ALGORITMI DE CALCUL PARALEL

BOGDAN DUMITRESCU

septembrie 2001

Cuprins

Capitolul 1

Arhitecturi paralele

Deşi aeeastă carte este dedicată algoritmilor paraleli, primul capitol se ocupă de arhitecturi. Motivul este foarte simplu. Aşa cum între programatorii care rezolvă o problemă utilizând un acelaşi calculator secvenţial şi un acelaşi algoritm apar mari diferenţe în eficacitatea programelor scrise—datorită modului în care fiecare ştie să folosească particularităţi ale hardware-ului, ale sistemului de operare sau ale limbajului de programare folosit—departajarea este şi mai netă în cazul utilizării calculatoarelor paralele; cauza principală este în primul rând neadecvarea algoritmilor la arhitectură. Dacă a face un program să "meargă" pe o arhitectură paralelă nu este, până la urmă, o dificultate prea mare, a-l face eficient este cu siguranţă mult mai greu decât în cazul secvenţial. De aceea, un prim pas în abordarea paralelismului, chiar pentru programatorul obişnuit (cel care rezolvă probleme generale) este trecerea în revistă a tipurilor de arhitecturi paralele. Nu vom intra în amănunte, ci ne vom mulţumi cu o descriere generală, suficientă însă pentru a releva implicaţiile asupra algoritmilor, deci asupra programării; pentru o informare mai completă vezi, de exemplu, cartea scrisă de Hockney şi Jesshope [15], apărută şi în limba română; de asemenea, Internetul conţine vaste resurse pentru aflarea celor mai noi tendinţe în domeniu; un bun punct de start este de exemplu `http://www.top500.org`.

Tipuri de paralelism. La ora actuală, practic orice calculator, inclusiv micul PC pe care-l aveţi aproape zilnic în faţă, conţine elemente de paralelism. Ele sunt încă suficient de bine ascunse pentru a nu obliga utilizatorul să-şi schimbe semnificativ felul de a gândi. Apariţia lor a fost determinată de necesitatea creşterii vitezei de calcul, prin alte mijloace decât creşterea frecvenţei ceasului (permisă de permanente inovaţii tehnologice în conceperea şi producerea circuitelor electronice). Cel mai evident exemplu este cel al coprocesorului pentru PC-uri; este, în fond, un alt procesor specializat într-un singur gen de calcule, cele în virgulă mobilă; el lucrează în paralel cu procesorul, dar la comanda acestuia; dacă acum 10 ani coprocesorul era un circuit separat, acum el este parte a procesorului. Paralelismul acesta se numeşte *funcţional*; există unităţi de calcul pentru diverse tipuri de operaţii—unele pentru întregi, altele

pentru numere reale—care lucrează în paralel. Astfel, procesorul poate executa mai multe instrucţiuni, sau faze diferite (decodificare, citire/scriere operanzi, calcul efectiv) ale mai multor instrucţiuni, în acelaşi timp. Totuşi, din punctul de vedere al programării, un calculator secvenţial are prin excelenţă un singur procesor.

La extrema cealaltă se află calculatoarele care au zeci, sute sau chiar mii de procesoare identice, care pot executa fiecare un flux propriu de instrucţiuni, cooperând la rezolvarea unei probleme. Astfel de calculatoare se numesc *masiv paralele*. Deoarece procesoarele sunt ele însele extrem de puternice, un calculator masiv paralel este de sute/mii de ori mai rapid decât calculatoarele secvenţiale din aceeaşi generaţie. Acestor calculatoare le sunt destinaţi cei mai mulţi dintre algoritmii prezentaţi în această carte.

Paralel şi concurent. Nu este poate prea devreme pentru a face deosebirea între noţiunile de *paralel* şi *concurent*. Presupunând că există mai multe procese (taskuri, părţi de program, sau chiar programe) care trebuie să se execute, modul concurent implică numai o eventuală senzaţie de paralelism, pentru că există o singură unitate centrală, la ocuparea căreia concurează procesele; dacă mecanismul după care se permite alocarea unităţii centrale este bun, procesele pot părea că se execută în paralel datorită vitezei mari de execuţie. Este situaţia întâlnită în cazul sistemelor de operare multitasking. O execuţie cu adevărat paralelă a proceselor se face folosind mai multe procesoare, în cazul ideal câte unul pentru fiecare proces. În multitasking, problemele dificile apar în a face să coopereze mai multe procese ce utilizează aceeaşi resursă; în paralelism, problema importantă e de a face un program, format din mai multe procese care cooperează, să se execute pe mai multe procesoare. Aceste probleme au, totuşi, destule puncte comune; de aceea, nu în puţine cazuri, modurile de descriere paralel, respectiv concurent, sunt asemănătoare; diferenţa esenţială, repetăm, apare la execuţie.

