p.i].ly.a = (+i : i \in \overline{x} : a^i * vp.l.(p.i).ly.a^x)
\end{array}
$$

Definiţia funcţiei vp se bazează pe ideea folosită la structurile $PowerList$, de a împărţi lista coeficienţilor polinomului în subliste, care se consideră a fi coeficienţii unor polinoame de grad $n/x - 1$ (gradul polinomului iniţial este $n - 1$). (Dacă $x = 2$ se folosesc două subliste.)

Integrare numerică folosind formula repetată a dreptunghiului

Pentru o funcţie $f : [a, b] \to R$, integrala $I = \int_a^b f(x)\,dx$ poate fi aproximată folosind formula dreptunghiului:

$$Q_{D_0}(f) = f(x_0)$$
$$Q_{D_k}(f) = \tfrac{1}{3} Q_{D_{k-1}}(f) + h \sum_{i=0}^{2m-1} f(x_i), \text{ unde } h = \tfrac{b-a}{3^k},\ m = 3^{k-1} \text{ şi } k = 1, 2, \ldots$$

Punctul $x_0 = a + h/2$, iar pentru $i > 0$ punctele x_i se obţin după formula

$$x_i = \begin{cases} x_{i-1} + 2h, & \text{pentru } i \text{ impar} \\ x_{i-1} + h, & \text{pentru } i \text{ par} \end{cases}.$$

Şirul $Q_{D_k}(f)$ converge la valoarea I a integralei funcţiei f pe intervalul $[a, b]$.

Considerăm ca date iniţiale lista valorilor funcţiei în punctele diviziunii: $[x_0, x_1, ..., x_{3^n-1}]$; deci $p = [f(x_0), f(x_1), ..., f(x_{3^n-1})]$.

Observăm că 3^{n-1} puncte sunt folosite pentru calculul valorii $Q_{D_{n-1}}(f)$, iar $2 * 3^{n-1}$ puncte intervin în calculul termenului al doilea al sumei care ne dă valoarea $Q_{D_n}(f)$. În concluzie, definirea funcţiei care calculează $Q_{D_n}(f)$ va folosi în definiţia structurală operatorul #.

Funcţia $drept : Real \times PosList \times PList.Real.m \to Real(m = 3^n)$ are primul argument $hk = \frac{b-a}{3^n}$ pasul diviziunii, al doilea este o listă formată din n valori egale cu 3, iar al treilea este lista valorilor funcţiei. Definim funcţia *drept* prin:

$$valid.drept.l.p \equiv prod.l = length.p$$
$$drept.hk.[\,].[x] = hk * x$$
$$drept.hk.(3 \triangleright l).\left[\#i : i \in \bar{3} : p_i\right] = \tfrac{1}{3} * drept.(3 * hk).l.p_1 + hk * sum.(2 \triangleright l).(p_0 \# p_2)$$

În complexitatea algoritmului intervine ca şi factor $\log_3 m$ (numărul de paşi paraleli, $m = length.p$). Fiecare pas paralel trebuie să calculeze şi o sumă de $2k$, $k = m/3, m/3^2, \ldots, 1$ elemente; aceasta poate fi calculată într-un timp de $\log_3 k + 1$.

Complexitatea este $O(\log_3 m)$, dacă cei doi termeni din definiţia funcţiei pot fi calculaţi în paralel. Complexitatea în cazul secvenţial este $O(m)$.

8.4 Transformarea Fourier rapidă

Această secţiune prezintă algoritmul pentru *transformarea Fourier rapidă,* printr-o specificare care foloseşte structurile de date prezentate anterior.

Prin transformarea Fourier discretă se trece de la reprezentarea convenţională a unui polinom, prin suita de coeficienţi, la o altă reprezentare printr-o suită de valori care reprezintă valori ale polinomului în puncte distincte.

Această transformare este un instrument important, folosit în multe aplicaţii ştiinţifice. Permite o înmulţire rapidă a polinoamelor, calcule cu mare precizie, sinteza imaginilor precum şi alte aplicaţii.

Prin această transformare, reprezentarea unui polinom de grad $n-1$ şi cu coeficienţii $(a_i, 0 \leq i < n)$ se face prin valorile polinomului în rădăcinile de ordin n ale unităţii $(w_j, 0 \leq j < n)$.

În funcţie de valorile lui n se disting trei cazuri [134], care vor fi prezentate în continuare.

În toate cele trei cazuri vom folosi o funcţie scalară $root : Nat \rightarrow Com$, care aplicată lui n returnează următoarea rădăcină de ordin n a unităţii:

$$root.n = e^{\frac{2\pi i}{n}}.$$

8.4.1 Cazul n=2^k

Formula de calcul a valorii polinomului în punctul w_j este:

$$f\left(w_j\right) = \sum_{m=0}^{2^{k-1}-1} a_{2m} * e^{\frac{2\pi ijm}{2^{k-1}}} + e^{\frac{2\pi ij}{2^k}} \sum_{m=0}^{2^{k-1}-1} a_{2m+1} * e^{\frac{2\pi ijm}{2^{k-1}}}, 0 \leq j < n$$

Complexitatea unui algoritm recursiv secvenţial este $T_s.n = O(n \log_2 n)$.

Pentru specificarea unui algoritm paralel, în acest caz, se pot folosi structuri de tip $PowerList$.

Pentru a ajunge la algoritmul paralel se porneşte de la funcţia:

$$fft : PowerList.Com.k \rightarrow PowerList.Com.k.$$

Argumentul de tip *PowerList* conţine lista coeficienţilor polinomului şi funcţia este definită astfel:

$$fft.p = vp.p.u,$$

unde $u = powers.z.p, z = root.(length.p)$ şi am folosit funcţia vp definită în secţiunea 8.1.6.

Funcţia $powers$ are un argument număr complex şi un argument listă; returnează o listă de lungime egală cu lungimea listei argument şi care conţine puterile parametrului complex, în ordine, de la 0 la lungimea listei minus 1:

$$\begin{aligned} &powers : Com \times PowerList.X.n \rightarrow PowerList.Com.n \\ &powers.x.\,[a] = [x^0] \\ &powers.x.\,(p \# q) = powers.x^2.p \;\#\; \langle x* \rangle\,.\,(powers.x^2.p) \end{aligned}$$

Funcţia $< x* >$ este definită folosind funcţia map: $< x* > .p = map.(x*).p$.

Derivarea algoritmului pentru transformarea Fourier, presupune găsirea unei alte expresii pentru funcţia fft – mai eficientă şi aceasta se realizează cu ajutorul inducţiei structurale. Notăm $W.z.p = powers.z.p$, pentru care sunt valabile următoarele două proprietăţi, care se bazează pe proprietăţile rădăcinilor unităţii:

$$W^2.z.(p\#q) = W.z^2.p|W.z^2.q$$
$$W.z.(p\#q) = W.z.p| - W.z.q$$

Cazul de bază:

$$\begin{array}{ll} & fft.[x] \\ = & \{\text{definiţia } fft\} \\ & vp.[x].z \\ = & \{\text{definiţia } vp\} \\ & [x] \end{array}$$

Pasul inductiv: Considerăm operatorii extinşi ai operatorilor binari $+, -, *$ şi folosim aceleaşi simboluri.

$$\begin{array}{ll} & fft.(p\#q) \\ = & \{\text{definiţia } fft\} \\ & vp.(p\#q).(W.z.(p\#q)) \\ = & \{\text{definiţia } vp\} \\ & vp.p.(W.z.(p\#q))^2 + (W.z.(p\#q)) * (vp.q.(W.z.(p\#q))^2) \\ = & \{\text{proprietatea funcţiei } W\} \\ & vp.p.(W.z^2.p|W.z^2.q) + W.z.(p\#q) * vp.p.(W.z^2.p|W.z^2.q) \\ = & \{ \text{ definiţia } vp\} \\ & vp.p.(W.z^2.p)|vp.q.(W.z^2.q) + (W.z.(p\#q)) * (vp.p.(W.z^2.p)|vp.q.(W.z^2.q)) \\ = & \{\text{definiţia } fft\} \\ & fft.p|fft.q + (W.z.(p\#q)) * (fft.p|fft.q) \\ = & \{\text{proprietatea funcţiei } W\} \\ & fft.p|fft.q + (W.z.p| - W.z.q)) * (fft.p|fft.q) \\ = & \{\text{definiţia operatorului multiplicativ }\} \\ & fft.p|fft.q + (W.z.p * fft.p)|(-W.z.p) * fft.q) \\ = & \{ \text{ definiţia operatorului aditiv }\} \\ & (fft.p + W.z.p * fft.q)|(fft.p - W.z.p * fft.q) \end{array}$$

În consecinţă noua definire a funcţiei este următoarea:

$$fft.\,[a] = [a]$$
$$fft.\,(p\#q) = (r + u * s)\,|\,(r - u * s)$$

$$r = fft.p$$
$$s = fft.q$$
$$u = powers.z.p$$
$$z = root.\,(length.\,(p\#q))$$

Complexitatea pentru algoritmul paralel *fft* este $T.n = O(log_2 n)$.

8.4.2 Cazul n prim

În acest caz este necesară calcularea directă a valorilor polinomului şi se folosesc structurile $ParList$.

Functia $fft : ParList.Com.n \to ParList.Com.n$ este definită prin:

$$fft.p = vp.p.(powers.(root.(length.p)).p)$$

$$vp : ParList.X.n \times ParList.X.m \to ParList.X.m$$

$$vp.[a].[z] = a$$
$$vp.p.(z \triangleright u) = vp.p.[z] \ \triangleright \ vp.u.p$$
$$vp.p.(u \mid v) = vp.p.u \mid vp.p.v$$
$$vp.(p\#q).z = vp.p.z^2 + z * vp.q.z^2$$
$$vp.(a \triangleright p).z = a + z * vp.p.z$$

$$powers : Com \times ParList.X.n \to ParList.Com.n$$

$$powers.x.\,[a] = [x^0]$$
$$powers.x.\,(a \triangleright p) = [x^0] \ \triangleright \ \langle x* \rangle \,.powers.z.p$$
$$powers.x.\,(p\#q) = powers.x^2.p \ \# \ \langle x* \rangle \,.\,(powers.x^2.p)$$

Funcţia *vp*, care calculează valoarea unui polinom într-un punct dat, este extinsă pe structuri $ParList$. Ea primeşte ca parametrii lista rădăcinilor unităţii şi lista coeficienţilor polinomului.

Se foloseşte funcţia *powers* care calculează puterile unui parametru scalar. Este făcută extinderea funcţiei *powers* de la structuri $PowerList$ la structuri $ParList$.

Se observă că s-a folosit definiţia de la cazul întâi, pentru cazul în care $p \in EvenParList$.

Complexitatea maximă se obţine atunci când $n = 2^k - 1$, când sunt necesari $2k - 2$ paşi, iar cea minimă este egală cu cea a cazului $n = 2^k$. În consecinţă $T.n = O(\log_2 n)$, dacă cele n valori pot fi calculate în paralel.

8.4.3 Cazul $n = r_1 r_2 \ldots r_p$

În cazul în care n nu este o putere a lui 2, dar este un produs a două numere r_1 şi r_2 algoritmul prezentat în primul caz poate fi generalizat astfel:

$$f\,(w_j) = \sum_{k=0}^{r_1-1} \left\{ \sum_{t=0}^{r_2-1} a_{t\,r_1+k}\, e^{\frac{2\pi i j t}{r_2}} \right\} e^{\frac{2\pi i\, j\, k}{n}}, 0 \le j < n$$

Suma interioară reprezintă valoarea în punctul $w_{j\%r_2}$ a polinomului de grad $r_2 - 1$ cu coeficienţii $\left\{a_k, a_{k+r_1}, ..., a_{k+r_1(r_2-1)}\right\}$, k = 0,1,..r_1-1, care este calculată de transformarea Fourier pentru acest polinom.

Concluzia este că se poate defini un algoritm recursiv care combină r_1 transformate Fourier.

Teorema 8.1 *Cea mai bună factorizare $n = r_1 r_2$, pentru algoritmul FFT (din punctul de vedere al complexităţii) este alegerea lui r_1 printre divizorii primi ai lui n.*

Demonstraţia poate fi găsită în [172].

Ca urmare, pentru specificarea algoritmului paralel, considerăm descompunerea în factori primi a lui $n(n = r_1 r_2 \ldots r_p)$ şi folosim structurile *PList.*

$$fft : PosList \times PList.Com.n \rightarrow PList.Com.n$$

Lista *PosList* este formată din factorii primi ai lui $n : [r_1, r_2, \ldots, r_p]$.

Pentru a ajunge la o definiţie a funcţiei fft, pornim derivarea ca şi în cazul $n = 2^k$, de la definiţia transformatei Fourier: $fft.l.p = vp.l.p.l.w$, unde $w = powers.z.l$.

Am folosit funcţia *vp* definită pe structuri *PList* şi funcţia *powers* care calculează lista care conţine puterile lui z, de la 0 până la $prod.l - 1$. Funcţia *powers* are următoarea definiţie:

$$\begin{aligned} &powers : Com \times PosList \rightarrow PList.Com.(prod.l) \\ &powers.z.[] = [z^0] \\ &powers.z.(x \triangleright l) = [\#i : i \in \overline{x} :< (z^i) * > .q] \end{aligned}$$

unde $q = powers.(z^x).l$.

Vom folosi notaţia $W.z.l = powers.z.l$. Pentru funcţia $powers.z.l$ avem următoarele proprietăţi:

$$\begin{aligned} &W.z.(x \triangleright l) = [|i : i \in \overline{x} :< z^{\frac{n}{x} i} * > .(W.z.l)] \\ &(W.z.(x \triangleright l))^x = [|i : i \in \overline{x} : W.z^x.l] \\ &(W.z.l)^i = W.(z^i).l \end{aligned}$$

unde $n = x * prod.l$.

Derivarea funcţiei fft:

Cazul de bază:

$$\begin{aligned} & fft.[x].[\#i : i \in \overline{x} : [p.i]] \\ = \; & \{\text{definiţia } fft\} \\ & vp.[x].[\#i : i \in \overline{x} : [p.i]].[x].[\#i : i \in \overline{x} : [z^i]] \\ = \; & \{\text{calcul}\} \\ & [|j : j \in \overline{x} : (+i : i \in \overline{x} : p.i * z^{(i*j)})] \end{aligned}$$

Pasul inductiv:

$$
\begin{array}{ll}
 & fft.(x \triangleright l).[\#i : i \in \overline{x} : p.i] \\
= & \{\text{definiţia } fft\} \\
 & vp.(x \triangleright l).[\#i : i \in \overline{x} : p.i].(x \triangleright l).(W.z.(x \triangleright l)) \\
= & \{\text{definiţia } vp\} \\
 & (+i : i \in \overline{x} : (W.z.(x \triangleright l))^i * vp.l.(p.i).(x \triangleright l).(W.z.(x \triangleright l))^x) \\
= & \{\text{proprietatea funcţiei } W.z.l, \text{definiţia } vp\} \\
 & (+i : i \in \overline{x} : (W.z.(x \triangleright l))^i * [|j : j \in \overline{x} : vp.l.(p.i).l.(W.z^x.l)|]) \\
= & \{\text{definiţia } fft\} \\
 & (+i : i \in \overline{x} : (W.z.(x \triangleright l))^i * [|j : j \in \overline{x} : fft.l.(p.i)|]) \\
= & \{\text{proprietatea funcţiei } W.z.l\} \\
 & (+i : i \in \overline{x} : [|j : j \in \overline{x} :< z^{\frac{n}{x}j} * > W.z.l]^i * [|j : j \in \overline{x} : fft.l.(p.i)|]) \\
= & \{\text{calcul}\} \\
 & (+i : i \in \overline{x} : [|j : j \in \overline{x} :< z^{\frac{n}{x}ij} * > (W.z.l)^i] * [|j : j \in \overline{x} : fft.l.(p.i)|]) \\
= & \{\text{proprietatea funcţiei } W.z.l\} \\
 & (+i : i \in \overline{x} : [|j : j \in \overline{x} :< z^{\frac{n}{x}ij} * > W.(z^i).l * fft.l.(p.i)]
\end{array}
$$

Prin urmare, definiţia funcţiei fft este:

$$
\begin{array}{l}
valid.fft.l.p \equiv (prod.l = length.p) \\
fft.[x][\#i : i \in \overline{x} : [a.i]] = [|j : j \in \overline{x} : (+i : i \in \overline{x} : a.i * z^{(i*j)})|] \\
fft.(x \triangleright l).[\#i : i \in \overline{x} : p.i] = [|j : j \in \overline{x} : (+i : i \in \overline{x} : r.i * u.i.j)|]
\end{array}
$$

$$
\begin{array}{l}
r.i = fft.l.(p.i) \\
u.i.j =< z^{(ij*\frac{n}{x})} * > .powers.(z^i).l \\
z = root.n \\
n = length.p
\end{array}
$$

Dacă lista $PosList$ are un singur element, cazul se reduce la aplicarea funcţiei fft pe structuri $ParList$ (structura $PList$ corespunzătoare putând fi tratată ca şi o $ParList$):

$$fft.\,[x]\,[\#i : i \in \bar{x} : [a.i]] = fft_{|ParList}.[\#i : i \in \bar{x} : [a.i]]$$

care este mai eficientă.

Pentru cazul $x = 2$ se poate observa că se ajunge la algoritmul specificat cu $PowerList$:

$$
\begin{array}{l}
\big[|\ j : j \in \bar{2} : \big[+i : i \in \bar{2} : r.i * u.i.j\big]\big] = \\
(r.0 * u.0.0 + r.1 * u.1.0)\,|\,(r.0 * u.0.1 + r.1 * u.1.1) = \\
(r.0 + r.1 * powers.z.l)\,|\,\big(r.0 + r.1 * \big\langle \exp .z.\tfrac{n}{2} * \big\rangle .powers.z.l\big) = \\
(r.0 + r.1 * powers.z.l)\,|\,(r.0 - r.1 * powers.z.l)
\end{array}
$$

Algoritmul pentru transformarea Fourier poate fi aplicat simultan cu descompunerea în factori prim al lui n. Dacă se ajunge la divizori mari se poate aplica algoritmul corespunzător cazului n prim, pentru simplificarea calculelor.

Complexitatea algoritmului în acest caz depinde de factorii primi ai descompunerii lui n şi de numărul acestora m. Dacă considerăm că toţi factorii primi sunt mai mici decât un număr M atunci complexitatea este $O(m)$ şi constanta de multiplicitate depinde de M. Dacă, de exemplu, $n = 3^k$ atunci $T.n = O(\log_3 n)$.

În cazul secvenţial se obţine o complexitate $T_m.n = n(a_1(p_1 - 1) + \ldots + a_m(p_m - 1))$, dacă $n = p_1^{a_1} \ldots p_m^{a_m}$ [172].

8.5 Structuri de date n-dimensionale

Structurile de date *PowerList, ParList* şi *PList* sunt structuri liniare. Pentru reprezentarea tablourilor multidimensionale se pot folosi aceste structuri cu elemente care la rândul lor sunt structuri *PowerList, ParList* şi *PList* . Dar în acest mod anumiţi algoritmi, cum este de exemplu transpoziţia matricelor, sunt dificil de specificat. O variantă mai adecvată este generalizarea la structuri n-dimensionale prin folosirea unor constructori pe fiecare dimensiune.

Structurile rezultate se vor numi *PowerArray, ParArray* şi respectiv *PArray.* În acestă secţiune se prezintă algebra şi notaţia *PowerArray* cu exemplificări în cazul bidimensional. Pentru celelalte tipuri de structuri n-dimensionale extinderea se face în mod analog, ţinând cont de teoria asociată structurilor liniare corespunzătoare.

8.5.1 Definiţii

O structură *PowerArray* este o structură de date nevidă care conţine o putere a lui 2 elemente de acelaşi tip.

Constructorul de tip asociat structurii de date *PowerArray* este:

$$PowerArray : Type \times (Nat)^n \rightarrow Type$$

care are ca argumente un tip (X) şi n numere naturale $(n_i : 0 \leq i < n)$ şi returnează tipul tuturor structurilor cu elemente de tipul X şi cu 2^{n_i} elemente pe dimensiunea i. De exemplu, $PowerArray.Nat.2.3$ este tipul tuturor matricelor cu 2^2 coloane şi 2^3 linii care conţin elemente numere naturale.

Două structuri *PowerArray* care au aceleaşi dimensiuni şi elemente de acelaşi tip se numesc *similare.* O structură *PowerArray* cu un element pe fiecare dimensiune se numeşte *singleton* şi este reprezentată astfel: $[a]$, unde a este unicul element al structurii.

Două structuri *PowerArray* similare p, q, pot fi combinate pe o dimensiune sau pe alta, în două moduri diferite:

- $p \mid_i q$ *(tie)* este structura obţinută prin concatenarea structurii p cu structura q pe dimensiunea i.
- $p \#_i q$ *(zip)* este structura obţinută prin interclasarea structurii p cu structura q pe dimensiunea i.

De exemplu, dacă $p, q \in PowerArray.X.m.n$ şi $u = p|_0 q, v = p\#_0 q$ atunci $u, v \in PowerArray.X.(m+1).n$. Structura u este matricea care se obţine din coloanele lui p concatenate cu coloanele lui q, iar v este matricea ale cărei coloane sunt luate alternativ din p şi q.

Deoarece matricele sunt cele mai folosite structuri de date multidimensionale, vom defini doar algebra *PowerArray* pentru acest tip de structuri. Pentru celelalte structuri multidimensionale, algebrele se definesc similar.

Algebra structurilor *PowerArray* bidimensionale se defineşte în mod analog cu cele definite în cazurile liniare. Alegem, prin convenţie, prima dimensiune –0– coloanele şi a doua –1– liniile.

Constructorii pentru structurile *PowerArray* bidimensionale sunt:

$$
\begin{array}{l}
[.] : X \to PowerArray.X.0.0 \\
.|_0. : PowerArray.X.m.n \times PowerArray.X.m.n \to PowerArray.X.\,(m+1)\,.n \\
.|_1. : PowerArray.X.m.n \times PowerArray.X.m.n \to PowerArray.X.m.\,(n+1) \\
.\#_0. : PowerArray.X.m.n \times PowerArray.X.m.n \to PowerArray.X.\,(m+1)\,.n \\
.\#_1. : PowerArray.X.m.n \times PowerArray.X.m.n \to PowerArray.X.m.\,(n+1)
\end{array}
$$

Funcţiile de lungime pe fiecare dimensiune:
$length_i : PowerArray.X.n_0.n_1 \to Nat$ şi
$loglen_i : PowerArray.X.n_0.n_1 \to Nat$ pentru $i = 0, 1$
sunt definite prin:

$$
\begin{array}{l}
(\forall p : p \in PowerArray.X.n_0.n_1 : length_i.p = 2^{n_i}) \\
(\forall p : p \in PowerArray.X.n_0.n_1 : loglen_i.p = n_i)
\end{array}
\text{ pentru } i = 0, 1.
$$

Algebra corespunzătoare structurii *PowerArray* este formată din axiomele existente în algebra *PowerList* replicate pentru fiecare dimensiune.

1. $[a] = [b] \equiv a = b$
2. $loglen_0 > 0 \Rightarrow (\exists\, !u, v :: p = u|_0 v)$
3. $loglen_0 > 0 \Rightarrow (\exists\, !u, v :: p = u\#_0 v)$
4. $p|_0 q = u|_0 v \equiv p = u \wedge q = v$
5. $p\#_0 q = u\#_0 v \equiv p = u \wedge q = v$
6. $[a]\,|_0\,[b] = [a]\,\#_0\,[b]$
7. $loglen_1 > 0 \Rightarrow (\exists\, !u, v :: p = u|_1 v)$
8. $loglen_1 > 0 \Rightarrow (\exists\, !u, v :: p = u\#_1 v)$
9. $p|_1 q = u|_1 v \equiv p = u \wedge q = v$
10. $p\#_1 q = u\#_1 v \equiv p = u \wedge q = v$
11. $[a]\,|_1\,[b] = [a]\,\#_1\,[b]$

Comutativitatea este valabilă între oricare doi operatori, indiferent de dimensiune:

$$(p \pi q)\delta(u \pi v) = (p \delta u)\pi(q \delta v)$$

oricare ar fi π, δ operatori *tie* sau *zip*.

8.5.2 Un principiu al inducţiei pentru *PowerArray*

Fie $\Pi : PowerArray.X.m.n \to Bool$ un predicat al cărui adevăr trebuie stabilit pentru toate structurile *PowerArray* cu elemente de tipul X. Putem să demonstrăm că predicatul

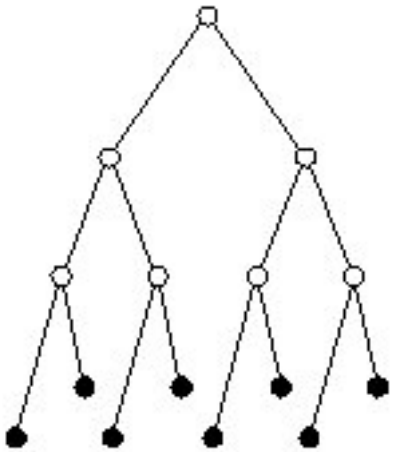

Figura 8.2: Structura arborescentă pentru matricea M

Π este adevărat, folosind următorul principiu al inducţiei [131]:

$$
\begin{array}{l}
\quad (\forall x : x \in X : \Pi.\,[x]) \\
\wedge(\quad (\forall p, q, m, n : p, q \in PowerArray.X.m.n \wedge n, m \in Nat : \Pi.p \wedge \Pi.q \Rightarrow \Pi.\,(p|_0 q)) \\
\quad \vee (\forall p, q, m, n : p, q \in PowerArray.X.m.n \wedge n, m \in Nat : \Pi.p \wedge \Pi.q \Rightarrow \Pi.\,(p\#_0 q)) \quad) \\
\wedge(\quad (\forall p, q, m, n : p, q \in PowerArray.X.m.n \wedge n, m \in Nat : \Pi.p \wedge \Pi.q \Rightarrow \Pi.\,(p|_1 q)) \\
\quad \vee (\forall p, q, m, n : p, q \in PowerArray.X.m.n \wedge n, m \in Nat : \Pi.p \wedge \Pi.q \Rightarrow \Pi.\,(p\#_1 q)) \quad) \\
\Rightarrow \\
\quad (\forall p, m, n : p \in PowerArray.X.m.n \wedge n, m \in Nat : \Pi.p)
\end{array}
$$

Pentru cazul n-dimensional principiul inducţiei va conţine un pas inductiv pentru fiecare dimensiune.

Este necesar să se folosească operatorii de construcţie într-o ordine bine determinată: întâi operatorii pentru prima dimensiune, apoi cei pentru cea de a doua şi aşa mai departe. Aşadar, dacă asociem o structură arborescentă unei structuri *PowerArray*, aceasta va fi unică. Structura arborescentă (care poate fi considerată un arbore binar de construcţie în spaţiu) reprezintă de fapt o descompunere unică a structurii. Nodurile arborelui reprezintă elementele structurii *PowerArray*. Folosim mai întâi operatorii primei dimensiuni şi rezultă un arbore ale cărui frunze sunt toate rădăcini ale altor arbori corespunzători dimensiunilor rămase.
De exemplu, dacă avem o matrice cu 4 coloane şi 2 linii, care are tipul $PowerArray.X.2.1$, structura arborescentă asociată structurii este prezentată în Figura 8.2.

8.5.3 Operatori, relaţii şi funcţii

Fie $\otimes : X \times X \to X$ un operator binar definit pe tipul scalar X. Extindem operatorul prin definirea lui pe structuri *PowerArray*. În cazul bidimensional, operatorul extins este:

$$\otimes : PowerArray.X.m.n \times PowerArray.X.m.n \to PowerArray.X.m.n$$

definit prin relaţiile:

$$\begin{array}{l}[a] \otimes [b] = [a \otimes b] \\ (p|_0 q) \otimes (u|_0 v) = (p \otimes u)\,|_0\,(q \otimes v) \quad \text{sau} \\ (p\#_0 q) \otimes (u\#_0 v) = (p \otimes u)\,\#_0\,(q \otimes v) \\ (p|_1 q) \otimes (u|_1 v) = (p \otimes u)\,|_1\,(q \otimes v) \quad \text{sau} \\ (p\#_1 q) \otimes (u\#_1 v) = (p \otimes u)\,\#_1\,(q \otimes v)\end{array}$$

Relaţiile peste tipurile scalare sunt extinse în acelaşi mod ca şi operatorii binari.

Fie o relaţie peste tipul X, definită prin funcţia booleană $\Delta : X \times X \to Bool$ şi fie $p, q, u, v \in PowerArray.X.m.n$ şi $x, y \in X$, definim relaţia extinsă prin:

$$\Delta : PowerArray.X.n.m \times PowerArray.X.n.m \to Bool$$

$$\begin{array}{l}[x]\,\Delta\,[y] \equiv [x \Delta y] \\ (p\#_0 q)\,\Delta\,(u\#_0 v) \equiv (p \Delta u) \wedge (q \Delta v) \\ (p\#_1 q)\,\Delta\,(u\#_1 v) \equiv (p \Delta u) \wedge (q \Delta v)\end{array}$$

Deci două *PowerArray* sunt în relaţia Δ dacă şi numai dacă elementele de pe aceeaşi poziţie sunt în relaţie. Se poate folosi şi operatorul (*tie*) pentru definirea relaţiei:

$$\begin{array}{l}(p|_0 q)\,\Delta\,[u|_0 v] \equiv (p \Delta u) \wedge (q \Delta v) \\ (p|_1 q)\,\Delta\,[u|_1 v] \equiv (p \Delta u) \wedge (q \Delta v)\end{array}$$

Funcţiile definite pe structurile *PowerArray* se definesc pe baza principiului inducţiei enunţat anterior, în mod analog cu cele definite pe structurile *PowerList.*

1. Funcţia de reducere $reduce : (X \times X \to X) \times PowerArray.X.m.n \to X$

   definită prin:

   $$\begin{array}{l}reduce.\otimes.[a] = a \\ reduce.\otimes.(p|_0 q) = reduce.\otimes.p \otimes reduce.\otimes.q \\ reduce.\otimes.(p|_1 q) = reduce.\otimes.p \otimes reduce.\otimes.q\end{array}$$

   unde $\otimes : X \times X \to X$ este un operator asociativ.

   Funcţia $sum : PowerArray.X.m.n \to X$ se obţine prin instanţierea funcţiei *reduce* cu operatorul aditiv.

2. Funcţia

   $$map : (X \to Z) \times PowerArray.X.m.n \to PowerArray.Z.m.n$$

   care primeşte ca argumente o funcţie şi o structură *PowerArray* şi aplică funcţia fiecărui element din structură. Este definită prin:

   $$\begin{array}{l}map.f.[a] = [f.a] \\ map.f.(p\#_0 q) = map.f.p\,\#_0\,map.f.q \\ map.f.(p\#_1 q) = map.f.p\,\#_1\,map.f.q\end{array}$$

3. Funcţia $rev : PowerArray.X.m.n \to PowerArray.X.m.n$ este o funcţie de permutare care returnează o matrice formată din elementele matricei argument în ordine inversă pe fiecare dimensiune:

$$\begin{aligned} &rev.\,[a] = [a] \\ &rev.\,(p|_0 q) = rev.q|_0 rev.p \\ &rev.\,(p|_1 q) = rev.q|_1 rev.p \end{aligned}$$

4. Funcţii de permutare
   Elementele unei matrici pot fi rotite pe linii, la stânga sau dreapta, iar coloanele rotite în sus sau în jos. Prezentăm aici două funcţii rr (rotire la dreapta) şi rd (rotire în jos):

$$\begin{array}{l} rr : PowerArray.X.m.n \to PowerArray.X.m.n \\ rd : PowerArray.X.m.n \to PowerArray.X.m.n \end{array}$$

$$\begin{array}{lcl|lcl} rr.[a] & = & [a] & rd.[a] & = & [a] \\ rr.(p \natural_0 q) & = & rr.q \natural_0 p & rd.(p \mid_0 q) & = & rd.p \mid_0 rd.q. \\ rr.(p \mid_1 q) & = & rr.p \mid_1 rr.q & rd.(p \natural_1 q) & = & rd.q \natural_1 p \end{array}$$

Pentru a ilustra modul de demonstrare a proprietăţilor folosind principiul inducţiei structurale, demonstrăm următoarea propoziţie:

Propoziţia 8.4

$$rev.\,(map.f.p) = map.f.\,(rev.p)$$

Demonstraţie:

Cazul de bază:

$$\begin{array}{lll} & rev.\,(map.f.\,[a]) & \\ = & \quad \{\text{definiţia } map\} & \\ & rev.\,([f.a]) & \\ = & \quad \{\text{definiţia } rev\} & \\ & [f.a] & \\ = & \quad \{\text{definiţia } map\} & \\ & map.f.\,[a] & \\ = & \quad \{\text{definiţia } rev\} & \\ & map.f.\,(rev.\,[a]) & \end{array}$$

Pasul inductiv pentru $\#_0$:

$$\begin{array}{lll} & rev.\,(map.f.\,(p\#_0 q)) & \\ = & \quad \{\text{definiţia } map\} & \\ & rev.\,(map.f.p\#_0 map.f.q) & \\ = & \quad \{\text{definiţia } rev\} & \\ & rev.\,(map.f.q)\,\#_0 rev.\,(map.f.p) & \\ = & \quad \{\text{ipoteza inducţiei }\} & \\ & map.f.\,(rev.q)\,\#_0 map.f.\,(rev.p) & \\ = & \quad \{\text{definiţia } map\} & \\ & map.f.\,(rev.q\#_0 rev.p) & \\ = & \quad \{\text{definiţia } rev\} & \\ & map.f.\,(rev.\,(p\#_0 q))\,. & \end{array}$$

Pasul inductiv $\#_1$ este analog cu cel pentru $\#_0$.

Complexitatea

Complexitatea computaţională a unei funcţii:

$f : PowerArray.X.(n_0) \ldots (n_i) \ldots (n_{k-1}) \rightarrow Type$
definită de
$f.(p\ \pi_i\ q) = \phi_i.(f.p).(f.q), i = 0, 1, \ldots, (k-1)$
unde π_i este un operator *tie* sau *zip*, iar ϕ_i este o funcţie de combinare,

poate fi evaluată cu ajutorul următoarei formule

$$T_f = \sum_{i=0}^{k-1} n_i * (T.\phi_i)$$

unde $T.\phi_i$ reprezintă complexitatea de calcul a funcţiei ϕ_i.

8.5.4 Aplicaţii

Transpusa unei matrice

Algoritmul de transpunere a unei matrice are o descriere foarte simplă folosind structurile *PowerArray*:

$Trans : PowerArray.X.n.m \rightarrow PowerArray.X.n.m$
$Trans.\,[a] = [a]$
$Trans.\,(p|_0 q) = Trans.p\ |_1\ Trans.q$
$Trans.\,(p|_1 q) = Trans.p\ |_0\ Trans.q$

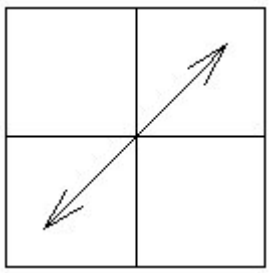

Figura 8.3: Transpunerea unei matrice

Atunci când o matrice este transpusă, liniile devin coloane şi reciproc (Figura 8.3); acest lucru este evidenţiat foarte simplu de funcţia definită anterior. Complexitatea este $m + n$.

Înmulţirea a două matrice

Algoritmul prezentat se bazează pe segmentarea matricelor şi multiplicarea elementelor lor pe o a treia dimensiune pentru realizarea în paralel a operaţiilor de înmulţire [131].

$Mult : PowerArray.X.n.m \times PowerArray.X.p.n \rightarrow PowerArray.X.p.m.$
$Mult.a.b = SumT.((R1.(TR.a).b) * (R2.b.a)$

Se foloseşte operatorul multiplicativ extins $*$.

$TR : PowerArray.X.n.m \rightarrow PowerArray.X.0.n.m$
$TR.[a] = [a]$
$TR.(p|_0 q) = TR.p|_1 TR.q$
$TR.(p|_1 q) = TR.p|_2 TR.q$

$R1 : PowerArray.0.n.m \times PowerArray.X.n.p \rightarrow PowerArray.X.p.n.m$
$R2 : PowerArray.p.n \times PowerArray.X.n.m \rightarrow PowerArray.X.p.n.m$
$R1.a.[b] = a$
$R1.a.(p|_0 q) = R1.a.p$
$R1.a.(p|_1 q) = R1.a.p|_0 R1.a.q$

$R2.b.[a] = b$
$R2.b.(p|_0 q) = R1.b.p$
$R2.b.(p|_1 q) = R1.b.p|_2 R1.b.q$

$SumT : PowerArray.X.p.n.m \rightarrow PowerArray.X.m.p$
$SumT.[a] = [a]$
$SumT.(p|_0 q) = SumT.p|_0 SumT.q$
$SumT.(p|_1 q) = SumT.p + SumT.q$
$SumT.(p|_2 q) = SumT.p|_1 SumT.q$

Funcţia TR realizează o transpoziţie şi schimbă dimensiunea 0 în dimensiunea 2.

Folosim două funcţii de multiplicare $R1$ şi $R2$; prima pentru prima matrice şi a doua pentru a doua matrice. Cel de-al doilea parametru al lor aduce doar informaţii despre numărul necesar de multiplicări.

Funcţia $SumT$ execută o reducere cu operatorul aditiv pe dimensiunea 1 şi schimbă dimesiunea 2 în dimensiunea 1.

Complexitatea depinde de complexitatea fiecărei funcţii în parte:

$$T_{TR}.n.m = n + m,$$
$$T_{R2}.p.n.m = n + m,$$
$$T_{R1}.p.n.m = n + p,$$
$$T_{SumT}.p.n.m = p + n + m.$$

Funcţiile TR şi $R2$ pot fi calculate în paralel. Rezultă, deci, $T_{Mult}.m.n.p = 3n + 2m + 2p$.

Pentru matrice pătratice, algoritmul poate fi simplificat prin condensarea într-o singură etapă (funcţie) a multiplicării matricei a şi a transpunerii ei; deasemenea, nu sunt necesare funcţii care au ca parametrii două structuri $PowerArray$.

8.5.5 Evaluarea relaţiilor de recurenţă

În numeroase cazuri, soluţia unei probleme se reduce la evaluarea ultimului termen, sau a tuturor termenilor unei relaţii de recurenţă.

O recurenţă liniară cu doi termeni de forma:

$x_i = a_i x_{i-1} + b_i x_{i-2}, (i = 2, \ldots, n)$, unde se cunosc x_0 şi x_1, a_i şi b_i se poate scrie folosind ecuaţia matriceală:

$y_i = M_i y_{i-1}$, unde $y_i = \begin{bmatrix} x_i \\ x_{i-1} \end{bmatrix}, \quad (i = 1, ..., n) \quad M_i = \begin{bmatrix} a_i & b_i \\ 1 & 0 \end{bmatrix}, \quad (i = 2, ..., n),$

prin urmare:

$$\begin{bmatrix} x_n \\ x_{n-1} \end{bmatrix} = y_n = M_n M_{n-1} ... M_2 \begin{bmatrix} x_1 \\ x_0 \end{bmatrix}, n \geq 2.$$

O recurenţă liniară cu k termeni

$$x_i = a_i^0 x_{i-1} + a_i^1 x_{i-2} + ... + a_i^{k-1} x_{i-k}, \quad (i = k, ..., n)$$

poate fi scrisă matriceal astfel:

$$\begin{bmatrix} x_i \\ x_{i-1} \\ \vdots \\ x_{i-k+1} \end{bmatrix} = \begin{bmatrix} a_i^0 & a_i^1 & \ldots & a_i^{k-2} & a_i^{k-1} \\ 1 & 0 & \ldots & 0 & 0 \\ \vdots & \ddots & & & \\ 0 & 0 & \ldots & 1 & 0 \end{bmatrix} \begin{bmatrix} x_{i-1} \\ x_{i-2} \\ \vdots \\ x_{i-k} \end{bmatrix},$$

iar o recurenţă de forma

$$x_i = a_i^0 x_{i-1} + a_{i-2}^1 x_i + ... + a_i^{k-2} x_{i-k+1} + c_i, \quad (i = k-1, ..., n)$$

poate fi scrisă astfel:

$$\begin{bmatrix} x_i \\ x_{i-1} \\ \vdots \\ 1 \end{bmatrix} = \begin{bmatrix} a_i^0 & a_i^1 & \ldots & a_i^{k-2} & c_i \\ 1 & 0 & \ldots & 0 & 0 \\ \vdots & \ddots & & & \\ 0 & 0 & \ldots & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{i-1} \\ x_{i-2} \\ \vdots \\ 1 \end{bmatrix}.$$

Recurenţele neliniare se pot reduce la recurenţe liniare printr-o schimbare de variabilă corespunzătoare.

De exemplu, fie recurenţa neliniară de ordinul întâi

$$x_i = a_i + b_i/x_{i-1},$$

cu x_0 dat.

Dacă înmulţim ambii membrii ai ecuaţiei precedente cu $x_0x_1 \ldots x_{i-1}$ şi notăm $y_i = x_i * \ldots * x_0$, vom obţine:

$$y_i = a_iy_{i-1} + b_iy_{i-2},$$

cu $y_0 = x_o$ dat, care este o recurenţă liniară de ordinul doi şi poate fi rezolvată conform metodei anterioare. Apoi x_i se poate calcula prin $x_i = y_i/y_{i-1}$.

Cazul mai general de recurenţă neliniară

$$x_0 \text{ dat, iar } x_i = \frac{a_ix_{i-1} + b_i}{c_ix_{i-1} + d_i},$$

se reduce la forma unei recurenţe liniare de ordinul întâi prin schimbarea de variabilă $X_i = c_{i+1}x_i + d_{i+1}$.

O altă posibilitate de rezolvare a recurenţelor neliniare este prin exprimarea lor directă în formă matriceală. Dacă înlocuim expresia

$$x_1 = \frac{a_1x_0 + b_1}{c_1x_0 + d_1}$$

în formula corespunzătoare a lui x_2, vom obţine o exprimare a lui x_2 în funcţie de x_0:

$$x_2 = \frac{(a_2a_1 + b_2c_1)\,x_0 + a_2b_1 + b_2d_1}{(c_2a_1 + d_2c_1)\,x_0 + c_2b_1 + d_2d_1},$$

şi procesul poate continua.

Dacă expresia finală este de forma

$$x_n = \frac{\alpha x_0 + \beta}{\gamma x_0 + \delta}$$

atunci

$$\begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix} = \begin{bmatrix} a_n & b_n \\ c_n & d_n \end{bmatrix} \begin{bmatrix} a_{n-1} & b_{n-1} \\ c_{n-1} & d_{n-1} \end{bmatrix} \cdots \begin{bmatrix} a_1 & b_1 \\ c_1 & d_1 \end{bmatrix}.$$

Algoritmul paralel pentru rezolvarea recurenţelor

Folosind structuri *PowerArray* şi *ParList*, putem obţine un algoritm eficient şi simplu pentru rezolvarea formulelor recurente liniare.

Presupunem că recurenţa are k termeni şi $k = 2^t$ (în caz contrar putem să completăm cu zerouri). Primul argument este o structură *ParList* ale cărei n elemente sunt structuri *PowerArray* reprezentând matricele M_i. Cel de-al doilea argument reprezintă un vector care conţine primii k termeni ai recurenţei (cei daţi).

$Recc : ParList.(PowerArray.X.t.t).n \times PowerArray.X.t.1 \rightarrow PowerArray.X.t.1$
$Recc.p.x = Mult.(reduce.(Mult).p).x.$

Algoritmul de înmulţire a matricelor poate fi extins uşor pentru structuri *ParArray* şi în acest caz, nu mai este nevoie să se impună condiţii asupra lui k.

Folosim acest algoritm pentru a calcula polinoame ortogonale, cum sunt polinoamele *Legendre* şi *Cebâşev* şi *Hermite* [37]. Pentru toate cazurile prezentate k este egal cu 2.