1.1 O clasificare generală

Una dintre marile probleme ale algoritmicii paralele este că, spre deosebire de cea secvenţială, unde modelul de calculator—von Neumann—era unic şi foarte simplu conceptual, există o multitudine de clase de arhitecturi care trebuie avute în vedere. Secvenţial se gândeşte într-un singur fel (sau aproape), paralel în foarte multe. Un algoritm excelent pe un anume tip de calculator poate fi ineficient pe un altul. De aceea nu se poate vorbi despre varianta paralelă a unui algoritm secvenţial, ci despre adaptări paralele pe diverse arhitecturi.

Principalele categorii de arhitecturi sunt departajate—într-o clasificare datorată lui Flynn, probabil cea mai populară datorită simplităţii ei—prin numai două criterii: al fluxului de instrucţiuni şi al fluxului de date; ambele pot fi sau unice, sau multiple. Conform acestei clasificări calculatoarele pot fi:

- SISD (Single Instruction Single Data): la un moment dat se execută o singură instrucţiune asupra unei singure date; este calculatorul secvenţial obişnuit.

- SIMD (Single Instruction Multiple Data): la un moment dat se execută o singură instrucţiune, dar asupra mai multor date; este un calculator paralel, în care procesoarele execută sincron acelaşi program, dar fiecare asupra altui set de date.
- MIMD (Multiple Instruction Multiple Data): fiecare procesor execută un program propriu, folosind date proprii.
- MISD (Multiple Instruction Single Data): o categorie deocamdată vidă, în care aceleaşi date ar fi prelucrate simultan în mod diferit, de către mai multe procesoare.

Se reţin deci principalele două categorii de calculatoare paralele—SIMD şi MIMD—de care ne vom ocupa în continuare.

1.1.1 Arhitecturi SIMD

Arhitecturi pipeline. Din punct de vedere istoric, primul tip de paralelism a fost cel *pipeline*, în care prelucrarea datelor se face ca pe o bandă rulantă, pe care există mai multe posturi de lucru; la fiecare dintre acestea se execută o operaţie asupra produsului; la un moment dat există pe bandă atâtea produse câte posturi, în diverse faze de execuţie. Aşadar, modul de prelucrare pipeline este un caz particular de paralelism funcţional. Ideea de bază este de a descompune operaţiile în cele mai mici părţi componente; de exemplu, pentru a aduna două numere reprezentate în format virgulă mobilă, se parcurg secvenţial următoarele etape:

E1: compararea exponenţilor;

E2: aducerea la acelaşi exponent cu numărul mai mare a celui mai mic, a cărui mantisă se modifică corespunzător;

E3: adunarea mantiselor;

E4: normalizarea rezultatului.

Ilustrăm prelucrarea datelor în figura 1.1. O pereche de date D_k trebuie să parcurgă etapele secvenţial, dar pot exista patru perechi prelucrate la un moment dat; practic se execută aceeaşi operaţie asupra mai multor date, dar prin suboperaţii diferite. Aceasta permite execuţia unei adunări—sau a unei operaţii, în general—la fiecare perioadă de ceas, cu condiţia asigurării unui flux de date suficient; desigur, este necesar un timp de amorsare, timpul în care prima pereche de date parcurge cele patru "posturi de lucru".

Pe acest principiu s-au bazat primele calculatoare *vectoriale*, a căror arhitectură este reprezentată extrem de schematic în figura 1.2. Denumirea provine din eficienţa cu care se executau, pe aceste calculatoare, operaţii vectoriale, ca de exemplu $y = \alpha x + y$, cu x, y vectori n-dimensionali şi α un scalar.

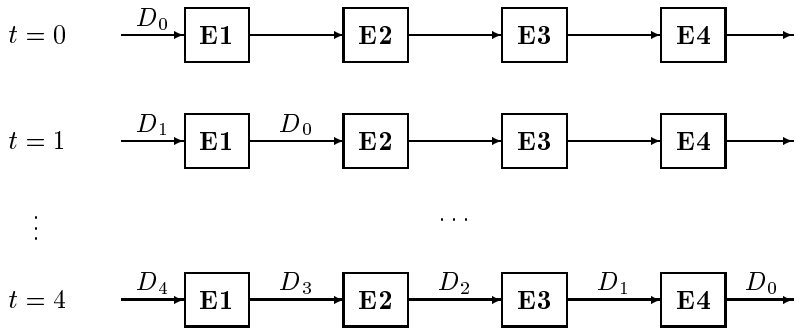


Figura 1.1: Evoluţia datelor într-o arhitectură pipeline. Notăm generic D_k datele în toate fazele prelucrării.

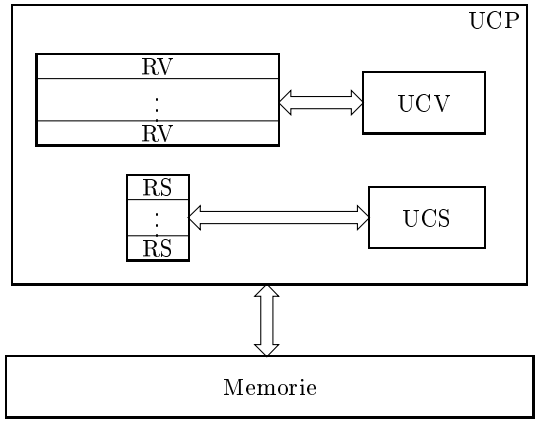


Figura 1.2: Structura unui calculator vectorial monoprocesor. UCP - unitate centrală de prelucrare; RV - registru vectorial; UCV - unitate de calcul vectorial; RS - registru scalar; UCS - unitate de calcul scalar.

Unitatea centrală conţine unităţi de calcul şi registre, atât scalare cât şi vectoriale. Prin stocarea operanzilor de tip vector în registrele vectoriale, unitatea de calcul vectorial poate fi alimentată foarte rapid, asigurându-i-se funcţionare în regim pipeline. Desigur că de foarte mare importanţă este şi modul de conectare a unităţii centrale cu memoria, astfel încât să se poată asigura traficul dintre aceasta şi registrele vectoriale. Există şi unităţi arhitecturale pentru operaţii scalare, care pot funcţiona (uneori) în paralel cu cele vectoriale.

Consecinţa acestei organizări arhitecturale este rescrierea algoritmilor în aşa fel încât ei să conţină cât mai multe operaţii vectoriale cu vectori de dimensiune cât mai apropiată de cea a registrelor; astfel, timpul de execuţie este mai mic decât pentru aceiaşi algoritmi în variantă scalară.

La această oră toate procesoarele—inclusiv cele dintr-un simplu PC—conţin elemente de paralelism pipeline; însă este vorba de obicei despre paralelism între decodificarea unei instrucţiuni şi execuţia celei anterioare, sau/şi între acestea şi accesul la memorie. Se poate spune că ideea de pipeline a migrat de la nivelul calculatorului, la nivelul procesorului.

SIMD pur paralel. O altă variantă de arhitectură SIMD este cea prezentată în figura 1.3, aici fiind vorba cu adevărat de paralelism. Procesoarele P_i execută simultan aceeaşi instrucţiune, dictată de unitatea de comandă—cea care deţine programul. Operaţiile se execută asupra unor date aflate în memoriile locale M_i; acestea sunt de capacitate mică, dar rapide, şi sunt alimentate cu date de la o memorie externă. Uneori există o reţea de interconectare între procesoare şi memorii (figurată printr-un dreptunghi cu linie întreruptă), astfel încât, la un moment dat, un procesor poate accesa alte memorii, dar nu pot exista două memorii accesate simultan de acelaşi procesor; de exemplu, fiecare procesor accesează memoria vecinului său din dreapta. Acest tip de arhitectură—SIMD veritabil, spre deosebire de cazul pipeline—este adecvat tot calculelor vectoriale; de aceea, astfel de calculatoare sunt numite şi ele vectoriale.

De exemplu, adunarea a doi vectori de lungime egală cu numărul de procesoare şi cu elementele stocate câte două în fiecare memorie locală, în locaţii aflate la aceeaşi adresă relativă la începutul memoriei respective, durează tot atât cât adunarea a două numere reale, adică, în general, un singur tact.

Arhitecturile SIMD de tipul celor din figura 1.3 au fost foarte populare spre sfârşitul anilor '80; exemple de astfel de arhitecturi sunt calculatoarele Connection Machine 2, DAP, MasPar MP-1, Zephir. După 1990, ele au devenit din ce în ce mai puţin competitive, arhitecturile MIMD dovedind posibilităţi superioare de progres.

1.1.2 Arhitecturi MIMD

Calculatoarele MIMD se împart în două mari categorii, după modul de repartizare a memoriei: cu memorie *partajată (comună)* şi cu memorie *distribuită*. În ambele cazuri vom presupune că există p procesoare, notate P_i, $i = 0 \ldots p-1$; vom numi i *adresa* (sau identitatea) procesorului; vom utiliza în egală măsură reprezentările zecimală