Polinoame Legendre

Relaţia de recurenţă care defineşte polinoamele ortogonale Legendre este:

$$\begin{cases} l_0(x) & = & 1 \\ l_1(x) & = & x \\ l_{(n+1)}(x) & = & xl_n(x) - \frac{n^2}{(2n-1)(2n+1)} l_{(n-1)}(x), n \geq 1 \end{cases}$$

Algoritmul paralel corespunzător bazat pe funcţia *Recc*, calculează valorile polinoamelor de grad $n+1$ şi n în punctul x dat:

$$\left[\ l_{n+1}(x) \quad l_n(x)\ \right] = Recc.\left[|i : i \in \overline{n} - \{0\} : \left[\begin{array}{cc} x & \frac{-i^2}{(2i-1)(2i+1)} \\ 1 & 0 \end{array}\right]\right][x\ 1]\,.$$

Polinoame Cebâşev (speţa I)

Relaţia de recurenţă care defineşte polinoamele ortogonale Cebâşev (speţa I) este:

$$\begin{array}{l} t_0\,(x) = 1 \\ t_1\,(x) = x \\ t_{n+1}\,(x) = 2x\,t_n\,(x) - t_{n-1}\,(x)\,, n \geq 1. \end{array}$$

Algoritmul paralel corespunzător bazat pe funcţia *Recc* calculează valorile polinoamelor de grad $n+1$ şi n în punctul x dat:

$$\left[\ t_{n+1}(x) \quad t_n(x)\ \right] = Recc.\left[|i : i \in \overline{(n-1)} : \left[\begin{array}{cc} 2x & -1 \\ 1 & 0 \end{array}\right]\right][x\ 1]\,.$$

Polinoame Cebâşev (speţa II)

Relaţia de recurenţă care defineşte polinoamele ortogonale Cebâşev(speţa I) este:

$$\begin{aligned} &q_0(x) = 1 \\ &q_1(x) = 2x \\ &q_{n+1}(x) = 2x\, q_n(x) - q_{n-1}(x), n \geq 1. \end{aligned}$$

Algoritmul paralel corespunzător bazat pe funcţia *Recc* calculează valorile polinoamelor de grad $n+1$ şi n în punctul x dat:

$$\left[\begin{array}{cc} q_{n+1}(x) & q_n(x) \end{array}\right] = Recc.\left[|i : i \in \overline{(n-1)} : \left[\begin{array}{cc} 2x & -1 \\ 1 & 0 \end{array}\right]\right][2x\ 1].$$

Polinoame Hermite

Relaţia de recurenţă care defineşte polinoamele ortogonale Hermite este:

$$\begin{aligned} &h_0(x) = 1 \\ &h_1(x) = 2x \\ &h_{n+1}(x) = 2x\, h_n(x) - 2n\, h_{n-1}(x), n \geq 1. \end{aligned}$$

Algoritmul paralel corespunzător, calculează valorile polinoamelor de grad $n+1$ şi n în punctul x dat:

$$\left[\begin{array}{cc} h_{n+1}(x) & h_n(x) \end{array}\right] = Recc.\left[|i : i \in \overline{n} - \{0\} : \left[\begin{array}{cc} 2x & -2n \\ 1 & 0 \end{array}\right]\right][2x\ 1].$$

Sumar

Structurile de date definite în acest capitol permit specificarea unor algoritmi paraleli cu ajutorul definirii unor funcţii pe aceste structuri. Deşi sunt dedicate numai anumitor tipuri de algoritmi (Divide&Impera), ele sunt foarte folositoare, permiţând specificarea şi construcţia algoritmilor într-un mod simplu şi corect.

PowerList este o structură de date care permite descrierea unei diversităţi de algoritmi. Teoria *PowerList* este foarte simplă. Proprietăţile structurilor *PowerList* pot fi demonstrate folosind principiul inducţiei structurale, pe baza căruia se definesc şi funcţiile peste structuri *PowerList*. Paralelismul este introdus implicit prin constructorii | şi #. Legătura dintre *PowerList* şi hipercuburi conduce la implementări eficiente pe arhitecturi cu o astfel de topologie.

Structurile de date *ParList* reprezintă o generalizare a structurilor de date *PowerList* şi permit specificarea unor algoritmi diverşi. Funcţiile definite peste structurile *PowerList* pot fi uşor extinse la cazul *ParList* prin adăugarea unei definiţii noi corespunzătoare pasului inducţiei bazat pe operatorii $\triangleleft$ şi $\triangleright$. De asemenea, folosind doar operatorii $\triangleleft$ şi $\triangleright$ se pot obţine algoritmi paraleli pentru anumite clase de probleme.

Structurile *PList* sunt cele mai generale ele permiţând descrierea unor algoritmi Divide&Impera folosind divizare multiplă şi combinată. Gradul de paralelizare poate fi astfel mult mărit. Datorită faptului că divizarea datelor se poate face în mai mult decât două părţi, eficienţa creşte. Totuşi o segmentare prea mare a datelor nu este până la urmă eficientă, datorită faptului că determină creşterea complexităţii pasului de combinare. În acest caz, o posibilitate de sporire a eficienţei algoritmilor specificaţi prin *PList* ar fi tratarea fiecărui pas de combinare ca un algoritm specificat prin *ParList.*

Pentru calculul transformatei Fourier, a treia formă de specificare -$PList$ este cea mai generală, ea incluzându-le şi pe primele două. Dacă n este o putere a lui 2 algoritmul este cel specificat cu notaţia *PowerList.* Dacă n este prim, lista *PosList* din definiţia funcţiei *fft* pe *PList* va conţine un singur element egal cu n, şi este echivalentă cu algoritmul specificat prin $ParList$.

Structurile *PowerArray* permit specificarea concisă a diferiţi algoritmi. Cu ajutorul lor se pot defini algoritmi paraleli generali pentru diferite clase de probleme în care intervin date cu structură matriceală. Extinderea pe mai multe dimensiuni conduce la obţinea unor algoritmi paraleli clari şi eficienţi.

Corectitudinea acestor algoritmi este uşor de demonstrat datorită faptului că definirea lor se bazează pe principiul inducţiei structurale asociat mulţimii de structuri specificate.

Deşi folosirea acestor structuri se restrânge doar la un anumit tip de algoritmi, ele pot fi incluse în definiţia unui model de dezvoltare de programe paralele mai complex, care să susţină avantajele aduse de acestea.

Partea IV

Modele

Capitolul 9

Modele de calcul paralel

Neconcordanţa dintre arhitecturile paralele şi softul paralel, poate fi rezolvată prin dezvoltarea unui model de calcul paralel (MCP). Un asemenea model, trebuie să ascundă detaliile arhitecturale, dar să rămână suficient de concret, astfel încât să permită realizarea de programe eficiente.

Un *model de calcul paralel* este o interfaţă între soft şi hard. Un model de calcul paralel poate fi considerat o *maşină abstractă*, care furnizează anumite operaţii nivelului de programare şi care necesită implementări pentru fiecare dintre aceste operaţii, pe toate tipurile de arhitecturi paralele. Rolul unui model de calcul paralel este de a separa dezvoltarea programelor paralele de detaliile ce ţin de execuţia efectivă pe diferite arhitecturi. Furnizează atât abstractizare cât şi stabilitate. Abstractizarea apare datorită faptului că operaţiile furnizate de către model sunt de nivel mai înalt decât cel al arhitecturilor, simplificând astfel structura softului şi diminuând dificultatea construirii lui. În acelaşi timp, modelul reprezintă un punct de start fix pentru implementarea pe fiecare arhitectură. Prin urmare, un model de calcul paralel separă dezvoltarea programelor de detaliile de implementare. Mai mult decât atât, permite ca deciziile de implementare să fie considerate doar o singură dată pentru fiecare arhitectură ţintă şi nu pentru fiecare program în parte.

În acest capitol vom analiza caracteristicile necesare ale unui MCP precum şi o clasificare a modelelor de calcul paralel. Această analiză se bazează pe analiza făcută de D. Skilicorn and D. Talia în [156].

9.1 Caracteristicile unui model de calcul paralel ideal

Deoarece putem considera modelele ca fiind maşini abstracte, aceste modele sunt pe diferite nivele de abstractizare. De exemplu, fiecare limbaj de programare poate fi interpretat ca şi un model în acest sens, deoarece furnizează o vedere simplificată a harware-ului. Comparaţia modelelor este prin urmare dificilă, nu numai datorită diferitelor nivele de abstractizare dar şi datorită faptului că anumite modele de nivel înalt pot fi emulate de către altele de nivel mai scăzut. Un model de nivel scăzut poate emula nu doar un model de nivel mai înalt, ci chiar mai multe.

După cum am mai precizat, un program paralel poate fi extrem de complex. Putem considera un program care se execută pe un sistem cu o sută de procesoare, care execută la orice moment o sută de procese în fiecare moment. Pentru a limita întârzierile datorită comunicaţiilor şi accesului la memorie, fiecare procesor multiplexează câteva fire de execuţie, deci numărul de fire de execuţie active este mai mare decât o sută (să-l considerăm 300). Orice fir de execuţie poate să comunice cu orice alt fir şi această comunicaţie poate fi sincronă sau asincronă. Prin urmare pot fi până la 300^2 de interacţiuni "în execuţie" la fiecare moment. Concluzia este deci că programul corespunzător trebuie să permită o descriere mult mai abstractă decât descrierea fiecărei entităţi în parte. Acest lucru implică faptul că modelele de calcul paralel trebuie să aibă un nivel mai ridicat de abstractizare decât cele corespunzătoare programării secvenţiale.

De asemenea, datorită diferenţelor de implementare pe diferite arhitecturi, modelele de calcul paralel trebuie să conţină abstractizări corespunzătoare fiecărei clase de arhitecturi.

Abstractizarea însă poate conduce la o micşorare a performanţei, iar creşterea performanţei este principala motivaţie a utilizării calculului paralel. Prin urmare, abstractizarea nu trebuie deci să depăşească un nivel de la care nu se mai poate ajunge la o execuţie suficient de eficientă pe diferite tipuri de arhitecturi.

Un model trebuie să fie deci şi abstract şi eficient. Precizăm în continuare principalele caracteristici pe care ar trebui să le aibă un asemenea model:

1. **Abstractizare.** Deoarece un program paralel executabil este atât de complex, modelul trebuie să ascundă programatorului detaliile de implementare. Este preferabil ca structura programului executabil să fie specificată pe cât posibil printr-un mecanism de translaţie (compilator, sistem run-time) decât de către programator. Aceasta implică ca modelul să trateze următoarele:

   - *Partiţionarea* programului în procese paralele.
   - *Maparea* proceselor pe procesoare.
   - *Comunicaţia* între procese.
   - *Sincronizarea* între procese.

   (Amănunte cu privite la partiţionare, mapare, comunicaţie şi sincronizare au fost discutate în Capitolul 2, Secţiunea 2.1.)

   Partiţionarea şi maparea sunt cunoscute ca fiind probleme cu complexitate-timp exponenţială în cazul în care se doreşte obţinerea optimelor. Comunicaţia necesită plasarea celor două capete ale comunicaţiei în procesele corespunzătoare la poziţiile corespunzătoare. Sincronizarea impune evaluarea stării globale a calculului, care este de dimensiuni mari. Prin urmare, productivitatea ar scădea foarte serios, dacă s-ar pretinde programatorului să trateze toate acestea.

   Deci, modelul trebuie să fie cât mai abstract şi mai simplu posibil. Nu este necesar să existe o cuplare foarte strânsă între modul de specificare al programului şi modul său de execuţie.

2. **Metodologie de dezvoltarea a softului.** Cerinţa anterioară implică o distanţă mare între informaţia furnizată de către programator despre structura semantică a programului şi structura detaliată a execuţiei lui. Cuplarea acestor două necesită un fundament semantic foarte strict pe care să se poată construi tehnici de transformare corecte. Tehnicile de compilare *ad hoc* nu pot face faţă unor probleme de asemenea complexitate.

   Există de asemenea o mare distanţă între specificaţii şi programe, care trebuie să fie şi ea rezolvată pe baza unui temeinic fundament semantic. Cu mici excepţii, softul secvenţial este construit pe baza unor blocuri standard. Corectitudinea acestor programe este foarte rar demonstrată; mai degrabă se folosesc diferite variante de testare. Acestă metodologie bazată pe testare şi depanare, nu se potriveşte programării paralele din cel puţin două motive. În primul rând, spaţiul stărilor care trebuie să fie testate este mult mai mare datorită partiţionării şi mapării. Depanarea ar fi extrem de dificilă pentru că, de exemplu, un simplu "checkpoint" ar fi extrem de dificil de construit. În al doilea rând, un programator ar putea face testarea doar pe o gamă redusă de arhitecturi ţintă; prin urmare testarea nu ar implica o validare completă nici pentru stările considerate de testare.

   Verificarea unui program după construcţia sa nu este o tehnică larg răspândită şi probabil nici nu va fi. Astfel, doar o metodologie de dezvoltare a programelor corect prin construcţie poate fi avută în vedere. Folosirea unor asemenea metodologii pentru programarea secvenţială este considerată de către mulţi supefluă, dar în cazul programării paralele este esenţială. Spre deosebire de programarea secvenţială unde folosirea unor asemenea metodologii este considerată a fi mult prea costisitoare în raport cu costul programului rezultat, în cazul programării paralele folosirea unei astfel de metodologii conduce nu doar la asigurarea corectitudinii dar şi la micşorarea costurilor de dezvoltare.

   Un program paralel este dificil de realizat fără structurarea, într-un fel sau altul, a procesului de construcţie. Este deci necesară structurarea construcţiei şi construcţia corectă se poate face pe baza acestei structurări. O variantă foarte bună este de a adapta abordarea derivaţională şi de calcul a construcţiei programelor secvenţiale pentru cazul paralel. Folosind această abordare, un program este construit pornind de la o specificaţie, care este rafinată în mod repetat, până se ajunge la o formă executabilă. Dacă este necesar, procesul de rafinare poate fi continuat pentru a se ajunge la o formă executabilă mai eficientă. Avantajele abordării acestei variante sunt:

   - structurează procesul de dezvoltare, simplificând şi micşorând paşii,
   - evidenţiază punctele de decizie, forţând programul să aleagă cum să fie făcut fiecare calcul, mai degrabă decât să folosească prima metodă disponibilă,
   - furnizează o înregistrare a construcţiei programului, care serveşte ca şi documentaţie,
   - păstrează corectitudinea în timpul procesului, prin folosirea doar a transformărilor

pentru care este demonstrat faptul că păstrează corectitudinea; deci nu mai este nevoie de o demonstrare finală,

- demonstraţiile pentru proprietăţile de păstrare a corectitudinii, trebuie făcute doar în timpul construcţiei sistemului de derivare.

3. **Independenţa de arhitectură.** Un model nu trebuie să înglobeze trăsături particulare ale unei arhitecturi, astfel încât un program dezvoltat pe baza modelului să nu necesite schimbări pentru a se executa pe diferite arhitecturi. Această cerinţă este esenţială, dacă se are în vedere dezvoltarea pe scară largă a softului pentru calculatoare paralele.

   Arhitecturile paralele au durata de viaţă relativ scurtă, datorită vitezei cu care se dezvoltă tehnologia procesoarelor şi de interconectare. Şi, pe de altă parte, este puţin probabil ca un calculator paralel nou să semene foarte mult cu arhitecturile pe care le înlocuieşte. Pentru a impune programarea paralelă pe viitor este necesar să se izoleze softul de schimbările ce au loc la nivel de arhitectură.

   Prin urmare, un model trebuie să nu depindă de caracteristicile nici unei arhitecturi. Această cerinţă nu ar fi dificil de realizat, datorită faptului că orice model suficient de abstract o poate satisface, dar este în opoziţie cu alte cerinţe asupra modelului, cum sunt de exemplu, cerinţele asigurării unei implementări eficiente şi de evaluare a costurilor.

4. **Simplitate.** Un model trebuie să fie simplu de înţeles şi predat pentru a se putea asigura pregătirea de dezvoltatori de programe care să îl folosească. Această cerinţă este necesară dacă se doreşte ca programarea paralelă să se impună în lumea informaticii aplicative. Dacă modelele de calcul paralel vor putea să ascundă complexităţile impuse de calculul paralel atunci mulţi programatori vor putea să folosească calculul paralel. Aceasta ar putea duce la impunerea programarii paralele pe scară largă.

5. **Implementare eficientă.** Un MCP trebuie să facă posibilă implementarea, pe întreaga gamă de arhitecturi existente, cu o eficienţă rezonabilă pe fiecare. Nu este nevoie ca această cerinţă să fie respectată la un nivel la fel de ridicat ca şi în cazul calculului de înaltă performanţă ("high-performance computing"), deoarece pentru majoritatea aplicaţiilor paralele, costul dezvoltării softului este mult mai mare decât orice costuri asociate cu timpul de execuţie. Este suficient să se ceară ca timpii de execuţie pe diferite arhitecturi paralele să fie de acelaşi ordin şi diferenţele dintre constante să nu fie prea mari. Este acceptabil chiar să existe diferenţe logaritmice în performanţă.

   Pentru majoritatea problemelor (diferite de cele ce ţin de calculul de înaltă performanţă) asigurarea celei mai performante variante pentru o arhitectură dată, nu este o cerinţă chiar atât de stringentă, mai ales dacă aceasta s-ar obţine pe baza unor costuri foarte ridicate de dezvoltare şi de întreţinere.

Anumite constrângeri ale arhitecturilor, bazate pe proprietăţile de comunicare sunt bine fundamentate acum. O arhitectură este considerată a fi puternică dacă poate să execute orice aplicaţie eficient.

Cea mai puternică clasă de arhitecturi conţine calculatoare MIMD cu memorie partajată şi calculatoare MIMD cu memorie distribuită, pentru care capacitatea reţelei de interconectare creşte mai repede decât numărul de procesoare – p, cu un factor de cel puţin $p \log p$. Pe un asemenea calculator, execuţia unei aplicaţii arbitrare cu paralelism p care necesită un timp t, poate fi executată astfel încât costul $p * t$ să se conserve. Timpul estimat pentru un calcul abstract nu se poate regăsi pentru o implementare reală deoarece comunicaţiile şi accesurile la memorie impun întârzieri ("latencies"), care sunt în general proporţionale cu diametrul reţelei. Totuşi aceste întârzieri pot fi compensate prin folosirea multiprogramării. Arhitecturile din această clasă sunt foarte puternice, dar nu sunt scalabile datorită proporţiei mari a resurselor lor care sunt implicate în reţeaua hardware de interconectare.

O altă clasă de arhitecturi conţine calculatoare MIMD cu memorie distribuită, pentru care capacitatea reţelei creşte doar liniar cu numărul de procesoare (factor p). Aceste calculatoare sunt scalabile deoarece necesită doar un număr constant de legături de comunicaţie pentru fiecare procesor (vecinii unui procesor nu sunt modificaţi prin scalare) şi deoarece doar o proporţie constantă din resurse sunt afectate reţelei hardware. Costul execuţiei unei aplicaţii este în acest caz $p * t * d$, unde d este diametrul reţelei de interconectare. În execuţia aplicaţiilor arbitrare este posibil ca fiecare procesor să fie angrenat în câte o operaţie de comunicaţie, care ar necesita d paşi. Arhitecturile din această clasă sunt deci scalabile, dar nu atât de puternice ca şi cele din prima clasă.

Calculatoarele SIMD formează o clasă de calculatoare scalabile, dar ineficiente pentru aplicaţii arbitrare. Aceasta se datorează incapacităţii de a executa mai mult decât un mic număr de operaţii diferite simultan.

În general, timpul necesar comunicaţiilor pentru un program poate fi micşorat în două moduri: fie prin reducerea numărului de operaţii de comunicaţie simultane, fie prin reducerea distanţei pe care o traversează un mesaj.

6. **Măsuri de cost.** În afara timpului de execuţie, care este critic pentru un program paralel, trebuie avute în vedere şi alte metrici care permit evaluarea performanţei unui program paralel.

   În cazul programării secvenţiale, interacţiunea dintre metricile de cost şi procesul de proiectare este una relativ simplă. Procesul de proiectare trebuie să ia în calcul doar modificarea complexităţii asimptotice. Sau altfel spus, procesul de proiectare pentru programarea secvenţială poate fi divizat în două părţi: în prima, se iau decizii cu privire la algoritmi şi costurile asimptotice ale programului ce pot fi afectate, iar în cea de-a doua, se are în vedere un sistem ţintă şi în funcţie de acesta se pot lua anumite decizii de eficientizare (aranjarea textului sursă), dar care nu schimbă decât constantele din faţa costurilor asimptotice.

În cazul programării paralele, chiar şi mici schimbări în codul sursă sau schimbarea arhitecturii ţintă pot conduce la modificări ale costurilor asimptotice ale unui program.

Prin urmare, un MCP trebuie să conţină un mecanism de determinare a costurilor pentru un program (timp de execuţie, număr de procesoare, memorie folosită, etc.) încă din timpul ciclului de dezvoltare, într-un mod care să nu depindă critic de arhitectura ţintă. Aceste măsuri de cost sunt singura cale de a decide în timpul dezvoltării, alegerea unui algoritm sau al altuia. Pentru a putea fi practice, asemenea măsuri de cost, trebuie să se bazeze pe selectarea câtorva proprietăţi, deopotrivă ale arhitecturilor şi a softului. Dacă se foloseşte o metodologie de dezvoltare a programelor bazată pe derivare, se poate impune ca măsurile de cost să formeze un calcul de cost. Un asemenea calcul permite evaluarea costului unui program, independent de evaluarea altora şi permite ca transformările să fie clasificate după proprietăţile lor de modificare a costurilor.

Această cerinţă pare a viola proprietatea de abstractizare a unui MCP. Nu se pot evalua cu acurateţe costurile unui program fără a avea ceva informaţii despre calculatorul pe care programul se va executa, dar ceea ce se cere este ca aceste informaţii să fie cât mai minimale posibil. Vom presupune că un model are măsuri de cost dacă este posibilă determinarea costului unui program din dimensiunea datelor de intrare, din textul său şi din proprietăţile minimale ale arhitecturii. Aceasta înseamnă că modelul trebuie să furnizeze *costuri predictive*.

Alte cerinţe de măsurarea costurilor pentru un MCP sunt *compoziţionalitatea* şi *convexitatea*. Adică costurile unui întreg să poată fi calculate din costurile părţilor lui şi costul total să nu poată fi redus prin creşterea unuia dintre părţile sale.