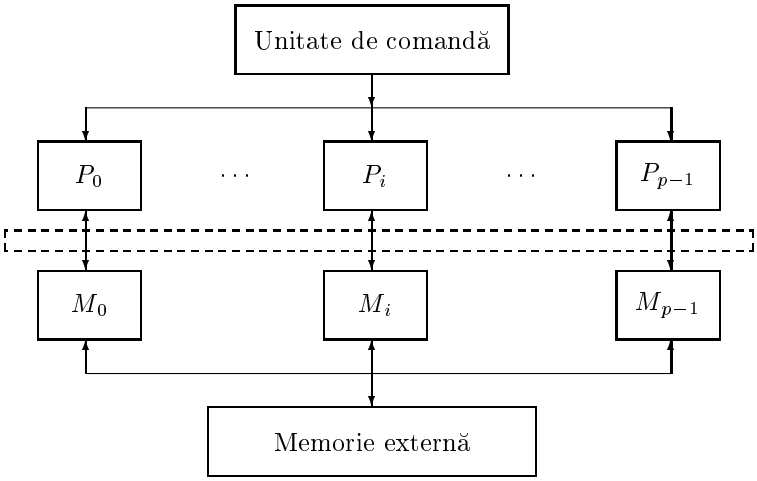


Figura 1.3: Structura unui calculator SIMD cu reţea de procesoare.

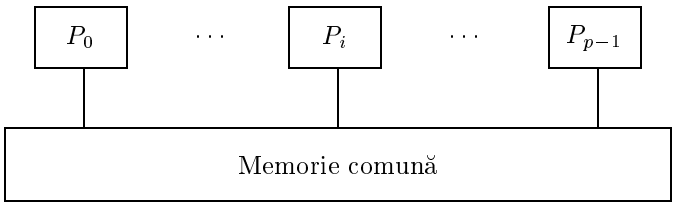


Figura 1.4: Arhitectura de principiu a unui calculator MIMD cu memorie comună.

şi binară ale numărului i. Fiecare procesor poate executa programul său propriu, independent de cel al celorlalte procesoare; aceasta nu înseamnă că procesoarele au întotdeauna programe diferite; diferenţa esenţială faţă de arhitecturile SIMD este că execuţia programelor nu se face sincron, ci independent pe fiecare procesor.

MIMD cu memorie partajată. La arhitectura cu memorie partajată, cum este cea prezentată în figura 1.4, există o memorie comună de dimensiuni mari şi, eventual, memorii locale (nereprezentate în figură) de dimensiuni modeste, dar mai rapide; în figura 1.5 este reprezentată o configuraţie cu un singur procesor şi cu o astfel de memorie, numită de obicei *cache*; în cazul multiprocesor, accesul la memoria comună (globală) este coordonat de o unitate logică specială, care are rolul de a arbitra conflictele apărute în urma mai multor cereri simultane de acces la memorie.

Principala problemă a acestor arhitecturi este accesul la memoria comună; posibilele conflicte introduc întârzieri, ceea ce nu permite o bună eficienţă atunci când

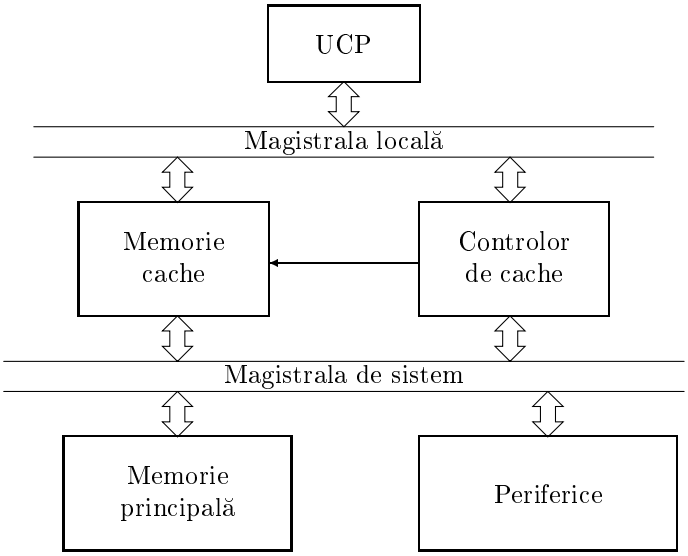


Figura 1.5: Structură monoprocesor cu memorie cache.

numărul de procesoare este mare (peste câteva zeci), adică o dată cu creşterea probabilităţii apariţiei conflictelor. Tehnologia actuală permite realizarea unor calculatoare cu memorie comună performante cu cel mult 32 de procesoare.

Se ştie că viteza procesoarelor este mai mare, uneori semnificativ, decât cea a memoriilor. De aceea este naturală utilizarea de către un procesor a mai multor blocuri (pagini) de memorie, la care să facă pe rând cereri de citire/scriere, la fiecare tact câte una; de exemplu, dacă un bloc are nevoie de patru tacte pentru a satisface cererea, atunci patru blocuri vor fi, în general, suficiente pentru a nu se obţine timpi morţi în funcţionarea procesorului. Ori în arhitecturile MIMD cu memorie partajată pare a fi exact invers. În fapt nu e chiar aşa; imensa memorie comună este împărţită într-un număr relativ mare de blocuri accesibile independent; există acea unitate funcţională a calculatorului care gestionează cererile de acces la memorie ale procesoarelor şi care permite simultaneitatea, dar numai în cazul accesului la blocuri diferite; practic, toate blocurile de memorie pot lucra în paralel, reuşindu-se alimentarea procesoarelor; de exemplu, dacă un acces la memorie durează patru tacte şi sunt 16 procesoare, e nevoie de minimum 64 de blocuri de memorie. În plus, memoriile cache permit micşorarea traficului cu memoria comună, prin reutilizarea unor date depuse acolo, deci deja locale.

Aşadar, accesul la memoria comună nu este secvenţial. Implicaţia algoritmică este organizarea accesului la memorie astfel încât să se evite conflictele, să se acceseze deci simultan locaţii de memorie aflate în blocuri diferite. Esenţial în caracterizarea

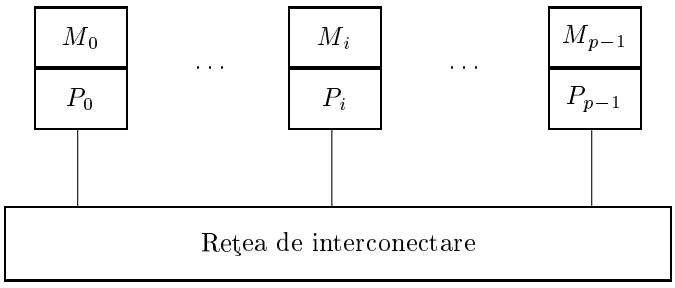


Figura 1.6: Arhitectura de principiu a unui calculator MIMD cu memorie distribuită.

arhitecturii rămâne faptul că fiecare procesor are acces la întreaga memorie. Printre calculatoarele actuale cu memorie comună se numără Sun 10000 Enterprise.

MIMD cu memorie distribuită. În arhitectura cu memorie distribuită, aşa cum este prezentată în figura 1.6, fiecare procesor P_i are o memorie locală M_i; nu există memorie comună. Cooperarea procesoarelor se realizează prin intermediul legăturile asigurate de o reţea de interconectare, cu topologie fixată sau programabilă; de altfel, configuraţia acestei reţele este una dintre cele mai importante trăsături ale unui astfel de calculator MIMD. Fiecare procesor P_i execută un program propriu aflat în M_i şi prelucrează date aflate tot în M_i. Vom dezvolta prezentarea acestui tip arhitectural în secţiunea următoare, deoarece el pare a avea cel mai bun viitor, limitarea numărului de procesoare venind deocamdată mai mult din motive economice.

Arhitecturi mixte. Deoarece arhitecturile cu memorie comună sunt foarte eficiente pentru un număr mic de procesoare, o tendinţă actuală este de a realiza calculatoare în care clustere de 4-8 procesoare cu memorie comună sunt conectate printr-o reţea de comunicaţie. Putem spune astfel că, într-un calculator cu memorie distribuită, procesoarele sunt înlocuite cu astfel de clustere. Se îmbină astfel calităţile celor două tipuri de calculatoare MIMD. Este interesant că modul de programare a acestor arhitecturi mixte este tipic fie memoriei comune, fie celei distribuite, fie amândurora (dar nu simultan); aşadar, nu s-a creat şi un mod de programare specific arhitecturilor mixte. Exemple de astfel de arhitecturi sunt calculatoarele HP 9000 Superdome, SGI Origin 3000.

Accesul la un calculator paralel. O problemă destul de delicată este modul de realizare a legăturilor între calculatoarele MIMD şi exterior. De obicei aceasta se face prin intermediul unui calculator gazdă, pe care utilizatorii lucrează în regim multitasking; pe acest calculator se realizează pregătirea programului, adică editarea şi compilarea. Cum acest tip de legătură este lent, transmiţându-se printr-un singur canal informaţional programe şi date pentru mai multe procesoare, de către un singur utilizator la un moment dat, tendinţa actuală este de a permite conectarea utilizatorilor direct la procesoare ale calculatorului paralel, pe care se execută nucleul

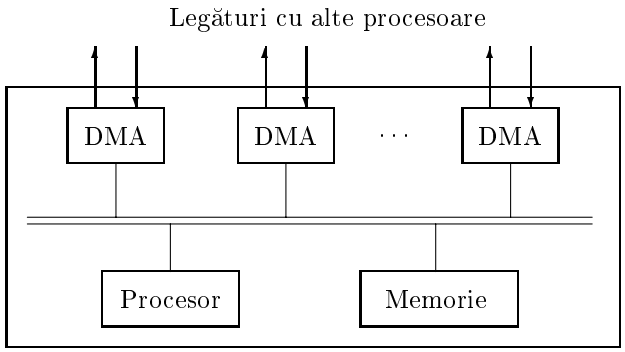


Figura 1.7: Structura simplificată a unui nod într-o arhitectură MIMD cu memorie distribuită.

unui sistem de operare, căruia îi revine sarcina de a ocupa mai multe procesoare la momentul execuţiei.