Reconcilierea între cerinţe

Toate aceste proprietăţi, care sunt de dorit pentru un MCP performant, sunt într-o oarecare măsură mutual exclusive şi de aceea nu este uşor de a se găsi un echilibru între ele.

Modelele abstracte permit o construcţie simplă a programelor, dar acestea sunt dificil de compilat în cod eficient, în timp ce modelele mai puţin abstracte fac dificilă construcţia softului dar asigură implementare eficientă.

Cerinţele teoretice pentru un MCP pot fi considerate metrici pentru clasificarea şi compararea modelelor.

Nivelul de abstractizare va fi utilizat ca bază în gruparea acestora. Abstractizarea implică şi nivelul de simplitate a modelului, deoarece un model abstract nu necesită specificarea detaliilor cu privire la structura proceselor, a comunicaţiilor şi a sincronizărilor. Abstractizarea este de asemenea în strânsă legătură cu metodologia de dezvoltare a softului, deoarece operaţiile abstracte pot fi în general direct implementate dacă sunt corecte semantic.

În cazul în care structura programului de implementare este retricţionată de structura programului abstract, cerinţele de implementare eficientă şi asigurării evaluării costurilor

se pot considera îndeplinite. Un model care permite o comportare perfect dinamică a proceselor nu va asigura o implementare eficientă sau măsuri de cost, pentru că acestea depind de interacţiunile dintre procese, care nu vor fi cunoscute exact decât la momentul execuţiei. În cazul în care nu se permite crearea dinamică de procese, este posibil totuşi să nu se poată controla comunicaţiile şi acestea să nu poată fi limitate. Deci, doar în cazul unei structuri statice a programelor şi comunicaţiei limitate, cele două cerinţe pot fi asigurate. Prin urmare, a doua bază de clasificare va fi constituită din controlul structurii şi al comunicaţiilor.

Este foarte probabil ca să nu se poate construi un model care să satisfacă toate cerinţele şi să fie satisfăcător pentru toţi utilizatorii de calcul paralel din toate domeniile. Totuşi modelele care satisfac în mare măsură cerinţele enunţate sunt potenţiali candidaţi pentru un paralelism de tip "scop general", pentru aplicaţii paralele dintr-o gamă largă de domenii.

De asemenea, analiza modelelor de calcul paralel cu privire la modul în care acestea respectă cerinţele ne conduce la posibilitatea alegerii celui mai potrivit model pentru clasa de probleme pe care o avem în vedere.

9.2 Clasificarea modelelor

În funcţie de gradul lor de abstractizare modelele pot fi clasificate în şase mari categorii:

1. *Modele care abstractizează paralelismul complet.* Prin aceste modele se descrie doar scopul unui program nu şi modul în care acest scop este realizat. Dezvoltatorii de programe care folosesc aceste modele nu vor ţine cont de modul în care se vor executa programele – în paralel sau secvenţial; ei doar specifică prin notaţiile furnizate de către model ceea ce trebuie să realizeze programele. Aceste modele sunt foarte abstracte şi relativ simple deoarece nu vor fi mai complexe decât un model pentru calcul secvenţial.

2. *Modele în care paralelismul este explicit, dar partiţionarea programelor în procese (componente) este implicită.* Datorită faptului că descompunerea în componente este implicită rezultă că şi maparea, comunicarea şi sincronizarea lor sunt tot implicite. Folosind aceste modele, programatorii ţin cont de faptul că paralelismul va fi folosit şi trebuie să evidenţieze potenţialul de calcul care poate fi paralelizat; ei vor ignora totuşi măsura în care acest potenţial va fi folosit la execuţie. Aceste modele pretind în general evidenţierea paralelismului maxim prezent în algoritm şi apoi prin adaptarea algoritmului la o arhitectură ţintă se reduce în mod corespunzător gradul de paralelism. Implicaţiile legate de mapare, comunicare şi sincronizare sunt analizate la momentul adaptării unui program descris cu ajutorul unui astfel de model pentru execuţia pe o anumită arhitectură.

3. *Modele în care paralelismul şi partiţionarea sunt explicite, dar maparea, comunicaţia şi sincronizarea sunt implicite.* În acest caz partiţionarea se face explicit, dar nu sunt analizate implicaţiile legate de mapare, comunicare şi sincronizare a

componentelor rezultate. Aceste implicaţii sunt analizate ca şi în cazul modelelor anterioare la implementarea programelor.

4. *Modele în care paralelismul, partiţionarea şi maparea sunt explicite, dar comunicaţia şi sincronizarea sunt implicite.* În acest caz, programatorii nu trebuie doar să descompună calculul în procese, dar trebuie să analizeze şi cum trebuie aceste procese să fie mapate pe diferitele procesoare ale unei arhitecturi paralele, pentru a se obţine performanţa. Aceste modele furnizează o anumită abstractizare pentru acţiunile de comunicare dintre procese. Cea mai dificilă parte a descrierii comunicaţiei este necesitatea de a eticheta cele două capete ale fiecărei acţiuni de comunicare. Aproape toate modelele din această categorie încearcă să decupleze aceste două capete ale unei comunicaţii furnizând abstractizări de nivel înalt, prin şabloane de comunicare ("skeletons") sau să folosească alte căi pentru specificarea comunicaţiei. Deoarece localizarea are în general un efect important asupra performanţei, tipul reţelei de interconectare trebuie să fie luat în considerare. Datorită acestui lucru este dificil de a construi programe portabile folosind asemenea modele.

5. *Modele în care paralelismul, partiţionarea, maparea şi comunicaţia sunt explicite, dar sincronizarea este implicită.* În acest caz aproape toate detaliile de implementare sunt analizate cu excepţia deciziilor de sincronizare. În general aceasta se face prin considerarea unei semantici asincrone: mesajele sunt trimise, dar procesul care trimite nu poate depinde de momentul în care aceasta transmisie se face, şi de asemenea trimiterea de mesaje multiple nu presupune recepţionarea lor în ordinea trimiterii.

6. *Modele în care totul este explicit.* Majoritatea dintre primele modele de calcul paralel fac parte din aceasta categorie. Ele sunt destinate unui singur tip de arhitectură, gestionată în mod explicit. Este deci extrem de dificil de a construi programe folosind aceste modele datorită faptului că atât corectitudinea cât şi performanţa se asigură doar prin analiza atentă a foarte multor detalii – ceea ce poate duce la o complexitate foarte ridicată în construcţia softului. Aceasta este clasa care include modelele bazate pe "message passing" cum sunt foarte cunoscutele PVM şi MPI.

Pentru fiecare dintre aceste categorii se poate aplica o subclasificare în funcţie de gradul de control al structurii şi comunicaţiei:

- *Modele în care structura proceselor este dinamică.* Structura proceselor fiind dinamică noi procese pot fi create în timpul execuţiei, iar altele pot fi distruse. În general, acestea nu sunt foarte eficiente datorită faptului că nu pot limita volumul de comunicaţie şi astfel vor depăşi capacitatea de comunicaţie a anumitor arhitecturi. De asemenea, nu pot defini măsuri de cost deoarece costurile programului depind de decizii care se vor lua la momentul execuţiei şi deci nu pot fi evaluate la momentul proiectării.

- *Modele care sunt statice, dar nu limitează comunicaţia.* Nici acestea nu sunt în general eficiente, tot datorită nelimitării volumului de comunicaţie, dar pot defini măsuri de cost.

- *Modele care sunt statice şi care limitează comunicaţia.* Aceste modele pot restricţiona comunicaţia şi astfel pot deveni eficiente. Se pot defini de asemenea măsuri de cost.

Tabelele 9.1 şi 9.2 prezintă cele mai folosite modele de calcul paralel definite până în prezent, clasificate în funcţie de criteriile date mai înainte.

Vom analiza pe rând aceste tipuri de modele de calcul paralel pentru a evidenţia mai bine caracteristicile şi diferenţele dintre ele.

9.2.1 Paralelism implicit

Pentru programatori modelele de calcul paralel cele mai simplu de folosit sunt acelea în care nu trebuie să se evidenţieze paralelismul în mod explicit. Ascunzându-se toate acţiunile care sunt necesare pentru a se executa un calcul paralel, programatorii pot folosi cunoştinţele dobândite în dezvoltarea de programe secvenţiale. Evident aceste modele sunt foarte abstracte şi prin urmare sarcina implementatorilor nu este deloc uşoară deoarece transformarea, compilarea şi sistemul de execuţie trebuie să fie în concordanţă cu structura programului. Aceasta înseamnă divizarea calculului în componente pentru execuţie, maparea acestor componente şi planificarea tuturor comunicaţiilor şi sincronizarea dintre acestea.

S-a crezut mult timp că translatarea automată a unui program abstract în implementare poate fi realizată eficient pornind de la un limbaj imperativ secvenţial ordinar. Deşi s-a investit foarte multă muncă în crearea de *compilatoare de paralelizare*, această abordare s-a dovedit nerealistă datorită dificultăţii de a determina dacă un anumit aspect al programului este esenţial sau nu. Este un fapt cunoscut acum că procesul de translatare automată de nivel înalt este practic doar dacă începe de la un model ales cu grijă, care este în aceeaşi măsură abstract dar şi expresiv.

Este posibil să se trateze toate detaliile necesare pentru obţinerea unei implementări eficiente, independente de arhitectură, dar este în schimb extrem de dificil. În prezent doar foarte puţine asemenea modele pot garanta implementări eficiente.

Modele cu structură dinamică

Programarea funcţională a introdus un nivel înalt de abstractizare în programare. Calculul este precizat folosind o mulţime de funcţii şi ecuaţii, iar rezultatul calculului este specificat fără a se preciza cum este obţinut. Rezultatul calculului este o soluţie, de obicei un punct fix, a acestor ecuaţii. Reprezintă un cadru de lucru în care programele sunt şi abstracte şi posibil a fi transformate formal folosind substituţii ecuaţionale. Problema implementării constă în a găsi soluţia acestor ecuaţii.

Programarea funcţională de nivel înalt tratează funcţiile ca şi λ-termeni şi calculează valorile lor folosind reducerea din λ-calcul, permiţând stocarea lor în structuri de date

Clasa	Dinamice	Statice	Statice şi cu comunicaţii limitate
Paralelism implicit	Higher order functional – Haskel Concurrent Rewriting - OBJ, Maude Interleaving – UNITY Implicit Logic Languages – PPP, AND/OR, REDUCE/OR, Opera,Palm, concurrent constraint languages	Algorithmic Skeletons – PL3, Cole, Darlington	Homomorphic Skeletons – BMF Cellular Processing Languages – Cellang, Carpet, CDL, Ceprol, Crystal
Paralelism explicit, Descompunere implicită	Dataflow – Sisal, Id Explicit Logic Languages – Concurrent Prolog, PARLOG, GHC, Delta-Prolog, Strand Multilisp	Data Parallelism using Loops – variante Fortran, Modula 3* Data Parallelism on types – pSETL, parallel sets, Gamma, PEL, MOA, Nial, AT	Data-Specific Skeletons – scan, multiprefix, paralations, dataparallel C, NESL,CamlFlight
Descompunere explicită, Mapare implicită		BSP, LogP	

Tabela 9.1: Clasificarea modelelor de calcul paralel.

Clasa	Dinamice	Statice	Statice şi cu comunicaţii limitate
Mapare explicită, Comunicaţii implicite	Coordonation Languages – Linda, SDL Non-message Communication Languages – ALMS, PCN, Compositional C++ Virtual Shared Memory Annotated Functional Languages – ParAlf RPC - DP, Cedar, Concurrent CLU, DP	Graphical Languages – Enterprise, Parsec, Code Contextual Coordination Languages- Ease, ISETL-Linda, Opus	Communication Skeletons
Comunicaţii explicite, Sincronizare implicită	Process Networks – Actors, Concurrent Aggregates, ActorSpace, Darwin External OO – ABCL/1, ABCL/R, POOL-T, EPL, Emerald, Concurrent Smalltalk Objects and processes – Argus, Presto, Nexus Active Messages - Movie	Process Networks – static dataflow Internal OO - Mentat	Systolic Arrays - Alpha
Totul explicit	Message Passing – PVM, MPI Shared Memory – FORK, Java, thread packages Rendezvous- Ada, SR, Concurrent C	Occam	

Tabela 9.2: Clasificarea modelelor de calcul paralel(cont.).

transmise ca parametrii şi returnate ca şi rezultate. Un exemplu de asemenea limbaj, care foloseşte funcţionale de nivel înalt, este Haskel [163]. Acesta include câteva caracteristici clasice ale programării funcţionale cum sunt evaluarea întârziată, tipuri de date utilizator, potrivirea modelelor ("pattern matching"). Limbajul Haskel are chiar un sistem paralel de intrare-ieşire şi permite lucrul cu module.

Tehnica folosită de limbajele funcţionale de nivel înalt pentru calcularea valorilor funcţiilor se numeşte *reducere de graf* [145]. Funcţiile sunt exprimate ca şi arbori cu subarbori comuni pentru subfuncţiile partajate, prin urmare prin grafuri. Regulile de calcul selectează substructuri ale grafului, le reduc la forme mai simple pe care apoi le înlocuiesc în graful iniţial. Atunci când nu se mai poate face nici o reducere, graful rămas reprezintă rezultatul calculului.

Reducerea de graf poate fi paralelizată; se pot alege subgrafuri independente care se pot reduce în paralel. De exemplu, dacă trebuie să evaluăm o expresie $(exp1 * exp2)$, unde $exp1$ şi $exp2$ sunt expresii arbitrare, putem folosi două fire de execuţie care evaluează independent $exp1$ şi $exp2$.

Deşi ideea este simplă, este destul de dificil de aplicat în practică. În primul rând, doar calculele care contribuie la rezultatul final ar trebui calculate, altminteri s-ar face risipă de resurse şi s-ar putea ajunge, de asemenea, la alterarea semanticii programului, dacă un asemenea calcul eşuează (nu se termină). De exemplu, majoritatea limbajelor funcţionale permit construcţii condiţionale de forma:

```
if b(x) then
   f(x)
else
   g(x)
```

Evident doar una dintre valorile $f(x)$ şi $g(x)$ este necesară, dar acest lucru nu este cunoscut decât după ce valoarea $b(x)$ este cunoscută. Deci evaluarea mai întâi a valorii $b(x)$ va preveni calculul inutil, dar măreşte lungimea drumului critic. Dacă se alege evaluarea valorilor $f(x)$ şi $g(x)$ mai înainte, şi de exemplu, calculul valorii $f(x)$ eşuează doar pentru valori ale lui x pentru care care $b(x)$ este fals, atunci programul nu se va termina.

Reducerea de graf în paralel a avut un succes limitat pentru calculatoarele cu memorie partajată, iar pentru calculatoarele cu memorie distribuită eficienţa folosirii ei este încă neobţinută.

Asemenea sisteme sunt simple şi abstracte şi permit dezvoltarea de programe prin transformare, dar nu sunt eficient implementabile. Ceea ce se întâmplă la execuţie este determinat dinamic de sistemul de execuţie şi prin urmare nu se pot furniza măsuri de cost.

Rescrierea concurentă este o abordare asemănătoare în care se folosesc reguli de rescriere a unor părţi de program. Şi în acest caz programele sunt formate din termeni care descriu un anumit rezultat. Ele sunt rescrise prin aplicarea unor seturi de reguli aplicate subtermenilor, în mod repetat, până când nu se mai poate aplica nici o regulă. Termenul rezultat este chiar rezultatul programului. Setul de reguli este ales astfel încât să fie con-

fluent (adică aplicarea regulilor pe subtermeni care se suprapun duce la acelaşi rezultat în final) şi să asigure terminarea (nu există secvenţe infinite de rescrieri). În acest fel, ordinea în care se aplică aceste reguli nu modifică rezultatul final. Exemple de asemenea modele sunt OBJ [66] - un limbaj funcţional cu o semantică bazată pe logică ecuaţională şi Maude [117].

Prezentăm un exemplu bazat pe unul din [109] care va da o idee despre cum se aplică aceste rescrieri. Considerăm un modul funcţional pentru derivarea polinomială. Regulile de rescriere sunt notate pe linii care încep cu **eq**, iar cele care încep cu **ceq** sunt reguli de rescriere condiţionate.

```
fmod PolyDer is
   protected POLYNOMIAL .
   op der : Var Poly -> Poly .
   op der : Var Mon -> Poly .
   var A : Int .
   var N : NzNat .
   vars P Q : Poly .
   vars U V : Mon .
   eq der(P + Q) = der(P) +  der(Q) .
   eq der(U .  V ) = (der(U) .  V) + (U . der(V) .
   eq der(A * U) = A * der(U) .
   eq der(X ^ N) = N * (X ^ (N-1)) if N>1 .
   eq der(X ^ 1) = 1 .
   eq der(A) = 0 .
endfm
```

O expresie cum este `der(X ^ 5 + 3 * X ^ 4 - X ^ 2 + 3)` poate fi calculată în paralel deoarece se pot aplica simultan mai multe reguli de rescriere.

Şi aceste modele sunt simple şi abstracte şi permit dezvoltarea softului prin transformare, dar sunt dificil de implementat eficient şi sunt prea dinamice pentru a putea defini măsuri de cost.

Întreţeserea este o abordare care derivă din ideile multiprogramării folosite în sistemele de operare prin modele de concurenţă cum sunt *sistemele de tranziţie*. Dacă un calcul poate fi exprimat ca un set de subcalcule care comută astfel încât acestea pot fi efectuate în orice ordine şi în mod repetat, atunci există o mare libertate pentru sistemul de implementare în a decide structura reală a executării calculului. Nu este foarte uşor de a exprima un calcul în această formă, dar exprimarea poate fi uşurată prin precedarea subcalculelor de *gărzi*, care sunt valori booleene. Informal, se poate considera că semantica programului în această formă este de a evalua fiecare gardă şi apoi a executa unul sau mai multe subcalcule care au gărzile adevărate; iar această operaţie se repetă. Gărzile pot determina întreaga ordine de execuţie a subcalculelor, chiar şi cea secvenţială, dacă se folosesc gărzi de tipul `pas=i`. Totuşi intenţia e de a folosi gărzi cât mai slabe.

Un model care foloseşte această abordare este UNITY - model care a fost prezentat pe larg în Capitolul 5.

Modele bazate pe întreţesere sunt simple şi abstrate, dar nu par a fi eficient implementabile şi nu se pot defini măsuri de cost.

Limbajele logice paralele implicite exploatează faptul că procesul logic de rezoluţie conţine multe activităţi care se pot executa în paralel. Principalele tipuri de paralelism în programele logice sunt: paralelismul OR şi paralelismul AND. Paralelismul OR este exploatat prin unificarea în paralel a unui subscop prin potrivirea cu capurile mai multor clauze. De exemplu, dacă trebuie rezolvat subscopul $? - a(x)$ şi clauzele de potrivire sunt

$$a(x) : -b(x). \quad a(x) : -c(x).$$

atunci prin paralelismul OR se unifică în paralel $a(x)$ cu cele două clauze.

Paralelismul AND divide calculul scopului în mai multe fire, fiecare dintre acestea rezolvând un singur subscop în paralel. De exemplu dacă trebuie rezolvat scopul:

$$? - a(x), b(x), c(x).$$

atunci subscopurile $a(x), b(x), c(x)$ sunt rezolvate în paralel.

Limbajele logice paralele implicite determină descompunerea automată a arborelui de execuţie a unui program logic într-o reţea de fire de execuţie paralele. Aceasta se realizează de limbaj atât printr-o analiză statică la compilare cât şi la momentul execuţiei. Nu sunt necesare adnotări explicite ale programului. Exemple de asemenea limbaje sunt: PPP[57], modelul proces AND/OR [40], modelul REDUCE/OR [94], OPERA [27], PALM [28]. Aceste modele diferă prin modul în care este văzut paralelismul, dar sunt proiectate în principal pentru a fi implementate pe sisteme MIMD cu memorie distribuită. Pentru a implementa paralelismul, aceste modele folosesc fie fire de execuţie, fie strategii bazate pe subarbori. Aceste abordări corespund unor granularităţi diferite: în modelele bazate pe fire de execuţie granularitatea este fină, iar în cele bazate pe subarbori granularitatea este medie sau brută.

Ca şi celelalte modele prezentate anterior în această secţiune, aceste modele sunt simple şi abstracte dar nu pot fi implementate eficient. Totuşi există şi excepţii, unele dintre ele conducând la o performanţă foarte bună. Măsuri de cost nu pot fi definite datorită faptului că limbajele logice implicite sunt foarte dinamice.

Programarea logică prin constrângeri este o importantă generalizare a programării logice, prin care se înlocuieşte mecanismul de unificare prin potrivirea modelului ("pattern matching") cu o operaţie mai generală numită satisfacerea constrângerilor *("constraint satisfaction").* În acest cadru, o constrângere este o submulţime a spaţiului tuturor valorilor posibile pe care o variabilă o poate lua. Programatorul nu foloseşte explicit construcţii paralele în program, dar defineşte un set de constrângeri pe variabile. În programarea concurentă logică prin constrângeri, un calcul evoluează prin executarea firelor de execuţie care comunică concurent, prin plasarea constrângerilor într-un spaţiu global, iar sincronizarea se face prin verificarea dacă o constrângere este determinată de acest spaţiu. Firele de execuţie corespund unor scopuri atomice şi deci sunt activate în mod dinamic în timpul execuţiei programului. Modelele de programare concurentă logică prin constrângeri includ limbajul *cc* [150] şi limbajele CHIP CLP [86], CLP [92]. Ca şi pentru alte modele paralele logice şi acestea sunt prea dinamice pentru a se putea defini măsuri de cost.

Modele cu structură statică

Dacă impunem ca un program abstract să fie construit pe baza unor unităţi sau componente fundamentale a căror implementare este predefinită, atunci structura necesară pentru a executa acel program abstract se poate deduce fără probleme. Deci o abordare posibilă în construcţia programelor paralele este de a conecta blocuri de construcţie gata făcute, aceasta având următoarele avantaje:

- Blocurile de construcţie măresc nivelul de abstractizare pentru că reprezintă unităţi fundamentale cu care programatorul lucrează. Ele ascund o parte însemnată a complexităţii interne.
- Blocurile de construcţie pot reprezenta calcul paralel, dar pot fi compuse secvenţial, astfel încât programatorul nu trebuie să evidenţieze paralelismul explicit.
- Implementarea blocurilor de construcţie trebuie să fie făcută doar odată pentru fiecare arhitectură. Această implementare poate fi făcută de către specialişti, astfel încât să se ajungă a soluţii eficiente.