Probleme

P 1.1.1 Presupunem că o unitate pipeline realizează o operaţie în trei etape, pentru fiecare existând câte un "post de lucru". Timpul de calcul în fiecare dintre cele trei etape A, B şi C este diferit: 1, 2, respectiv 1 (tacte). La câte tacte produce unitatea pipeline un rezultat ? Cum se poate face astfel încât la fiecare tact să se producă un rezultat ? În cât timp sunt prelucrate 100 de date ?

1.2 Arhitecturi MIMD cu memorie distribuită

1.2.1 Caracteristici ale nodurilor

Revenind la figura 1.6, o pereche procesor-memorie (P_i, M_i) este denumită *nod*. Fiecare nod are un număr de canale de comunicaţie (porturi) prin care schimbă informaţii cu alte noduri. Canalele sunt fie prevăzute on-chip—ca la transputerul T800 sau la procesorul actual Compaq Alpha EV67/68, care au câte patru canale—fie asigurate de procesoare (circuite) specializate pentru comunicaţie. În ambele cazuri, comunicaţia poate decurge în paralel cu alte operaţii (calcule) efectuate de către procesor; structura unui nod poate fi simplificată la modelul prezentat în figura 1.7, în care canalele de comunicaţie sunt realizate prin interfeţe DMA (Direct Memory Acces); pentru a comunica pe un canal, procesorul indică interfeţei DMA adresa din memoria locală la care se găseşte sau la care trebuie depus mesajul, precum şi lungimea acestuia; apoi, DMA-ul transmite mesajul pe legătura fizică, accesând memoria concurent cu procesorul şi, eventual, alte interfeţe DMA.

Procesoarele utilizate în calculatoarele paralele cu memorie distribuită actuale sunt de obicei de tip RISC (Reduced Instruction Set Computer); exemple actuale sunt Hewlett Packard PA/RISC 8600, MIPS R14000, Sun UltraSPARC III; setul de instrucţiuni al acestor procesoare este redus, obiectivele avute în vedere fiind:

– posibilitatea de a executa o instrucţiune pe ciclu maşină;

– reducerea suprafeţei de circuit ocupate de partea de decodificare a instrucţiunilor, eliberând loc pentru memoria cache (o parte din M_i fiind deci pe aceeaşi pastilă cu P_i) sau pentru partea de comunicaţie.

S-a ajuns aici deoarece s-a constatat că procesoarele cu set mare de instrucţiuni (CISC – Complex Instruction Set Computer) au două dezavantaje: compilatoarele nu folosesc la întreaga valoare acest set de instrucţiuni şi, în plus, partea de decodificare a instrucţiunilor ocupă foarte mult din suprafaţa circuitului. Simplificând tipurile instrucţiunilor şi modurile de adresare, se câştigă în viteza de execuţie. Suprafaţa eliberată astfel poate fi folosită şi în scopul introducerii paralelismului în interiorul procesorului. Execuţia instrucţiunilor se face, aşa cum am menţionat, în mod pipeline, cu următoarele faze: citirea instrucţiunii, decodarea ei şi citirea operanzilor, execuţia propriu-zisă de către unitatea aritmetică şi logică, depunerea rezultatului. Aceasta funcţionare decurge la viteză maximă atâta vreme cât nu există conflicte sau dependenţe de date între instrucţiuni succesive, instrucţiuni de salt sau întreruperi.

O altă tendinţă este aceea de încerca execuţia mai multor instrucţiuni pe tact (procesoare VLIW – Very Long Instruction Word); pentru aceasta e nevoie de mai multe unităţi funcţionale pentru diverse tipuri de calcule (întregi, în virgulă mobilă, sau chiar mai specializat: adunare, înmulţire), fiecare putând funcţiona în mod pipeline; procesoarele menţionate mai sus au câte două unităţi de calcul în virgulă mobilă. De asemenea, magistrala de comunicaţie cu memoria are dimensiuni mari, de exemplu 64 de biţi, un cuvânt putând conţine mai multe instrucţiuni, care se vor executa în paralel.

1.2.2 Câteva noţiuni din teoria grafurilor

Putem asocia imediat o arhitectură cu memorie distribuită unui graf neorientat: procesoarele sunt nodurile iar liniile de comunicaţie între procesoare (considerate bidirecţionale)—arcele. De aceea este necesară reamintirea unor noţiuni din teoria grafurilor şi reliefarea semnificaţiilor lor specifice acestui context. (De asemenea, cum se va vedea mai târziu, şi unui algoritm îi putem asocia un graf, important pentru detectarea paralelismului.)

Un *graf* neorientat $G = (V, E)$ este constituit dintr-o mulţime V, a nodurilor, şi o alta, $E \subset V \times V$, conţinând perechi de noduri, (în care nu distingem ordinea, deci $(x, y) = (y, x)$), adică arcele care unesc aceste noduri[1].

Numărul de noduri ale unui graf este numit *ordin*; îl vom nota cu p; în cazul arhitecturilor paralele, p este numărul de procesoare.

[1] Notaţia (V, E) este oarecum tradiţională; în engleză, *vertex* înseamnă nod, iar *edge* arc.

Un arc de forma (v, v) se numeşte *buclă*. Un graf fără bucle se numeşte *simplu*. Grafurile care ne interesează vor fi toate simple, o cale de comunicaţie de la un procesor la el însuşi neavând sens.

Două noduri sunt *adiacente* (sau vecine), dacă există un arc între ele. Numărul de vecini ai unui nod se numeşte *gradul* nodului respectiv; este deci numărul de arce plecând (sau venind, sensul nu contează) dintr-un nod. Altfel zis, gradul este numărul de canale de comunicaţie ale unui procesor.

Gradul unui graf (Δ) este maximul gradelor nodurilor componente. Graful este *regulat* dacă gradele tuturor nodurilor sunt egale. Observăm că numărul de arce al unui graf regulat este $p\Delta/2$.

Se numeşte *drum* (sau *cale*) între două noduri $x, y \in V$ o secvenţă de noduri x_0, x_1, ..., x_k, astfel încât două noduri consecutive sunt adiacente şi $x_0 = x$, $x_k = y$. Drumul se numeşte *elementar* dacă nodurile componente $x_1, \ldots, x_{k-1}$ sunt distincte. Într-o arhitectură cu memorie distribuită, un drum este o cale de comunicaţie între două procesoare; vor fi utilizate doar drumuri elementare, după cum este logic.

Lungimea unui drum este numărul de arce ale sale, k în notaţia de mai sus. *Distanţa* dintre două noduri, $dist(x, y)$, este lungimea celui mai scurt drum între acestea; în contextul memorie distribuită, vom nota $dist(P_i, P_j)$ sau chiar $dist(i, j)$ distanţa între procesoarele cu adresele i şi j.

Diametrul grafului (D) este maximul tuturor distanţelor între perechi de noduri ale grafului, adică distanţa dintre cele mai depărtate noduri. *Excentricitatea* unui nod este cea mai mare distanţă dintre acesta şi un alt nod; pentru orice nod, excentricitatea este mai mică sau egală cu diametrul (o putem privi ca diametrul din punct de vedere al nodului respectiv).

Vom numi *ciclu* un drum elementar pentru care nodul iniţial şi cel final sunt identice. Un *ciclu hamiltonian* este un ciclu trecând (exact o dată) prin fiecare nod al grafului; lungimea sa este egală cu ordinul grafului. Un graf posedând un ciclu hamiltonian se numeşte *graf hamiltonian*.

Un graf se numeşte *conex* dacă există (cel puţin) un drum între oricare două noduri ale sale.

Un *arbore* este un graf conex fără cicluri.

1.2.3 Topologii curente

Calităţile topologiei reţelei de interconectare. Din punctul de vedere al algoritmilor pentru arhitecturi cu memorie distribuită, este de mare importanţă topologia reţelei de interconectare, deoarece aceasta restrânge posibilităţile de comunicare. Să vedem întâi care ar trebui să fie calităţile acestei topologii.

Evident, graful trebuie să fie conex. Apoi, pe de o parte, e de dorit ca gradul grafului să fie cât mai mic şi, dacă se poate, graful să fie regulat; deci, un procesor să nu aibă prea multe canale de comunicaţie, pentru a fi uşor de realizat (un număr mare de canale înseamnă complexitate mare a procesorului şi număr ridicat de pini), şi toate procesoarele—identice, în mod obişnuit—să-şi folosească integral canalele;

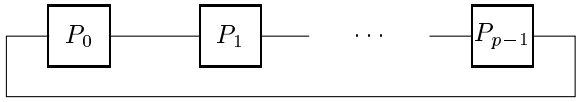


Figura 1.8: Calculator MIMD cu topologie de inel.

strâns legată de aceasta este cerinţa ca numărul total de arce să fie relativ mic, fiecare arc însemnând o conexiune fizică realizată printr-un traseu electric pe o placă şi, eventual, între plăci. Pe de altă parte, e cu atât mai bine cu cât diametrul este mai mic, astfel încât distanţele între procesoare să fie limitate, iar comunicarea cât mai facilă.