În contextul programării paralele asemenea blocuri au fost numite *"skeletons"* (*şabloane*) [35]. În Secţiunea 4.13 a Capitolului 4 am prezentat pe scurt avantajele lucrului cu aceste şabloane. De asemenea, în Capitolul 7 am analizat formalismul BMF, unde s-a evidenţiat rolul de şablon al funcţiilor *map* şi *reduce*.

În această secţiune, ne vom focaliza atenţia asupra *şabloanelor algoritmice ("algorithmic skeletons")*, acele şabloane care încapsulează structuri de control. Fiecare şablon de acest fel corespunde unui algoritm standard sau unui fragment de algoritm, iar aceste şabloane pot fi compuse secvenţial. Pentru fiecare arhitectură, compilatorul va alege modul în care fiecare algoritm încapsulat este implementat şi cum se exploatează paralelismul intra- şi inter- şabloane pentru fiecare arhitectură ţintă.

Un exemplu al acestei abordări este limbajul de programare paralelă Pisa (P^3L) [43], care foloseşte un set de şabloane algoritmice care cuprind paradigme comune ale programării paralele cum sunt: *pipeline, worker farms* şi *reductions*. De exemplu, pentru paradigma *worker farm* se foloseşte constructorul `farm`:

```
farm P in (int data) out (int result)
  W in (data) out (result)
  result = f(data)
  end
end farm
```

Când şablonul este executat, un număr de componente (workers, tasks) W se execută în paralel cu două procese P (emiţătorul şi colectorul). Fiecare componentă execută funcţia $f()$ pe partea sa de date.

Cole [35, 36] a dezvoltat şabloane similare, pentru care a furnizat de asemenea şi măsuri de cost. S-au dezvoltat şabloane şi pentru *reduce* şi *map over pairs*, *pipeline* şi *farms* [44].

Şabloanele algoritmice sunt simple şi abstracte. Sunt posibile implementări eficiente pentru şabloane şi de asemenea şi măsuri de cost. Totuşi datorită faptului că programele sunt dezvoltate printr-o compunere de şabloane, expresivitatea limbajului de programare abstract este discutabilă.

Modele cu structură statică şi comunicaţie limitată

Există şi abordări care folosesc şabloane ce limitează comunicaţia, în general datorită faptului că încorporează şi informaţie geometrică a datelor.

Şabloanele omeomorfe bazate pe tipuri de date, derivate din formalismul BMF reprezintă un asemenea model. Şabloanele acestui model se bazează pe anumite tipuri de date cum sunt listele, tablourile, arborii şi altele. Toate omeomorfismele pe un tip de date pot fi exprimate ca o instanţă a unui singur şablon de calcul recursiv, astfel încât paşii de implementare trebuie făcuţi doar o dată pentru fiecare tip de dată. Mai multe despre această abordare au fost prezentate în Capitolul 7.

Această abordare este simplă şi abstractă şi generează un mediu de transformări ecuaţionale. Şablonul de comunicaţie necesar pentru fiecare tip de dată este cunoscut ca o topologie standard pentru acel tip. Se pot construi implementări eficiente dacă această topologie standard poate fi scufundată eficient pe reţeaua de interconectare a arhitecturii ţintă. Datorită faptului că atât calculul cât şi comunicaţiile sunt determinate în avans, pot fi furnizate măsuri de cost.

Un alt model care se încadrează în această clasă îl formează *limbajele de procesare celulară* bazate pe modelul de execuţie a automatelor celulare. Un automat celular constă dintr-o latice de celule de dimensiune n, posibil infinită. Fiecare celulă este conectată la un număr limitat de celule adiacente. O celulă are o stare care este aleasă dintr-un alfabet finit. Starea globală a automatului celular este determinată complet de valorile stărilor celulelor. Starea fiecărei celule este dată de o variabilă simplă sau structurată care ia valori într-o mulţime finită. Stările tuturor celulelor sunt actualizate simultan, periodic, la intervale de timp discrete. Celulele îşi actualizează valorile folosind o funcţie de tranziţie, care preia ca intrare starea curentă a celulei locale şi a câtorva celule vecine aflate la o distanţă mărginită.

Exemple de asemenea limbaje sunt Cellang [54], CARPET [159], CDL şi CEPROL [154]. Ele permit decrierea algoritmilor celulari prin definirea stării celulelor ca variabile tipizate sau ca o înregistrare de variabile tipizate şi printr-o funcţie de tranziţie care conţine regulile de evoluţie ale automatului. Aceste limbaje folosesc modelul SIMD sau SPMD în funcţie de clasa arhitecturii ţintă. În varianta SPMD, algoritmii celulari sunt implementaţi ca o colecţie de procese cu granularitate medie, mapate pe diferite elemente de procesare. Fiecare proces execută acelaşi program (funcţia de tranziţie) pe date diferite (starea celulelor). Astfel, toate procesoarele execută în paralel aceeaşi transformare din care rezultă o transformare globală a automatului. Comunicaţiile apar doar între celulele vecine, iar şablonul de comunicaţie este cunoscut static. Aceasta duce la implementări scalabile şi eficiente atât pentru arhitecturile MIMD cât şi pentru SPMD. Pot fi furnizate şi măsuri de cost.

Crystal este un alt model care se bazează exploatarea aranjamentului geometric al

datelor [33]. Acesta este un limbaj funcţional cu tipuri de date adăugate, numite *domenii index*, necesare pentru a reprezenta geometria datelor. Caracteristica distinctivă constă în faptul că domeniile index pot fi transformate şi transformările se reflectă în partea de calcul a programelor. Este simplu şi abstract şi posedă un sistem transformaţional bazat atât pe semantica funcţională cât şi pe transformările domeniilor index. Domeniile index reprezintă o modalitate flexibilă de a incorpora topologia reţelei de interconectare a arhitecturii ţintă în derivări şi, de asemenea, Crystal posedă un set de măsuri de cost care pot ghida aceste derivări.

9.2.2 Descompunere implicită

A doua clasă conţine modele pentru care paralelismul este explicit dar descompunerea în componente de calcul (fire de execuţie, procese, ...) este implicită. În aceste modele, programatorii sunt conştienţi că paralelismul va fi folosit şi trebuie să exprime cât mai bine potenţialul de paralelizare al programelor, dar nu ştiu exact cât paralelism va fi folosit de fapt la execuţie. Ele necesită deseori ca programul să exprime paralelismul maximal al algoritmului şi apoi se reduce gradul paralelismul pentru a se adapta arhitecturii ţintă.

Modele cu structură dinamică

Dataflow [87] reprezintă un model care exprimă calculele ca şi operaţii cu intrări şi ieşiri explicite; operaţiile pot fi în principiu de orice mărime, dar sunt în general mici. Execuţia acestor operaţii depinde doar de dependenţele lor de date, adică o operaţie se execută doar după ce toate intrările ei sunt calculate. Operaţiile care nu au dependenţe mutuale de date pot fi executate concurent.

Operaţiile unui program de tip dataflow se consideră a fi conectate de căi care exprimă dependenţele de date şi pe aceste căi are loc curgerea fluxului de date. Aceste programe pot fi considerate colecţii de funcţii de ordin unu. Descompunerea este implicită deoarece compilatorul poate divide graful de execuţie în orice mod. Procesoarele execută operaţii într-o ordine care este determinată doar de datele valabile la un anumit moment. Deoarece operaţiile care au dependenţe directe între ele se execută la mari diferenţe de timp, foarte posibil pe procesoare diferite, localizarea spaţială nu aduce nici un avantaj. Prin urmare, descompunerea are un efect foarte mic asupra performanţei.

Descompunerea se poate face automat prin descompunerea programului în operaţii cât mai mici şi apoi clusterizarea (gruparea) lor pentru a obţine componente de dimensiuni corespunzătoare pentru a fi executate de procesoarele arhitecturii ţintă. Chiar şi alocarea aleatoare a operaţiilor pe procesoare conduce la performanţe bune pentru multe din sistemele dataflow.

Comunicaţia nu se face explicit în programe. Apariţia unui nume ca rezultat al unei operaţii este asociată cu toate locurile unde acel nume reprezintă o intrare a unei operaţii. Deoarece operaţiile se execută doar când toate intrările lor sunt disponibile, comunicaţia va fi întotdeauna asincronă.

Limbajele dataflow abordează diferite variante prentru a exprima operaţii repetitive. Limbaje precum Id [53] şi Sisal [116] sunt limbaje funcţionale de ordin întâi (cu atri-

buire singulară). Ele au structuri sintactice asemănătoare ciclurilor, care creează un nou context pentru fiecare execuţie a "corpului ciclului" (de aceea ele seamănă cu limbajele imperative cu excepţia faptului că fiecare nume de variabilă poate fi setat doar o dată într-un context).

În limbajul Sisal, de exemplu, paralelismul nu este explicit la nivelul codului sursă. Totuşi, sistemul de execuţie al limbajului poate exploata paralelismul; corpurile ciclurilor pot fi programate a fi executate simultan şi apoi se colectează rezultatele lor.

Limbajele dataflow sunt simple şi abstracte dar nu au o metodologie naturală de dezvoltare a programelor. Ele pot fi implementate eficient; de exemplu Sisal obţine o performanţa comparabilă cu cele mai bune compilatoare Fortran, pe arhitecturi cu memorie partajată. Dar pe sistemele cu memorie distribuită nu se obţine eficienţă. Măsuri de cost nu pot fi definite datorită faptului că planificarea operaţiilor se face mai ales în timpul execuţiei.

Limbajele logice explicite, numite şi limbaje logice concurente, specifică paralelismul explicit. Exemple de asemenea limbaje sunt PARLOG [74], Delta-Prolog [143], Concurrent Prolog[152], GHC şi Strand [60].

Aceste limbaje pot fi considerate ca o nouă interpretare a clauzelor Horn, o interpretare proces. Conform acestei interpretări, un scop atomic `<- C` poate fi văzut ca şi un proces, un scop conjunctiv `<- C1, C2, ..., Cn` ca şi un proces reţea, iar o variabilă logică partajată de două subscopuri poate fi văzută ca şi un canal de comunicaţie între cele două procese. Paralelismul se obţine prin îmbogăţirea unui limbaj precum Prolog cu un set de mecanisme de adnotare a programelor. De exemplu, unul dintre aceste mecanisme este adnotarea variabilelor logice partajate pentru a se asigura că ele sunt instanţiate doar de un singur subscop.

Un program într-un limbaj logic concurent este format dintr-o mulţime de clauze gardate:

`H <- G1, G2, ..., Gn | B1, B2, ..., Bm. n, m`$\geq$ `0`

unde `H` este capul clauzei, mulţimea `Gi` este garda şi `Bi` este corpul clauzei. Operaţional, garda este un test care trebuie evaluat cu succes cu unificarea capului pentru clauza care va fi aleasă. Simbolul | este numit operator *"commit"* şi este folosit ca o conjuncţie între gardă şi corp. Dacă garda este vidă, atunci operatorul | este omis.

Citirea declarativă a unei clauze gardate este: `H` este adevărată dacă atât conjuncţiile `Gi` cât şi `Bi` sunt adevărate. Conform interpretării proces, pentru a rezolva H este necesar a se rezolva garda `Gi` şi dacă rezoluţia sa este cu succes atunci `B1, B2, ..., Bm` se rezolvă în paralel.

În aceste limbaje este necesară specificarea explicită, folosind adnotări, ce precizează ce clauze pot fi rezolvate în paralel. De exemplu, în PARLOG, separatorii de clauze . şi ; controlează căutarea clauzelor candidate. Pentru fiecare grup de clauze separate prin . se încearcă execuţia în paralel. Clauzele care urmează după un ; sunt încercate doar dacă toate clauzele care preced ; au fost găsite ca fiind clauze care nu pot fi candidat.

Deşi limbajele logice concurente extind ariile de aplicaţii ale programării logice de la inteligenţa artificială la aplicaţii la nivel sistem, totuşi adnotarea programelor impune un

stil de programare diferit. Acestea diminuează natura declarativă a programării logice prin faptul că impune ca exploatarea paralelismului să fie responsabilitatea programatorului.

Multilisp [82] este un alt limbaj de programare simbolic în care paralelismul se realizează explicit. Acesta este o extensie a limbajului Lisp în care oportunităţile de paralelism sunt create prin *futures*. În implementarea limbajului există o corespondenţa unu-la-unu între fire de execuţie şi *futures*. O *future* aplicată unei expresii creează un fir de execuţie pentru a evalua aceea expresie în paralel. Expresia `(future X)` returnează o suspensie pentru valoarea X şi creează un fir de execuţie pentru a evalua concurent X, permiţând astfel paralelismul între două procese dintre care unul calculează valoarea şi unul o foloseşte. După ce valoarea X este calculată aceasta înlocuieşte expresia `future`. Aceste *futures* reprezintă un model de reprezentare a valorilor parţial calculate. Construcţia *future* generează un stil de programare asemănător cu cel din modelul *dataflow*.

Modele cu structură statică

Analiza algoritmilor din punct de vedere a situaţiilor, în care aceeaşi operaţie este aplicată unor date diferite şi aceste aplicaţii nu interacţionează, a dus la apariţia paralelismului de date. Aceste situaţii implică folosirea de tablouri şi pot fi văzute mai abstract ca instanţieri ale funcţionalei *map*. Astfel s-a ajuns a se considera paralelismul de date ca o abordare generală în care programele sunt compuneri de operaţii monolitice aplicate obiectelor unui tip de date şi care produc operaţii de acelaşi tip.

Se disting două abordări pentru descrierea paralelismului: prima bazată pe cicluri paralele şi cea de-a doua bazată pe operaţii monolitice pe tipuri de date.

Considerăm limbajul Fortran îmbogăţit cu instrucţinuea `ForAll`, în care iteraţiile corpului ciclului sunt conceptual independente şi pot fi executate concurent. De exemplu, instrucţiunea

```
ForAll (I=1:N, J=1:M)
   A(I,J) = I * B(J)
```

poate fi executată în paralel pe un calculator paralel. Trebuie însă ca ciclurile să nu refere aceleaşi locaţii de memorie. Această cerinţă nu se poate verifica automat, în general, şi de aceea majoritatea dialectelor de Fortran de acest fel lasă în responsabilitatea programatorului realizarea aceastei verificări. Aceste cicluri sunt instanţe *map*, deşi nu sunt aplicate întotdeauna asupra unui singur obiect de date

Multe dialecte de Fortran, cum este şi High Performance Fortran (HPF) [88] folosesc acest tip de paralelism la care se mai adaugă şi un paralelism mai direct prin includerea unor construcţii pentru specificarea modului în care structurile de date se alocă pe procesoare, cât şi a unor operaţii pentru alte calcule paralele pe date, cum sunt reducerile.

Au fost dezvoltate şi limbaje cu paralelism de date bazate pe alte tipuri de date, diferite de tablouri. Asemenea exemple sunt: parallel SETL, parallel sets, Gamma şi PEI. Limbajul paralel SETL este un limbaj asemănător limbajelor imperative cu operaţii paralele pe colecţii. De exemplu, produsul interior pentru înmulţirea matricelor se face

astfel:

```
c(i,j):=+/{a(i,k)*b(k,j): k over {1..n}}
```

Gamma este un limbaj cu operaţii paralele pe mulţimi finite.

Limbajele cu paralelism de date simplifică programarea prin faptul că operaţii care necesită ciclare în limbajele paralele de nivel jos pot fi scrise ca operaţii singulare. Maparea naturală pe arhitecturi, a operaţiilor paralele pe date, cel puţin pentru tipurile simple, determină implementări eficiente şi posibilitatea definirii unor măsuri de cost.

Modele cu structură statică şi cu comunicaţie limitată

Limbajele cu paralelism de date descrise anterior au fost dezvoltate avându-se în prim plan construcţia programelor. Există şi limbaje care au fost dezvoltate plecând de la caracteristici arhitecturale. Din această cauză, ele pun mai mult accent pe comunicaţiile care au loc la calcularea fiecărei operaţii.

Arhitectura Connection Machine 2 a inspirat o varietate de limbaje, operaţiile de bază fiind operaţii paralele pe liste. Aceste operaţii includ în general operaţia *map*, operaţii *reduce* şi posibil şi operaţii *scan* şi de permutare. Exemple de asemenea limbaje sunt: scan, multiprefix, paralations, limbajul paralel C^*, modelul scan-vector, NESL şi CamlFlight. Acestea sunt în general simple şi destul de abstracte. De exemplu, C^* este o extensie a limbajului C care încorporează trăsături ale modelului paralel SIMD. Paralelismul este implementat prin definirea unor tipuri de date paralele. Programele C^* mapează variabilele unui tip de date particular definit ca fiind paralel prin cuvântul cheie `poly`, pentru a separa elementele care se procesează. În acest fel, fiecare element de procesare execută în paralel aceeaşi instrucţiune pentru fiecare instanţă a tipului de date specificat.

Limbajele cu paralelism de date furnizează în general implementări eficiente pe cel puţin câteva arhitecturi. Se pot defini pentru ele şi măsuri de cost destul de precise. Punctul lor slab constă în faptul că operaţiile paralele sunt alese în funcţie de ceea ce poate fi implementat eficient şi prin urmare, nu există o bază pentru o metodologie formală de dezvoltare a softului.

9.2.3 Descompunere explicită

Modelele de acest tip impun ca programele abstracte să specifice componentele în care sunt divizate, dar fără a preciza plasarea acestor componente pe procesoare şi modul în care se face comunicarea.

Modele cu structură statică

Singurele exemple din această clasă sunt cele pentru care se renunţă la obţinerea *localităţii* datelor asigurându-se faptul că maparea nu va influenţa performanţa.

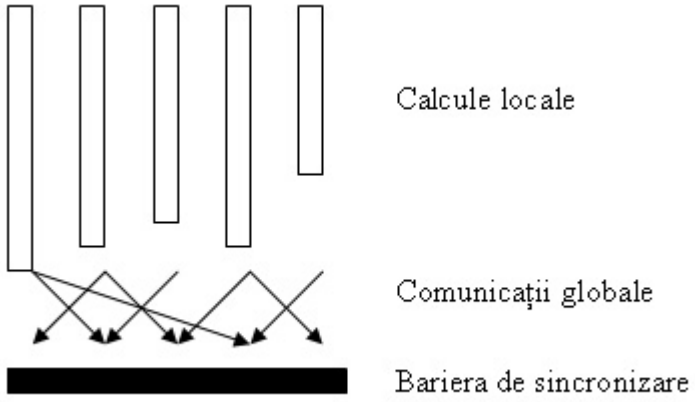

Figura 9.1: Un superpas BSP.

Bulk Synchronous Parallelism (BSP) [166] este un model în care caracteristicile reţelei de interconectare sunt evidenţiate prin câţiva parametrii arhitecturali. O maşină BSP abstractă constă într-o colecţie de p procesoare abstracte, fiecare cu memorie locală, conectate printr-o reţea de interconectare pentru care singurele proprietăţi care interesează sunt timpul necesar executării unei bariere de sincronizare (l) şi timpul în care o dată aflată la o adresă aleatoare poate fi transmisă (g). Aceşti parametrii pot fi determinaţi experimental pentru fiecare sistem paralel.

Un program abstract BSP constă din p componente şi este împărţit în *superpaşi*. Fiecare superpas constă din: un calcul pe fiecare procesor, care foloseşte doar valori locale; o comunicaţie globală de la fiecare procesor la orice submulţime de procesoare şi o barieră de sincronizare. La sfârşitul superpasului, rezultatele comunicaţiilor globale devin vizibile în memoria fiecărui procesor. Structura unui superpas este prezentată în Figura 9.1. Dacă maximul calculului global pentru un superpas necesită w unităţi de timp şi numărul maxim de valori trimise sau recepţionate de orice procesor este h, atunci timpul total pentru acel superpas este

$$t = w + hg + l$$

(unde g şi l sunt parametrii reţelei definiţi anterior). Astfel este uşor de determinat costul global al unui program.

Programele BSP trebuie să fie descompuse în componente de calcul, dar plasarea lor pe procesoare se va face automat. Comunicaţia va fi determinată de maparea componentelor, iar sincronizarea are loc pentru toate componentele.

Modelul este simplu şi destul de abstract, iar măsurile de cost ale unui program pe orice arhitectură sunt reale. Implementările sunt eficiente pe cât de eficient poate fi un program BSP (este posibil să existe pentru aceeaşi problemă, programe dezvoltate folosind alte modele care să fie mai eficiente).

Implementările curente ale modelului BSP folosesc biblioteci SPMD bazate pe C sau Fortran. În modelul Oxford BSP, se furnizează operaţii `put` care pun o dată în memoria

locală a unui alt procesor, operaţii *get* care preiau o dată din memoria locală a unui alt procesor şi operaţii pentru sincronizare. În continuare, este prezentat un exemplu pentru calculul sumelor prefix:

```
int prefixsums(int x) {
   int i, left, right;
   bsp_pushregister(&left, sizeof(int));
   bsp_sync();
   right = x;
   for(i=1;i < bsp_nprocs();i* = 2) {
      if (bsp_pid()+i < bsp_nprocs())
         bsp_put(bsp_pid()+i,&right, &left, 0, sizeof(int));
      bsp_sync();
      if (bsp_pid() >= i) right = left +right;
   }
   bsp_popregister(&left);
   return right;
}
```

Operaţiile `bsp_pushregister` şi `bsp_popregister` sunt folosite pentru a da posibilitatea fiecărui proces să refere variabile dintr-un alt proces prin nume, chiar dacă acestea au fost alocate în memoria heap sau pe stivă.

O abordare asemănătoare este LogP [41] care foloseşte procese similare, cu contexte locale, actualizate de comunicaţii globale. Nu sunt folosite bariere de sincronizare globale în LogP. Modelul LogP a fost construit ca un model abstract care să capteze realitatea tehnologică a calculului paralel. Se folosesc patru parametrii: întârzierea (L), timpul de overhead (o), lărgimea de bandă (g) şi numărul de procesoare (P). Cu toate acestea, LogP nu este un model mai puternic decât BSP, care este mult mai simplu.

9.2.4 Mapare explicită

Modelele din această clasă specifică descompunerea programelor în componente şi de asemenea şi cum sunt asignate aceste componente de calcul pe procesoare, dar furnizează câteva abstractizări referitor la operaţiile de comunicaţie. Partea dificilă a descrierii unei comunicaţii constă în necesitatea specificării celor două etichete ale capetelor fiecărei comunicaţii şi asigurării potrivirii între aceste specificaţii. Modelele din această clasă încearcă să simplifice această descriere fie prin decuplarea dintre cele două capete ale fiecărei comunicaţii, fie prin furnizarea unui nivel înalt de abstractizare dat de şabloane de comunicaţii, fie prin furnizarea unor modalităţi mai bune pentru specificarea comunicaţiei.

Modele cu structură dinamică

Limbajele de coordonare simplifică comunicaţia prin separarea aspectelor ce ţin de calcul de cele ce ţin de comunicaţie şi prin furnizarea unui limbaj separat pentru specificarea comunicaţiei. Această separare face ca partea de calcul şi cea de comunicaţie să fie ortogonale una faţă de cealaltă şi astfel un anumit stil de coordoonare poate fi aplicat la orice limbaj secvenţial.

Cel mai cunoscut exemplu este Linda [29] care înlocuieşte comunicaţia punct-la-punct cu un spaţiu partajat în care valorile datelor sunt plasate de către procesoare şi de unde sunt extrase în mod asociativ. Acest spaţiu este numit *spaţiu de tuple ("tuple space")*. Modelul de comunicare Linda conţine trei operaţii:

1. *in* – prin care se şterge un tuplu din spaţiul de tuple, în funcţie de aritatea sa şi de valorile anumitor câmpuri;
2. *read (rd)* – care este similară cu *in*, doar că se copiază tuplul corespunzător din spaţiul de tuple;
3. *out* – prin care se scrie un tuplu în spaţiul de tuple.

De exemplu, operaţia *read*: `rd("Romania", ?X, "Europa")` caută în spaţiul de tuple tuplele cu trei elemente dintre care primul este egal cu "Romania", ultimul cu "Europa" şi cel din mijloc de acelaşi tip cu variabila X. În afara acestor operaţii, Linda defineşte şi o operaţie `eval(t)` prin care se creează implicit un nou proces care evaluează tuplul t şi îl inserează în spaţiul de tuple.

Operaţiile limbajului Linda separă părţile de transmisie şi recepţie a unei comunicaţii – procesul care "transmite" nu are cunoştinţă de procesul care "recepţionează" şi nici nu ştie dacă acesta există. Modelul Linda impune gestiunea descompunerii, dar reduce complexitatea comunicaţiei. Din păcate, un spaţiu de tuple nu este în mod obligatoriu eficient implementabil şi nu se pot furniza măsuri de cost. Chiar mai mult, este posibilă situaţia de "deadlock" în Linda.

Pentru Linda s-a încercat şi dezvoltarea unei metodologii de dezvoltare a softului. Linda Program Builder (LPB) este un mediu de programare de nivel înalt care ajută la proiectarea şi dezvoltarea programelor Linda [4]. Acest mediu ghidează programatorul în proiectarea, codificarea, monitorizarea şi execuţia programelor Linda.

Limbajele cu comunicaţie nebazată pe mesaje reduc overhead-ul de gestiune al comunicaţiilor prin tratarea comunicaţiilor în mod mai natural firelor de execuţie. De exemplu, ALMS tratează trasmiterea de mesaje ca şi când canalele de comunicaţie ar fi mapate în memorie. Referinţa la anumite variabile mesaj care apar în diferite fire de execuţie se comportă ca şi un transfer de mesaj de la un fir la celelalte. PCN şi Compositional C++ ascund de asemenea comunicaţiile prin folosirea variabilelor de unică folosinţă. Încercarea de a citi dintr-una din aceste variabile blochează firul dacă nu s-a plasat anterior o valoare în variabilă de către alt fir.

Limbajul PCN este bazat pe două concepte foarte simple: compoziţie concurentă şi variabile cu o singură atribuire. Aceste variabile se numesc variabile *definiţie*. Compoziţia concurentă permite execuţia paralelă a blocurilor de instrucţiuni specificate, fără a preciza cum vor fi mapate pe procesoare aceste blocuri. Procesele care partajează o variabilă

definiţie pot comunica prin intermediul ei. De exemplu, în următoarea compoziţie paralelă

```
{|| producator(X), consumator(X)}
```

cele două procese `producator` şi `consumator` pot folosi variabila `X` pentru a comunica indiferent de locaţia lor pe calculatorul paralel.

Virtual shared memory este extensia logică a mapării comunicaţiei în memorie. În acest caz se foloseşte o abstractizare care defineşte un singur spaţiu de adrese partajat.

Limbajele funcţionale adnotate permit programatorilor să furnizeze informaţie în plus despre cele mai potrivite moduri de a descompune calculul în componente şi de plasare a lor. În acest caz, sarcina compilatorului este mult uşurată. Se folosesc tot regulile clasice de reducere şi astfel comunicaţia şi sincronizarea determinată de mapare decurge în acelaşi mod ca şi în cazul reducerii de graf pure.

Un exemplu de asemenea limbaj este limbajul funcţional Paralf, care se bazează pe evaluare întârziată, la cerere. Totuşi limbajul permite utilizatorului să controleze ordinea de evaluare prin adnotări explicite. În Paralf comunicaţia şi sincronizarea sunt implicite, dar furnizează o notaţie de mapare prin care se specifică ce expresii să fie evaluate şi pe ce procesor. De exemplu, expresia

$$(f(x)\ \$\text{on}\ (\$\text{self} + 1)) * (h(x)\ \$\text{on}\ (\$\text{self}))$$

specifică faptul că expresia $f(x)$ se va calcula pe procesorul vecin, în paralel cu calculul expresiei $h(x)$.

Remote Procedure Call (RPC) este mecanism care reprezintă o extensie a clasicului apel de procedură. Un RPC este un apel de procedură între două procese distincte: apelant şi receptor. Când un proces apelează o procedură la distanţă de pe alt proces, receptorul execută codul corespunzător procedurii şi trimite procesului apelant parametrii de ieşire. Ca şi conceptul de "rendezvous", RPC este o formă de cooperare sincronă. În timpul execuţiei procedurii procesul apelant este blocat şi este reactivat doar după primirea rezultatelor. Sincronizarea completă RPC poate duce la limitarea exploatării la un nivel înalt a paralelismului dintre procesele care sunt parte a unui program concurent. Pentru a micşora acest efect, majoritatea sistemelor bazate pe RPC folosesc fire de execuţie. Exemple de limbaje care folosesc RPC sunt: DP, Cedar şi Concurrent CLU.

Modele cu structură statică

Limbajele grafice simplifică descrierea comunicaţiilor prin faptul că permit ca acestea să fie inserate grafic la un nivel înalt şi structurat. De exemplu, limbajul Enterprise clasifică unităţile de program pe baza tipului şi inserează o structură de comunicaţie automat bazat pe tip. Limbajul Parsec permite unităţilor de program să fie conectate pe baza unei mulţimi de şabloane de conectare predefinite. Descompunerea în aceste modele este explicită, dar comunicaţia este mai simplu de descris. Şabloanele de comunicaţie se aleg în funcţie de necesităţi, mai degrabă decât bazat pe eficienţă şi astfel nu pot fi garantate implementări eficiente şi nici măsuri de cost.

Limbajele de coordonare cu contexte extind ideea folosită de limbajul Linda. Una din slăbiciunile existente în Linda este că se furnizează doar un singur spaţiu de tuple global şi astfel se exclude dezvoltarea modulară. Limbajul Ease extinde modelul Linda preluând idei de la limbajul Occam. Ease lucrează cu mai multe spaţii de tuple numite *contexte* şi care sunt vizibile doar anumitor fire de execuţie. Deoarece firele de execuţie care pot folosi un context particular sunt cunoscute, contextele pot prelua anumite proprietăţi ale canalelor definite în Occam. Firele de execuţie citesc şi scriu date în contexte ca şi cum ar fi spaţii de tuple, folosindu-se potrivirea asociativă. Dar se poate folosi şi un alt set de primitive prin care se mută date într-un context şi se poate renunţa la statutul de proprietate asupra datelor, sau se pot extrage date dintr-un context odată cu ştergerea lor. Asemenea operaţii pot folosi transmiterea prin referinţă, pentru a se garanta faptul că datele sunt referite doar de un fir de excuţie la un moment dat. Limbajul Ease păstrează multe din proprietăţile din Linda, dar permite implementări mai eficiente. Descompunerea este de asemenea simplificată prin structurarea proceselor în stil Occam.

Un alt limbaj asemănător este ISETL-Linda care este o extensie a limbajului SETL. Se aseamănă cu un limbaj cu paralelism de date, în care colecţiile sunt un tip de date şi potrivirea asociativă este o operaţie de selecţie pe colecţii. Astfel ISETL-Linda poate fi considerat ca un limbaj de tip SETL cu un nou tip de date, sau ca o extindere a unui limbaj de tip Linda cu şabloane ("skeletons").

Opus este un limbaj derivat din Fortran care se bazează atât pe paralelism de date cât şi pe paralelism funcţional, dar comunicaţiile sunt realizate prin abstractizări de date partajate. Acestea sunt obiecte autonome, vizibile oricărei submulţimi de componente de calcul; doar o componentă din fiecare obiect este activă la un anumit moment. Aceste abstractizări reprezintă un tip de generalizări ale monitoarelor.

Modele cu structură statică şi comunicaţie limitată

Şabloanele de comunicare extind ideea blocurilor prestructurate la comunicaţie. Un şablon de comunicare reprezintă o întreţesere a unor paşi de calcul (care constau în calcule locale independente) cu paşi de comunicaţie (care se bazează pe modele de comunicaţie într-o topologie abstractă). Acest model compune idei preluate din BSP şi din modelele bazate pe şabloane algoritmice, precum şi concepte cum sunt rutarea adaptivă şi broadcast. Pot fi implementate eficient şi pot defini măsuri de cost.

9.2.5 Comunicaţie explicită

Modelele din această clasă explicitează comunicaţia, dar reduc complicaţiile legate de sincronizare. În general aceasta se obţine prin folosirea unei semantici asincrone.

Modele cu structură dinamică

Reţelele de procese ("process nets") se aseamănă cu modelele bazate pe flux de date (*dataflow*) prin faptul că operaţiile sunt, în cele două cazuri, entităţi care răspund la venirea datelor prin calcul şi eventual prin transmiterea altor date. În cazul reţelelor de

procese însă operaţiile pot decide individual care va fi răspunsul lor la venirea datelor şi pot de asemenea să decidă individual schimbarea comportamentului lor. Nu există deci starea globală, care pentru modelele bazate pe flux de date, există cel puţin implicit.

Modelele bazate pe *actori* sunt principalele modele din această clasă. Sistemele actor constau în colecţii de obiecte numite actori, fiecare având o coadă de mesaje de intrare. Un actor repetă următoarea secvenţă de operaţii: citeşte următorul mesaj de intrare, trimite mesaje către alţi actori ale căror identităţi le cunoaşte şi defineşte o nouă comportare care guvernează răspunsul pentru următorul mesaj. Mesajele se transmit asincron şi fără o anumită ordine. Pentru a se putea obţine, însă, implementări eficiente ar trebui să se restricţioneze comunicaţia globală, dar acest lucru este dificil datorită naturii distribuite a modelului. Tot datorită acestui lucru cât şi a sistemului de transmitere de mesaje, introducerea de măsuri de cost este imposibilă. Modelul actor este de nivel jos, dar este modular şi simplu.

Limbajul Darwin, bazat pe *pi*-calcul, foloseşte alt tip de reţea de procese. Limbajul furnizează o submulţime de configurare, bine fundamentată semantic, pentru specificarea modului de conectare a proceselor ordinare şi a modului lor de comunicare. Legătura dintre semantica comunicaţiei şi conexiuni este dinamică, spre deosebire de alte limbaje de configurare.

Una dintre slăbiciunile modelelor actor este faptul că actorii îşi procesează coada de mesaje secvenţial şi aceasta poate duce la blocări. S-au propus modele de extindere, cum sunt Concurrent Aggregate şi ActorSpace, care tratează acestă problemă. Concurrent Aggregate (CA) este un limbaj orientat-obiect foarte potrivit pentru a trata paralelismul cu granularitate fină pe masive de procesoare. Cu acest limbaj s-a încercat înlăturarea surselor de secvenţialitate nenecesare. Un agregat în CA este o colecţie omogenă de obiecte (numite reprezentanţi) care pot fi referite printr-un singur nume. Fiecare agregat poate primi mai multe mesaje simultan. Sunt încorporate şi alte caracteristici inovative cum sunt: delegare, adresare intra-agregat, mesaje de primă clasă. Delegarea permite ca şi comportamentul unui agregat să poate fi construit incremetal din comportamentele altor agregate, iar adresarea intra-agregat permite cooperarea între părţi ale aceluiaşi agregat.

Modelul ActorSpace extinde modelul actor prin eliminarea sincronizărilor care nu sunt necesare. În modelul ActorSpace comunicaţiile sunt asincrone şi astfel un actor care trimite un mesaj nu trebuie să îşi blocheze execuţia până când receptorul este gata să preia sau să prelucreze mesajul. Prin necrearea dependenţelor de date nenecesare, abordarea dirijată de mesaje a acestui model permite exploatarea concurenţei maxime. Un spaţiu actor este o colecţie de actori, pasivă din punct de vedere computaţional, care acţionează ca un context pentru "pattern matching". Mesajele pot fi transmise unui membru arbitrar al grupului sau tuturor membrilor unui grup care corespund unui model ("pattern").

O posibilă abordare este de a extinde limbajele orientate-obiect astfel încât mai multe fire de execuţie să fie active la un moment dat. Acest lucru se poate realiza în două moduri. Primul, care a fost numit *orientare-obiect externă*, permite fire de control multiple la cel mai înalt nivel al limbajului. Starea unui obiect poate funcţiona în acest caz ca şi un mecanism de comunicare, deoarece poate fi modificată de o metodă executată de

un fir de execuţie şi observată de o metodă executată de un altul. A doua modalitate, numită *orientare-obiect internă*, încapsulează paralelismul în interiorul metodelor unui obiect şi astfel la nivel înalt, limbajul apare ca fiind secvenţial. Este astfel foarte strâns legat de paralelismul de date.

În cazul modelelor orientate-obiect extern, actorii sunt consideraţi indiferent dacă se comunică cu ei sau nu. Câteva exemple de asemenea limbaje sunt: ABCL/1, ABCL/R, POOL-T, EPL, Emerald şi Concurrent Smalltalk. În aceste limbaje, paralelismul se bazează pe atribuirea unui fir de execuţie fiecărui obiect şi concurenţa poate fi mărită prin transmiterea de mesaje asincrone.

În limbajele orientate-obiect extern, paralelismul poate fi exploatat în două moduri principale: folosind obiectele ca unităţi ale paralelismului prin atribuirea fiecărui obiect unul sau mai multe procesoare, sau prin definirea proceselor ca şi componente ale limbajului. Limbajele corespunzătoare primei abordări sunt bazate pe obiecte active. Fiecare proces este legat de un anumit obiect pentru care a fost creat. Folosind cea de-a doua abordare sunt definite două tipuri de entităţi: obiecte şi procese. Un proces nu este legat de un singur obiect, ci este folosit pentru a executa toate operaţiile necesare pentru a satisface o acţiune. Prin urmare, un proces se poate executa pentru mai multe obiecte, prin schimbarea spaţiului său de adrese atunci când se face o invocare de la alt obiect. Limbajele Argus şi Presto folosesc această a doua abordare. Aceste limbaje furnizează mecanisme de creare şi controlare a proceselor multiple, externe structurii de obiecte.

Mesaje active reprezintă o abordare care decuplează comunicaţia cât şi sincronizarea prin tratarea mesajelor ca obiecte active în loc de a le trata ca date pasive. În mod esenţial un mesaj activ constă din două părţi: o parte de date şi o parte de cod care se execută pe procesorul receptor când mesajul ajunge. Astfel mesajul se transformă într-un proces când ajunge la destinaţie. Nu există nici o sincronizare între procese şi astfel un mesaj *send* nu are un corespondent *receive*. Această abordare este folosită de sistemul Movie şi de mediile de limbaj pentru maşina J.

Modele cu structură statică

Limbajele orientate-obiect intern se încadrează în această categorie. Limbajul de programare Mentat (MPL) este un sistem paralel orientat-obiect proiectat pentru dezvoltarea de aplicaţii paralele independent de arhitectură. Acest sistem integrează un model de calcul dirijat de date în paradigma orientată-obiect. Modelul de calcul dirijat de date suportă un grad înalt de paralelism, iar orientarea obiect ascunde utilizatorului foarte mult din caracteristicile de paralelism. MPL este o extensie a limbajului C++, care suportă atât paralelism inter cât şi intra obiecte. Construcţiile limbajului sunt mapate modelului macro de flux de date care este modelul de calcul care stă la baza sistemului Mentat. Este un model cu granularitate medie, dirijat de date, în care programele sunt reprezentate ca şi grafuri orientate. Vârfurile grafurilor sunt elemente de calcul care realizează anumite funcţii. Muchiile evidenţiază dependenţele de date. Compilatorul generează cod pentru a construi şi executa aceste grafuri. Paralelismul între obiecte este în general transparent programatorului. În această abordare, programatorul ia deciziile de granularitate şi partiţionare folosind construcţiile limbajului, iar compilatorul şi sistemul de execuţie

gestionează comunicaţiile şi sincronizările.

Modele cu structură statică şi comunicaţie limitată

Reprezentanţi ai acestei clase sunt *tablourile sistolice*. Structura şi comunicaţia în asemenea sisteme au fost tratate în Capitolul 1, Secţiunea 1.1.1 şi Capitolul 4, Secţiunea 4.8. Un program paralel pentru un tablou sistolic trebuie să specifice cum sunt mapate datele pe elementele de procesare şi fluxul de date între elemente.

Sunt necesare modele de programare de nivel înalt pentru a promova folosirea pe scară largă a tablourilor sistolice programabile. Un astfel de exemplu este limbajul Alpha, unde programele sunt exprimate prin ecuaţii recurente. Acestea sunt trasformate în formă sistolică prin considerarea dependenţelor de date ca definind un spaţiu afin, care poate fi transformat geometric.

9.2.6 Totul explicit

În aceste modele programatorii trebuie să specifice toate detaliile legate de implementare. Este foarte dificil de construit aplicaţii folosind aceste modele datorită faptului că atât corectitudinea cât şi performanţa pot fi obţinute doar prin tratarea atentă a numeroase detalii.

Majoritatea din modelele de calcul de primă generaţie sunt de acest tip, fiind proiectate pentru un singur tip de arhitectură gestionată explicit.

Modele cu structură dinamică

Cele mai multe dintre aceste modele se bazează pe o anumită paradigmă pentru tratarea partiţionării, mapării şi comunicaţiilor. Există doar câteva modele care au încercat să furnizeze cadrul pentru abordarea mai multor paradigme. Un asemenea exemplu este limbajul Pi, care furnizează un set de primitive pentru fiecare stil de comunicaţie. Aceste modele pot fi implementate eficient şi dispun de asemenea de măsuri de cost, dar sarcina de construcţie a softului este dificilă datorită multitudinii de detalii care trebuiesc precizate pentru fiecare calcul. Un alt set de modele este reprezentat de limbajele de programare Orca şi SR. Orca este un limbaj bazat pe obiecte care foloseşte date de tip obiect partajate pentru comunicaţia interproces. Sistemul Orca este o mulţime de abstractizări structurată ierarhic. La cel mai de jos nivel operaţia broadcast este primitiva de bază, astfel încât scrierea într-o structură replicată se poate realiza rapid în sistem. La următorul nivel de abstractizare, datele partajate sunt încapsulate în obiecte pasive care sunt replicate în sistem. Paralelismul se exprimă în Orca prin crearea explicită a proceselor. Un proces nou poate fi creat prin intermediul instrucţiunii `fork`

```
fork proc name(params) [on (cpu_number)]
```

Partea opţională `on` specifică procesorul pe care să se execute procesul fiu. Parametrii specifică obiectele de date partajate care se vor folosi pentru comunicaţia între părinte şi

fiu.
Modelul SR ("Synchronizing Resources") se bazează pe conceptul de *resursă*. O resursă este un modul care poate conţine câteva procese. O resursă poate fi creată dinamic prin comanda `create`, iar procesele sale pot comunica prin folosirea semafoarelor. Procesele care aparţin de resurse diferite comunică folosind o mulţime restrânsă de operaţii definite explicit de program ca şi proceduri.

Există însă mult mai multe modele care se bazează doar pe o singură paradigmă de comunicaţie. Considerăm următoarele trei paradigme: transmitere de mesaje, memorie partajată şi *rendezvous*.

Transmiterea de mesaje reprezintă tehnologia de comunicaţie de bază pentru arhitecturile de tip MIMD şi deci modelele bazate pe transmitere de mesaje sunt disponibile pentru asemenea tipuri de maşini. Interfeţele sunt de nivel scăzut şi folosesc operaţii *send* şi *receive* pentru a specifica mesajul ce trebuie transmis, indentificatorii de procese şi adresele.

Sistemele bazate pe transmitere de mesaje se aseamănă foarte bine cu sistemele folosite pentru arhitecturile cu memorie distribuită. De aceea s-au construit interfeţe standard care să îmbunătăţească portabilitatea programelor cu transmitere de mesaje. Un exemplu destul de recent şi foarte folosit este MPI ("Message Passing Interface") [179] care furnizează un set bogat de primitive, în care sunt incluse comunicaţii punct-la-punct, broadcasting, abilitatea de a aduna procese în grupuri şi de a comunica doar în interiorul fiecărui grup. MPI a fost definit pentru a deveni un standard pentru interfaţa de transmitere de mesaje pentru aplicaţiile paralele şi biblioteci. Comunicaţiile punct-la-punct se bazează pe primitivele `send` şi `receive`

```
MPI_Send(buf, bufsize, datatype, dest, ...)
MPI_Receive(buf, bufsize, datatype, dest, ...)
```

MPI furnizează şi primitive pentru comunicaţii colective şi sincronizări, cum sunt `MPI_Bcast`, `MPI_Gather`, `MPI_Scatter` şi `MPI_Barrier`. În prima versiune MPI1 nu existau primitive care să permită crearea dinamică a proceselor, dar în versiunea MPI2 s-au adăugat primitive pentru mesaje active, crearea dinamică a proceselor, pornirea proceselor.

Multe din sistemele independente de arhitectură bazate pe transmiterea de mesaje au fost dezvoltate astfel încât să permită folosirea transparentă a reţelelor de staţii de lucru. În principiu, aceste reţele au multă putere de calcul nefolosită, care poate fi exploatată. În practică, întârzirea mare generată de comunicaţiile între staţiile de lucru face ca aceste arhitecturi să nu asigure o performanţă înaltă. Modelele pentru transmitere de mesaje între staţii de lucru includ sisteme cum sunt PVM, Parmacs şi p4. Aceste modele sunt la fel ca şi sistemele cu transmitere de mesaje inter-multiprocesor, doar că în general procesele au granularitate mai mare pentru a se încerca ascunderea întârzierilor ("latency") şi de asemenea ele trebuie să rezolve heterogenitatea procesoarelor. De exemplu, PVM ("Parallel Virtual Machine") [180] este foarte mult folosit şi este acceptat în general ca fiind instrumentul de programare pentru calcul distribuit heterogen. El furnizează un set de primitive pentru creare de procese şi comunicaţii care pot fi incorporate într-un

limbaj procedural existent, pentru implementarea programelor paralele. În PVM crearea de procese se face prin apelul procedurii `pvm_spawn()`. Instrucţiunea

```
proc_num=pvm_spawn("prog1", NULL, PVMTaskDefault, 0, n_proc)
```

creează `n_proc` copii ale programului `prog1`. Numărul real de procese pornite este dat de `proc_num`. Comunicaţia între două procese poate fi implementată prin primitivele:

```
pvm_send(proc_id, msg)
pvm_receive(proc_id, msg)
```

Pentru comunicaţiile de grup şi sincronizări se pot folosi operaţiile `pvm_bcast()`, `pvm_mcast()`, `pvm_barrier()`.

Folosind PVM şi alte modele similare, programatorii trebuie să precizeze descompunerea, maparea şi comunicaţia explicit. Acest lucru poate fi chiar foarte complicat deoarece se poate lucra cu diferite sisteme de operare. Aceste modele pot deveni mai eficiente prin folosirea în mai mare măsură a interconectării optice pentru legarea staţiilor de lucru.

Comunicaţia bazată pe *memorie partajată* este o extensie naturală a tehnicilor folosite de sistemele de operare, înlocuindu-se însă multiprogramarea cu multiprocesarea. Prin urmare, modelele care folosesc această paradigmă sunt uşor de înţeles. Sunt totuşi anumite aspecte care se schimbă când este vorba de paralelism. Calculatoarele paralele cu memorie partajată folosesc, în general pentru comunicare paradigmele standard cum sunt variabilele partajate şi semafoarele. Acest model de calcul este deosebit de atractiv deoarece etapele de descompunere şi mapare nu sunt atât de dificile. Totuşi, aceste modele sunt legate strâns de un singur tip de arhitecturi şi deci programale bazate pe memorie partajată nu sunt portabile.

Un exemplu important de acest tip îl reprezintă Java, care a devenit foarte popular datorită aplicaţiilor Web care pot fi construite cu ajutorul lui. Java permite crearea firelor de execuţie, care pot comunica şi se pot sincroniza. Aceste variabile partajate se pot accesa în interiorul metodelor sau secţiunilor sincronizate. Aceste secţiuni sunt incluse în secţiuni critice care sunt create automat. Secţiunile de acest tip sunt numite monitoare, deşi operaţiile `notify` şi `wait` trebuie să fie invocate explicit în aceste secţiuni şi nu asociate automat la intrare şi ieşire.

Există însă numeroase alte sisteme bazate pe fire de execuţie care se bazează pe comunicaţie cu memorie partajată.

Modelele de programare bazate pe conceptul de *rendezvous* se bazează pe paradigma memoriei distribuite şi folosesc un mecanism de cooperare particular. În modelul de comunicaţie rendezvous, o interacţiune între două procese A şi B are loc atunci când A apelează o *intrare ("entry")* a lui B, iar B execută o operaţie de acceptare (*"accept"*) pentru aceea intrare. Un apel de intrare este similar cu un apel de procedură, iar o instrucţiune de acceptare a celei intrări conţine o listă de instrucţiuni ce trebuie executate atunci când este apelată intrarea. Cele mai cunoscute limbaje de programare bazate pe conceptul de rendezvous sunt Ada şi Conccurent C. Ada [124] a fost proiectat pentru Departamentul apărării SUA pentru a permite programarea aplicaţiilor în timp real atât

pe calculatoare secvenţiale cât şi pe cele paralele. Paralelismul în Ada se bazează pe procese numite *taskuri*. Un task poate fi creat dinamic sau poate fi declarat static. În cel de-al doilea caz, taskul este activat atunci când se intră în blocul care conţine declaraţia sa. Taskurile sunt compuse dintr-o parte de specificare şi un corp. Declaraţiile de intrare *entry* pot fi făcute doar în partea de specificare a unui task, iar instrucţiunile de acceptare pot apare în corpul taskului. De exemplu, următoarea instrucţiune de acceptare execută operaţia de înmulţire specificată atunci când intrarea `square` este apelată:

```
accept SQUARE(X: INTEGER, Y: out INTEGER) do
   Y := X * X;
end;
```

Ada nu permite specificarea mapării taskurilor pe procesoare şi nu furnizează condiţii care să poată fi asociate cu declaraţiile *entry*.

Modele cu structură statică

Un exemplu de asemenea model este Occam [93], în care structura proceselor este fixă şi comunicaţia are loc folosindu-se canale sincrone. Programele Occam se construiesc dintr-un număr mic de primitive: atribuire, input (?) şi output (!). Pentru a se proiecta procese paralele complexe, trebuie să se folosească construcţia paralelă `PAR` împreună cu primitivele. Două procese pot fi executate în paralel dacă se specifică

```
PAR
      Proc1
      Proc2
```

construcţie care se termină doar după ce toate componentele sale se termină. Construcţia `ALT` permite implementarea nedeterminismului. Se aşteaptă o intrare de la un număr de canale de comunicaţie şi apoi se execută componenta proces corespunzătoare. De exemplu, prin codul

```
ALT
   request ?  data
      DataProc
   exec ?  oper
      ExecProc
```

se aşteaptă primirea unei date sau a unei operaţii, iar apoi se va executa procesul corespunzător gărzii selectate.

Limbajul Occam are un fundament semantic foarte puternic, bazat pe CSP ("Communicating Sequential Programs") [89] şi astfel dezvoltarea softului prin transformări este posibilă. Totuşi, este de nivel jos şi este practic doar pentru un număr mic de tipuri de aplicaţii.

Sumar

Analiza realizată de D. Skillicorn şi D. Talia se bazează pe structurarea modelelor în funcţie de cele şase criterii enunţate la început. Patru dintre aceste criterii se referă la posibilitatea folosirii modelelor pentru dezvoltarea softului paralel, acestea fiind: uşurinţa programării, existenţa unei metodologii de construcţie corectă a softului, independenţa de arhitectură şi abstractizarea. Celelalte două criterii au vedere execuţia modelelor pe maşini paralele reale şi se referă la implementare eficientă şi existenţa unor măsuri de cost.

Această analiză scoate în evidenţă faptul că dezvoltarea modelelor de calcul paralel pleacă de la abordări de nivel jos dar tinde spre abordări mult mai abstracte.

Sunt posibile, de asemenea, şi alte clasificări ale modelelor paralele existente, de exemplu, în funcţie de paradigma de programare de la care se pleacă: imperativă, funcţională sau logică.

Anexă

Noţiuni de teoria grafurilor

Graf: 1) un **graf neorientat** G este o pereche ordonată de mulţimi (X, U), unde X este o mulţime finită, numită mulţimea **vârfurilor** sau a **nodurilor**, iar U este formată din perechi neordonate de elemente distincte din X, numite **muchii**. O muchie fiind notată cu $[x, y]$, nodurile x şi y se numesc **extremităţile** acestei muchii. Dacă $[x, y] \in U$, se spune că nodurile x şi y sunt **adiacente** în graful G, iar nodurile x şi y sunt **incidente** cu muchia $[x, y]$.
2) Un **graf orientat** G este o pereche ordonată de mulţimi (X, U), unde X se numeşte mulţimea **vârfurilor** sau a **nodurilor**, iar U este formată din perechi ordonate de elemente distincte din X, numite **arce**. Un arc fiind notat cu $u = (x, y)$, nodul x este **extremitatea iniţială**, iar nodul y **extremitatea finală** a arcului u, care se spune că este orientat de la x la y. Dacă $(x, y) \in U$, nodurile x şi y sunt **adiacente** în G, şi amândouă sunt **incidente** cu arcul (x, y).

Graf parţial al unui graf $G = (X, U)$ este un graf $G_1 = (X, V)$, unde $V \subseteq U$, deci este graful însuşi sau se obţine din G prin suprimarea unor muchii (arce).

Subgraf al unui graf $G = (X, U)$ este un graf $H = (Y, V)$, unde $Y \subseteq X$, iar muchiile (arcele) din V sunt toate muchii (arce) din U care au ambele extremităţi în mulţimea de noduri Y.

Grad al unui nod x: 1) pentru un graf neorientat G gradul nodului x , notat cu $d(x)$, este numărul muchiilor incidente cu x;
2) Pentru un graf orientat G, gradul de intrare al nodului x, notat cu $d^-(x)$, este numărul arcelor care intră în nodul x, de forma (y, x); gradul de ieşire al nodului x, notat cu $d^+(x)$, este numărul arcelor de forma (x, y), care ies din nodul x; iar gradul nodului este $d(x) = d^-(x) + d^+(x)$.

Nod izolat este un nod de grad zero al unui graf.

Nod terminal este un nod de grad unu al unui graf.

Drum pentru un graf orientat $G = (X, U)$, este un şir de noduri $D = (x_0, x_1, \ldots, x_r)$ cu proprietatea că $(x_0, x1), (x_1, x_2), \ldots, (x_{r-1}, x_r)$ sunt arce ale grafului. Nodurile x_0 şi x_r se numesc **extremităţile drumului** D. **Lungimea** unui drum este dată de numărul de arce pe care le conţine. Dacă nodurile $x_0, x_1, \ldots, x_r$ sunt distincte două câte două, atunci drumul D este elementar.

Circuit pentru un graf orientat G este un drum $D = (x_0, \ldots x_r)$ cu proprietatea că $x_0 = x_r$ şi toate arcele $(x_0, x1), (x_1, x_2), \ldots, (x_{r-1}, x_r)$ sunt distincte două câte două.

Lanţ: 1) pentru un graf neorientat $G = (X, U)$ este un şir de noduri $L = [x_0, x_1, \ldots, x_r]$ cu proprietatea că oricare două noduri vecine sunt adiacente, adică $[x_0, x1], [x_1, x_2], \ldots,$ $[x_{r-1}, x_r] \in U$. Nodurile x_0 şi x_r se numesc **extremităţile lanţului**, iar r este **lungimea** acestui lanţ. Dacă nodurile $x_0, x_1, \ldots, x_r$ sunt distincte două câte două, atunci lanţul L este elementar.
2) pentru un graf orientat $G = (X, U)$, un lanţ $L = [u_1, \ldots, u_p]$ este un şir de arce, cu proprietatea că oricare două arce vecine u_i şi u_{i+1} au o extremitate comună pentru orice $1 \leq i < p$. Extremitatea lui u_1 care nu este extremitate şi pentru u_2 şi extremitatea lui u_p care nu este extremitate şi pentru u_{p-1} se numesc **extremităţile lanţului** L.

Ciclu pentru un graf neorientat G este un lanţ $L = [x_0, x_1, \ldots, x_r]$ cu proprietatea că $x_0 = x_r$ şi toate muchiile $[x_0, x1], [x_1, x_2], \ldots, [x_{r-1}, x_r]$ sunt distincte două câte două.

Lanţ (drum) hamiltonian este un lanţ (drum) elementar al unui graf, care conţine toate nodurile grafului.

Ciclu (circuit) eulerian al unui graf G este un ciclu (circuit) elementar care foloseşte toate muchiile (arcele) grafului G.

Ciclu (circuit) hamitonian al unui graf G este un ciclu (circuit) elementar care conţine toate nodurile grafului.

Graf conex este un graf G cu proprietatea că oricare două noduri sunt extremităţile unui lanţ al lui G.

Distanţa între nodurile x şi y ale unui graf neorientat şi conex G se notează cu $d(x, y)$ şi reprezintă lungimea minimă a lanţurilor cu extremităţile x şi y din G.

Diametrul unui graf conex G este distanţa maximă între perechile de noduri ale lui G şi se notează cu $d(G)$.

Arbore se numeşte orice graf neorientat conex şi fără cicluri.

Arbore cu rădăcină este un arbore în care unul dintre noduri este evidenţiat şi se numeşte rădăcina arborelui, iar între orice nod x şi rădăcină există un lanţ unic. Într-un

arbore T cu rădăcina r, orice nod y de pe unicul lanţ de la r la un nod x, este numit **strămoş** al lui x. Dacă y este un strămoş al lui x, atunci x este un **descendent** al lui y. **Subarborele cu rădăcină** x este arborele indus de către descendenţii lui x şi având rădăcina x. Dacă ultima muchie de pe lanţul de la radăcina r a unui arbore T până la un nod x este (y, x), atunci y este **părintele** lui x, iar x este un **copil** al lui y. Rădăcina este singurul nod fără părinte. Dacă două noduri au acelaşi părinte, atunci ele se numesc **fraţi**. Un nod fără nici un copil se numeşte **extern (terminal sau frunză)**. Un nod care nu este frunză se numeşte **nod intern**. Numărul copiilor unui nod x dintr-un arbore cu rădăcină T se numeşte **gradul** lui x. Lungimea lanţului de la rădăcina r la un nod x constituie **adâncimea** lui x în T. Cea mai mare adâncime a unui nod constituie **înălţimea** arborelui T. Nodurile cu aceeaşi adâncime formează un **nivel** al arborelui.

Arbore ordonat este un arbore cu rădăcină în care copii fiecărui nod sunt ordonaţi.

Arbore binar este o structură arborescentă definită pe o mulţime finită de noduri care

- nu conţine nici un nod, sau
- este constituită din trei muţimi de noduri distincte: un nod **rădăcină**, un arbore binar numit **subarborele stâng** şi un arbore binar numit **subarborele drept**.

Arbore poziţional este un arbore cu rădăcină în care toţi copii unui nod sunt etichetaţi cu numere întregi pozitive. Al i-lea copil al unui nod este absent dacă nici un copil al acelui nod nu este etichetat cu numărul i.

Arbore de acoperire (parţial) al unui graf G este un graf parţial al său care este şi arbore.

Arbore k-ar este un arbore poziţional în care, pentru fiecare nod, toţi copii cu etichete mai mari decât k lipsesc. Astfel, un arbore binar este un arbore k ar cu $k = 2$.

Arbore k-ar complet: este un arbore k-ar în care toate frunzele au aceeaşi adâncime şi toate nodurile interne au gradul k.

Arbore binar echilibrat este un arbore binar cu proprietatea că înălţimea subarborelui său stâng nu diferă cu mai mult de ± 1 de înălţimea subarborelui său drept.

Arbore binar total echilibrat este un arbore binar care are toate nodurile terminale pe ultimele două niveluri, astfel încât, pentru orice nod, numărul nodurilor din subarborele său stâng diferă cu cel mult 1 de numărul nodurilor din subarborele său drept.

Bibliografie

[1] M. Abadi, L. Lamport. Conjoining Specifications. *ACM Transaction on Programming Languages and Systems*, Vol. 17, No. 3, May 1995, pages 507-534.

[2] K. Abrahamson, N. Dadoun, D.A. Kirkpatric, T. Przytycka. A simple parallel tree contraction algorithm, In *Proc.* 25^{th} *ann. Allerton Conf. on Communication, Control and Computing*, 1987, pp. 624-633.

[3] R.C. Agarwal, F.G. Gustavson, S.M. Balle , M. Joshi, P. Palkar. A High Performance Matrix Multiplication Algorithm for MPPs. In *Proceedings of PARA'95*, pp. 1-7, 1995.

[4] S. Ahmed, N. Carriero, D. Gelernter. A program building tool for parallel applications. In *DIMACS Workshop on Specification of Parallel Algorithms*, pp. 161-178, Princeton University, USA, 1994.

[5] A.V. Aho, J.E. Hopcroft, J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[6] A. V. Aho, J.E. Hopcroft, J.D. Ullman. *Data Structures and Algorithms.*Addison-Wesley, 1983.

[7] S.G. Akl. *Parallel sorting algorithms*, Academic Press, 1985.

[8] G. Almasi, A. Gottlieb. *Highly parallel computing.* The Benjamin Cummings Publishing Company, Redwood City , 1989.

[9] G. Amdahl. Validity of the single processor approach of achieving large scale computing capabilities. In *Proceedings of AFIPS Conf.*, 30, 1967, pp. 483-485.

[10] W. Amsbury. *Data Structures.* Wadsworth Publishing Company, 1985.

[11] Sara Baase. *Computer Algorithms: Introduction to Design and Analysis*, Addison-Wesley, 1988.

[12] M. Barr, C. Wells, *Category Theory for Computing Science.* Prentice Hall, 1990.

[13] R. Barret, M. Berry, T.F. Chan, J.Demmel, J.Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, 1994.

[14] K.E. Batcher. Sorting networks and their applications. In *Proceedings AFIPS 1968 Spring Joint Comp. Conf.*, vol. 32, pp.307-314, 1968.

[15] G. Bell, The Future of High Performance Computers in Science and Engineering, *Comm. ACM*, Vol. 32, pp. 1091-1101, 1989.

[16] M. Ben-Ari. *Principles of Concurrent and Distributed Programming.* Prentice Hall 1990.

[17] A. J. Bernstein. Program Analysis for Parallel Processing. *IEEE Trans. on Electronic Computers*, EC-15, Oct 66, 757-762.

[18] R. Bird. An Introduction to the Theory of Lists. In M. Broy editor, *Logic of Programming and Calculi of Discret Design*, pp. 3-42. Springer-Verlag, 1987.

[19] R. Bird. Lectures on Constructive Functional Programming. In M. Broy editor, *Constructive Methods in Computing Science*, NATO ASI Series F: Computer and Systems Sciences, Vol. 55, pp. 151-216. Springer-Verlag, 1988.

[20] R. Bird. Algebraic Identities for Program Calculation. *The Computer Journal*, 32(2):122-126, February, 1989.

[21] G. Blelloch, Scans as Primitive Parallel Operations. *Proc. of the International Conference on Parallel Processing* August 1987, pp. 355–362, 1987.

[22] G. E. Blelloch. Programming Parallel Algorithms.*Communications of ACM*, March 1995/ Vol. 39, No. 3, pp. 85-97, 1995.

[23] G. E. Blelloch. B. M Maggs. Parallel Algorithms. *ACM Computing Surveys*, Vol. 28, No. 1, March 1996, pp. 51-54.

[24] G. H. Botorog, H. Kuchen. Efficient High-Level Parallel Programming. *Theoretical Computer Science* 196, pp. 71-107, 1998.

[25] W.W. Breckner. *Cercetări operaţionale*. Universitatea "Babeş-Bolyai", Cluj-Napoca, 1981.

[26] R.P. Brent. The Parallel Evaluation of General Arithmetic Expressions. *Journal of ACM*, 21 (2):201-206, 1974.

[27] J. Briat, M. Favre, C. Geyer, J.C. de Kergommeaux. Scheduling of OR-parallel Prolog on a scalable reconfigurable distributed memory multiprocessor. In *Proceedings of PARLE-91, Springer Lecture Notes in Computer Science 506*, pp. 385-402, 1991.

[28] M. Cannataro, G. Spezzano, D. Talia. A parallel logic system on a multicomputer architecture. In *Future Generation Computer Systems*, Vol. 6, pp. 317-331, 1991.

[29] N. Carriero, D. Gelernter. Application experience with Linda. In *ACM/SIGPLAN Symposium on Parallel Programming*, vol. 23, pp. 173-187, July 1988.

[30] M. Cesati, Miriam Di Ianni. Parameterized Parallel Complexity. In *Proceeedings of Euro-Par'98*, pp. 892-896, 1998.

[31] K.M. Chandy, J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley,1988.

[32] K.M. Chandy, S. Taylor, *An Introduction to Parallel Programming*, Jones and Bartlett Publishers, 1992.

[33] M. Chen, Y.I. Choo, J.Li. Crystal: Theory and pragmatics of generating efficient parallel code. In B.K. Szymanski editor, *Parallel Functional Languages and Compilers*, pp. 255-308, ACM Press Frontier Series, 1991.

[34] Ioana Chiorean. *Calcul Paralel*. Cluj-Napoca, Univ.Babes-Bolyai,1993.

[35] M. Cole. *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation*. PhD. thesis, University of Edinburgh, 1988.

[36] M. Cole. Parallel Programming with List Homomorphisms. *Parallel Processing Letters*, 5(2):191-204, 1994.

[37] Gh. Coman. *Analiză numerică*. Ed. Libris, Cluj-Napoca, 1995.

[38] Gh. Coman. On Some Parallel Methods in Linear Algebra. *Studia Univ. "Babes-Bolyai", Mathematica*, Vol. XXXVI, No. 3, 1991, pg.17-33.

[39] T.H. Cormen, C.E. Leiserson, R.R. Rivest. *Introductions to Algorithms*. Massachusetts Institute of Technology, 1990.

[40] J.S. Conery. *Parallel Execution of Logic Programs*. Kluwer Academic Publishers, 1987.

[41] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, T. von Eicken. LogP: Toward a realistic model of parallel computation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[42] O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare. *Structured programming*. Academic Press, 1972.

[43] M. Danelutto, R. di Meglio, S. Orlando, S. Pelagatti, M. Vanneschi. A methodology for the development and the support of massivelly parallel programs. *Future Generation Computer Science*, 1992; ap"arut "si ca "The P^3L language: an introduction", Hewlwtt-Packard report HPL-PSC-91-29, Dec. 1991.

[44] J. Darlington, A.J.Field, P.G. Harrison, P.H.J. Kelly, Q. Wu, R.L. While. Parallel programming using skeletons functions. In *PARLE'93, Parallel Architectures and Languages Europe*, June 1993.

[45] J. Darlington, Y. Guo, H.W. To, J. Yang. Functional Skeletons for Parallel Coordination, In *Proceedings of EuroPar'95*, LNCS 966, Springer, 1995.

[46] T. Delaitre, M.J. Zemerly, P. Vekariya, G.R. Justo, J. Bourgeois, F. Schinkmann, F. Spies, S. Randoux, S.C. Winter. EDPEPPS: A Toolset for the Design and Performance Evaluation of Parallel Applications. In *Proceedings of EuroPar'98*, pp. 113-125, 1998.

[47] E.W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs.*Communications of ACM*, 18(1975), no. 8, pp. 453-457.

[48] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englwood Clifs, 1976.

[49] Gh. Dodescu, M. Toma. *Metode de calcul numeric*. Ed. Didactică şi pedagogică, Bucureşti, 1976.

[50] Gh. Dodescu. *Metode numerice în algebră*. Ed. Tehnică, Bucureşti, 1979.

[51] B. Dumitrescu. *Algoritmi de calcul paralel*, `http://www.schur.pub.ro/download/ps/cursPS-5.3.ps`, 2001.

[52] Eisenbiegler, W. Lowe, W. Zimmermann. BSP, LogP, and Oblivious Programs. In *Proceedings of EuroPar'98*, pp. 865-874, 1998.

[53] K. Ekanadham. A perspective on Id. In B.K. Szymanski ed., *Parallel Functional Languages and Compilers*, pp. 197-254, ACM Press, 1991.

[54] J. Dana Eckart. Cellang 2.0: Reference manual. *ACM Sigplan Notices*, 27, No.8:107-112, 1992.

[55] P. Eleş, H. Ciocârlie. *Programare concurentă în limbaje de nivel înalt*. Ed Ştiiţifică, Bucureşti, 1991.

[56] T. Elrad, N. Frances. Decomposition of Distributed Programs into Communications Closed Layers. *Science of Computer Programming* (2):155-173, 1982.

[57] B.S. Fagin, A.M. Despain. The performance of parallel Prolog prgrams. *IEEE Transactions on Computers*, C-39, No. 12:1434-1445, 1990.

[58] F.E. Fich. New bounds for parallel prefic circuits. *In Proceedings of ACM Symposium on Theory of Computing*, pp 100-109, April 1983.

[59] G.C. Fox, S.W. Otto, J.G. Hey. Matrix Algorithms on a HipercubeI: Matrix Multiplication. *Parallel Computing* 4:17-31, 1987.

[60] I. Foster, S. Taylor. *Strand: New Concepts in Parallel Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1990.

[61] M. Flynn. Some Computers Organisations and their Effectiveness. *IEEE Transactions on Computers*, C-21(9), 1972, pp. 948-960.

[62] I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[63] G. Fox. Parallel Computing Comes of Age: Supercomputer Level Parallel Computations at Caltech. *Concurrency: Practice and Experience*, vol. 1(1), pg 63-103, 1989.

[64] A. Geser, S. Gorlatch. Parallelizing Functional Programs by Generalization. *Journal of Functional Programming*, în apariţie.

[65] J. Gibbson. The Third Homomorphism Theorem. *Journal of Functional Programming*, 6(4):657-665, 1996.

[66] J. Goguen, C. Kirchner, H. Kirchner, A. Megrelis, J. Mesenguer, T. Winkler. An introduction to OBJ3. In *Lecture Notes in Computer Science*, vol 308, pp. 258-263.

[67] L. Goldschlager. The monotone and planar circuit value problems are log space complete for P. *SIGACT News*, 9(2):25-29, 1977.

[68] S. Gorlatch. Extracting and Implementing List Homomorphisms in Parallel Programs Development. *Science of Computer Programming*, Vol. 33, no. 1, pp. 1-27, 1999.

[69] S. Gorlatch. *Optimizing Compositions of Scans and Reductions in Parallel Program Derivation*. Preprint Universitat Passau, April 1997.

[70] S. Gorlatch. *Abstraction and Performance in the Design of Parallel Programs*. Universitat Passau, MIP-9802, Januar, 1998.

[71] S. Gorlatch, C. Lengauer. Parallelization of Divide-and-Conquer in the Bird-Meertens Formalism. *Formal Aspects of Computing* 3, pp. 663-682, 1995.

[72] S. Gorlatch. Stages and Transformations in Parallel Programming. *Abstract Machine Models For Parallel and Distributed Programming*, Amsterdam, IOS Press pp. 147-162, 1996.

[73] S. Gorlatch. *Constructing List Homomorphisms for Parallelism*. Technical Report MIP-9512, Universitat Passau, 1995.

[74] S. Gregory. *Parallel Logic Programming in PARLOG*. Addison-Wesley, 1987.

[75] G. Grigoraş. *Programarea Calculatoarelor - Fundamente*, Editura "Spiru Haret", 1999.

[76] D. Grigoraş. *Calculul Paralel. De la sisteme la programarea aplicaţiilor*. Computer Libris Agora, 2000.

[77] R.D. Gumb. *Programming Logics*. John Wiley & Sons,1989.

[78] J.A. Gunnels, D.A. Katz. *Fault-Tolerant High-Performance Matrix Multiplication*. Report of University of Texas, Austin, 2000.

[79] J.A. Gunnels, G.H. Henry, R.A. van de Geijn. *High-Performance Matrix Multiplication Algorithms for Architectures with Hierarchical Memories*. Report of University of Texas, Austin, 2001.

[80] J.A. Gunnels. *A Systematic Approach to the Design and Analysis of Linear Algebra Algorithms*. PhD. Thesis, University of Texas, Austin, 2001.

[81] A. Gupta, M. Joshi, V. Kumar. WSSMP: A High-Performance Serial and Parallel Symmetric Sparse Linear Solver. In *Proceedings of PARA'98*, pp. 182-194, 1998.

[82] R.H. Halstead Jr. Parallel Symbolic Computing. *IEEE Computer*, 19, No.8, 1986.

[83] W. Handler, *Innovative computer architecture – how to increase parallelism but not complexity*, In Parallel Processing Systems, An Advanced course, Evans DJ ed., Cambridge Univ. Press, 1982, pp. 1-41.

[84] P.B. Hansen. Model Programs for Computational Science: A Programming Methodology for Multicomputers. *Concurrency: Practice and Experince*, vol. 5 (5), pp. 407-423, 1993.

[85] G. Henry. *Flexible High-Performance Matrix Multiply via a Self-Modifying Runtime Code.* Techical Report, Intel Corp. Computational Software Laboratory, 2001.

[86] P. van Hentenryck. Parallel constraint satisfaction in logic programming: Preliminary results of CHIP within PEPSys. In *Proceedings of the 6th International Congress on Logic programming*, pp. 165-180, Cambridge 1989.

[87] J. Herath, T. Yuba, N. Saito. Dataflow Computing. In *Parallel Algorithms and Architectures*, Springer Lecture Notes in Computer Science 269, pp. 25-36, MAy 1987.

[88] High Performance Fortran language specification. `titan.rice.cs.edu`, January 1993.

[89] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International Series in Computer Science, 1985.

[90] R. Hockney, C. Jesshope. *Calculatoare paralele. Arhitectura, programare, algoritmi.* Editura Tehnică, Bucureşti, 1991.

[91] K.E. Iverson. *A programming Language.* Wiley, New-York, 1962.

[92] J. Jaffar, J.L. Lassez. Constraint Logic Programming. In *ACM Symposium on Principles of Programming Languages*, pg.111-119, ACM Press, 1987.

[93] G. Jones, M. Goldsmith. *Programming in Occam2.* Prentice Hall, 1988.

[94] L.V. Kale. The REDUCE-OR process model for parallel evaluation of logic programs. In *Proceedings of the 4th International Conference on Logic Programming*, pp. 616-632, Melbourne, Australia, 1987.

[95] A. Kiper. An Efficient Parallel Triangular Inversion by Gauss Elimination with Sweeping. Proceedings of EuroPar'98, pp. 793-797, 1995.

[96] Knuth, D.E., *The Art of Computer Programming. Vol. 3 Sorting and Searching*, Addison-Wesley, 1973.

[97] D.E. Knuth. *The Art of Computer Programming, Vol. 2, Seminumerical algorithms.* Adison Wesley, 1979.

[98] J. Kornerup. *Mapping PowerLists onto Hypercubes.* Information Processing Letters, 53:153-158, 1995.

[99] J. Kornerup. *Data structures for Parallel Recursion* PhD thesis, University of Texas at Austin, 1997.

[100] J. Kornerup. Plist: Taking PowerLists Beyond Base Two. *CMPP'98 First International WorkShop on Constructive Methods for Parallel Programming*, MIP-9805 May 1998.

[101] S.R. Kosaraju, A.L. Delcher, Optimal parallel evaluation of tree-structured computation by ranking, In *VLSI Algorithms and Architectures, Proc.* 3^{rd} *Aegean Workshop on Computing*, Lecture Notes in Computer Science, Vol 319, 1988, pp. 101-110.

[102] A. R. Krommer. Parallel Sparse Matrix Computations Using the PINEAPL Library: A Performance Study. In *Proceedings of EuroPar'98*, pp. 804-811, 1998.

[103] R.E. Ladner, M.J. Fisher. Parallel prefix computation. *Journal of the ACM*, 27(4):831-838, October 1980.

[104] L. Lamport, How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs, *IEEE Trans. on Computers*, vol. C.-28, no. 9, pp. 690-691, 1979.

[105] L. Lamport, On Interprocess Communication. Parts I and II, *Distributed Computing*, vol. 1, no. 2 (1986), pp. 77-101.

[106] Lan Yang, Lan Jin. *Integrating Parallel Algorithm Design with Parallel Machine Models* ACM, pp. 131-135, 1995.

[107] R. Lee. Empirical Results on the Speed, Efficiency, Redundancy and Quality of Parallel Computations. *Int. Conf. on Parallel Processing*, 1980, pp. 91-100.

[108] T.G. Lewis. *Foundations of Parallel Programming. A Machine Independent Approach.* IEEE Computer Society Press, 1993.

[109] P. Lincoln, N. Marti-Oiliet, J. Mesengue. Specification, transformation, and programming of conccurent Systems in rewriting logic. Technical report SRI-CSL-94-11, SRI, May 1994.

[110] Barbara Liskov, J. Guttag. *Abstractions and Specification in Program Development* Massachusetts Institute of Technology, 1986.

[111] L.D.J.C. Loyens. *A Design Method for Parallel Programs.* Techniche Univ. Eindhoven, Proefschrift,1992.

[112] L.D.J.C. Loyens, R.H. Biesseling. The formal construction of a parallel triangular system solver. In *LNCS, Mathematics of Program Construction*, 375 pg.325–34, 1989.

[113] B.D. Lubacevscky, A. G. Greenbers. Simple, efficient asynchronous parallel prefix algorithms. In *Proceedings of International Conference on Parallel Processing*, pp. 66-69, August 1987.

[114] G.A. Mago. A network of computers to execute reduction languages. *International Journal of Computer and Information Sciences*, 1979.

[115] O. Maslennikov, J. Kaniewski, R. Wyrzykowski. *Fault Tolerant QR-Decomposition Algoritm and its Parallel Implementation* Proceedings of EuroPar'98, pp. 798-803, 1998.

[116] J. McGraw. Parallel functional programming in Sisal: Fictions, facts, and future. In *Advanced Workshop, Programming Tools for Parallel Machines*, June 1993.

[117] J. Mesenguer, T. Winkler. Parallel Programming in Maude. In J.P. Banatre, D, Le Metayer, editors, *Research Directions in High-level Parallel Programming Languages*, pp. 253-293, Springer Lecture Notes in Computer Science 574, June 1991.

[118] C. Mihu. *Sisteme de ecuaţii liniare şi forme pătratice* Ed. Tehnică, Bucureşti ,1985.

[119] M. Marin. *Asynchronous (Time-Wrap) Versus Syncrhonous (Event-Horizon) Simulation Time Advance in BSP.* Proceedings of EuroPar'98, pp. 897-905, 1998.

[120] G.L. Miller, J.H. Reif, Parallel tree contractionand its application, In *Proceedings of* 26^{th} *Ann. IEEE Symp. on Foundations of Computer Science*, 1985, pp. 478-489.

[121] J. Misra. PowerList: A structure for parallel recursion. *ACM Transactions on Programming Languages and Systems*, 16(6):1737-1767, November 1994.

[122] B. Monien, H. Sudborough. Embedding One Interconnection Network in Another. *Comuputing Suppl.* 7:257-282, 1990.

[123] Carroll Morgan. *Programming from Specification*. Prentice Hall, 1990.

[124] D.A. Mundie, D.A. Fisher. Parallel Processing in Ada. *IEEE Computer*, C-19 No.8:20-25, 1986.

[125] J. Nash. Scalable Sharing Methods Can Support a Simple Performance Model. *Proceedings of EuroPar'98*, pp. 906-915, 1998.

[126] P.A. Nelson, L. Snyder, Programming Paradigms for Non-Shared Memory Parallel Computers, In L.H.Jamieson, D. Gannon, R. Douglass (Eds.), *The Characteristics of Parallel Algorithms*, MIT Press,Cambridge, 1988.

[127] V. Niculescu. A Design Method for Parallel Programs. Applications. *Seminar on Numerical and Statistic Calculus*, Preprint no.1, 1996, pp. 61-77.

[128] V. Niculescu, Data Distributions for Parallel Programs. *Studia Universitatis, "Babeş-Bolyai", Informatica*, vol XLIII, No. 2, 1998, pp. 64-72.

[129] V. Niculescu. Parallel Programs Description with PowerList, ParList and Plist. *Studia Universitatis "Babeş-Bolyai", Informatica*, vol XLIV, No. 1, 1999, pp. 41-50.

[130] V. Niculescu. Boolean Matrices Multiplication. *Seminar of Numerical and Statistic Calculus*, Preprint no.1, 1999, pp. 89-96.

[131] V. Niculescu. Multidimensional Data Structures for Parallel Programs Description. *Journal of PU.M.A.* Vol. 11, No. 2, 2000, pp. 351-360.

[132] V. Niculescu. Some Nondeterministic Parallel Programs. *Studia Universitatis, "Babeş-Bolyai"*, Informatica, Vol. XLV, No. 2 , 2000, pp. 51-59.

[133] V. Niculescu. Parallel Programs Development. *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*, Las Vegas, Nevada, USA, June 25-28, 2001, CSREA Press, pp. 94-100.

[134] V. Niculescu, Parallel Algorithms for Fast Fourier Transformation using PowerList, ParList and PList Theories, *Proceedings of International Conference EuroPar'2002*, Paderborn, Germany, August 2002, Springer-Verlag, pp. 400-404

[135] V. Niculescu. A Model for Construction of Parallel Programs, *Proceedings of International Symposium SYNASC'02*, Timisoara, Romania, Oct. 9-12 , 2002, pp.215-232

[136] V. Niculescu, On Data Distribution in the Construction of Parallel Programs, *The Journal of Supercomputing*, 29(1): 5-25, July 2004, Kluwer Academic Publishers, 2004.

[137] V. Niculescu, Unbounded and Bounded Parallelism in BMF. Case Study: Rank Sorting, *Studia Universitatis "Babes-Bolyai"*, Informatica, Vol XLIX, No. 1, 2004, pp. 91-98.

[138] V. Niculescu, Formal Derivation Based on Set-Distribution of a Parallel Program for Hermite Interpolation, *Proceedings of International Symposium SYNASC'04*, Timisoara, Romania, Sept. 26 -30 , 2004, pp.250-258

[139] T. Nodera, N. Tsuno. *The Parallelisation of the Incomplete LU Factorisation on AP1000*. Proceedings of EuroPar'98, pp. 788-792, 1998.

[140] L.S. Nyland, J.F. Prins, A. Goldberg, P.H. Mills. *A Design Methodology for Data-Parallel Applications*. IEEE Transactions on Software Engineering, Vol. 26, No. 4, April 2000.

[141] K. Ogata, H. Hirata, S. Ioroi, K. Futatsugi. *Experimental Implementation of Parallel TRAM on Massively Parallel Computer*. Proceedings of EuroPar'98, pg.847-851, 1998.

[142] E. Pap, P. Szlavi, L. Zsako. *Joining Programming Theorems - a practical Approach to Program Building*. Proceeding of Forth Symposium on Programming Languages and Software Tools, Visegrad, Hungary, June 9-10, 1995.

[143] L.M. Pereira, R. Nasr. Delta-Prolog: A distributed logic programming language. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp. 283-291, Tokyo, 1984.

[144] D. Petcu,*Calcul Paralel*. Universitatea de Vest, Timisoara, 1993

[145] S.L. Peyton-Jones, C. Lester. *Implementing Functional Programming Languages.*, Prentice-Hall International Series in Computer Science, 1992.

[146] M. Popa. *Introducere în arhitecturi paralele şi neconvenţionale*. Editura Computer-Press, Timişoara, 1992

[147] D. Pritchard. Mathematical Models of Distributed Computation. In *Proceedings of OUG-7, Parallel Programming on Transputer Based Machines*, IOS Press, pp. 25-36, 1988.

[148] T. Rauber, Gudula Runger. *A Transformation Approach to Derive Efficient Parallel Implementations*. IEEE Transactions on Software Engineering, Vol. 26, No. 4, April 2000.

[149] M.A. Reniers. *Message Sequence Chart - Syntax and Semantics*. Techniche Univ. Eindhoven, Proefschrift, 1999.

[150] V.A. Saraswat, M. Rinard, P. Panangaden. Semantic foundations of concurrent constraint programming. In *Proceedings of the POPL'91 Conference*, pp. 333-352, ACM Press, 1991.

[151] J.T. Schwartz. *Ultracomputers*. ACM Transactions on Programming, Languages and Systems, 2(4), pp. 484-521.

[152] E. Shapiro. Concurrent Prolog: A progress report. *IEEE Computer*, 19:44-58, 1986.

[153] C. L. Seitz. The Cosmic Cube. *Communication of The ACM* 28(1):22-23, JaJanuary 1985.

[154] F. Seutter. CEPROL, a cellular programming language. *Parallel Computing*, 2:327-333, 1985.

[155] D.B. Skillicorn. *Foundations of Parallel Programming*. Cambridge University Press, 1994.

[156] D.B. Skillicorn, D. Talia. *Models and Languages for Parallel Computation*. ACM Computer Surveys, Vol. 30, No. 2 June 1998, pg.123-136

[157] J.R. Smith. *The Design and Analysis of Parallel Algorithms*. Oxford University Press, 1993.

[158] B. Smyth, G.D. Plotkin. *The Category Theoretic Solution of Recursive Equations*. SIAM Journal of Computing, 11(4):761-783, 1982.

[159] G. Spezzano, D. Talia. CARPET: A programming language for parallel cellular processing. In *Proceedings European School on Paralle Programming Environments 96*, Alpe d'Huez, France, April 1996.

[160] H.S. Stone. Parallel Processing with the perfect shuffle. *IEEE Transactions on Computers*, C-20(2):153-161, 1971.

[161] P. Stpinczynski. *Parallel Algoritms for Solving Linear Reccurence Systems*. Proceedins of CONPAR'92, pp. 343-348, 1992.

[162] D. Talia. *Parallel Computation Still Not Ready for the Mainstream*. Communications of ACM, July 1997, Vol. 40, No. 7, pp. 98-99.

[163] S. Thompson. *Haskell - The Craft of Functional Programming*. Addison-Wesley, 2001.

[164] P.W. Trinder, K. Hammond, H. W. Loidl, J. Peyton. *Algorithm + Strategy = Parallelism*. Journal of Functional Programming 1, January 1993.

[165] K. Ueda. Guarded Horn clauses. Technical report TR-103, ICOT, Tokyo, 1985.

[166] V. Valiant. A bridging model for parallel computation. *Communications of ACM*, 33(8):103-111, August 1990.

[167] E.F. Van de Velde. *Concurrent Scientific Computing*. Spring-Verlag, New-York Inc. 1994.

[168] U. Viskin. *Implementation of Simultaneous Memory Address Access in Models that Forbid it*. Journal of Algorithms, 4(1):45-50, 1983.

[169] B. Wagar. Hyperquicksort. In *Hypercube Multiprocessor 1987, Greece*, pp.292-299. SIAM, 1987.

[170] J. Wasniewski, B.S. Andersen, F. Gustavson. *Recursive Formulation of Cholesky Algoritm in Fortran 90*. Proceedings of CONPAR'98, pp. 574-578, 1992.

[171] C. Wedler, C. Lengauer. *Parallel Implementations of Combinations of Broadcast, Reduction and Scan*. PDSE'97, 1997.

[172] H.S. Wilf. *Algoritmes et complexite*. Mason & Prentice Hall, 1985.

[173] Wilkinson, B., Allen, M., *Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice Hall, 2002.

[174] G. Wilson *Parallel Programming for Scientists and Engineers*. MIT Press, Cambdrige, MA, 1995.

[175] J.C. Wyllie. *The Complexity of Parallel Computations*. Technical Report TR-79-387, Department of Computer-Science, Cornell University, Ithaca, NY, August 1979.

[176] Zhenjiang Hu, Hideya Iwasaki, Masato Takechi. *Formal Derivation of Efficient Parallel Programs by Construction of List Homomorfisms*. ACM Transaction on Programming Language and Systems, Vol. 19, No.3, May 1997, pp. 444-461.

[177] Z. Zlatev. *Computational Methods for General Sparse Matrices*. Kluwer Academic Publishers, 1991.

[178] J. Zwiers. *Compositionaly, Concurency and Partial Correctness: Proof Theories for Networks of Processes and their Connection*. Eindhoven, Proefschrift, 1988.

[179] The Message Passing Interface (MPI) standard, [http://www-unix.mcs.anl.gov/mpi/], 2000.

[180] PVM: Parallel Virtual Machine . A Users Guide and Tutorial for Networked Parallel Computing, MIT Press, 1994.

Glosar