Este clar că aceste două deziderate sunt antagonice; cu cât gradul e mai mic, cu atât sunt mai puţine arce, deci scad şansele să existe drumuri scurte între oricare două procesoare. Micşorarea gradului şi a numărului de arce este o cerinţă hardware, cea a diametrului provine din software.

La o extremă se află graful *total interconectat* (sau *complet*), în care există arce între oricare două noduri; diametrul său este 1, gradul $O(p)$ (mai exact[2] $p-1$) şi numărul de arce $O(p^2)$ (exact $p(p-1)/2$); dacă diametrul este perfect, gradul şi numărul de arce sunt imense; o astfel de topologie este realizabilă practic doar pentru un număr infim de procesoare. La cealaltă extremă se află inelul, despre care vom vorbi imediat.

În continuare vom enumera câteva tipuri curente de configuraţii, împreună cu unele proprietăţi ale lor. Este vorba despre grafuri cu foarte bune proprietăţi de simetrie, pentru care descrierea algoritmilor se face într-un mod mult mai simplu decât în cazul unor asimetrii pronunţate.

Inelul. Într-un inel (figura 1.8), procesorul P_i este legat de P_j, unde $j = (i-1) \bmod p$ sau $j = (i+1) \bmod p$; în primul caz vom spune că P_j este vecinul din stânga, în al doilea, cel din dreapta. Gradul inelului este $\Delta = 2$, numărul de arce p (ambele valori fiind bune), dar diametrul este $D = \lfloor p/2 \rfloor$, adică mare; de aceea, în astfel de arhitecturi se utilizează puţine procesoare; de altfel, nici unul dintre calculatoarele performante actuale nu are această topologie (deşi ea a fost utilizată la începutul anilor '80 în diverse modele experimentale) Din punct de vedere al notaţiei procesoarelor, pentru simplificare, vom considera că notaţia P_i este corectă, indiferent de valoarea întregului i, deci $P_i \equiv P_{i \bmod p}$; de exemplu, P_{-1} şi P_{p-1} indică acelaşi procesor.

Grila de dimensiune 2 (figura 1.9a) are $p = m \cdot n$ procesoare. Pentru simplitate, convenim să asociem procesorului o adresă cu două coordonate, e.g. (i, j); o adresă formată dintr-un număr în intervalul $0 : p-1$ se poate obţine banal, de pildă $(i-1) * p+j$ corespunde lui (i, j), dacă numerotăm procesoarele în ordinea liniilor. Procesorul

[2]Se spune despre o funcţie $f(n)$ că are valori de ordinul $O(g(n))$ dacă, pentru toţi n mai mari decât un n_0, există două constante c_1 şi c_2 astfel încât $c_1 g(n) \leq f(n) \leq c_2 g(n)$. Cu alte cuvinte, funcţia $g(n)$, de obicei simplă (monom sau logaritm), poate fi folosită pentru aproximarea ordinului de mărime al funcţiei mai "complicate" $f(n)$.

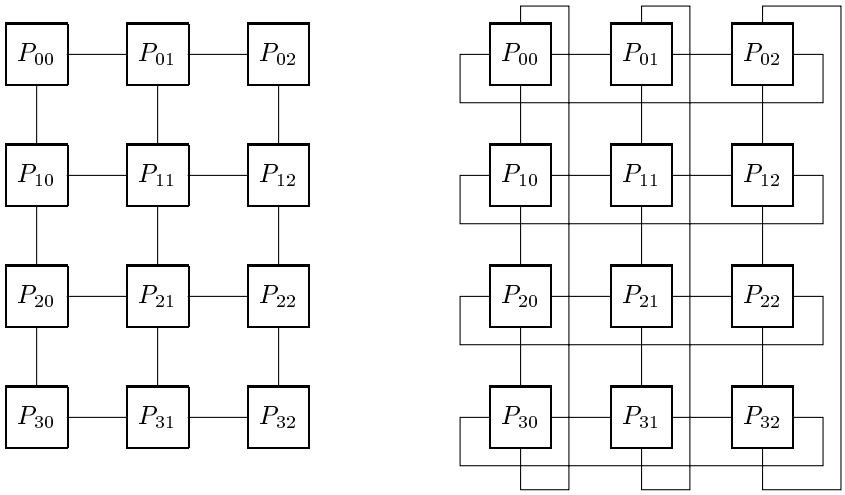


Figura 1.9: Exemple de calculator MIMD cu topologie de (a) grilă 4×3 (b) tor 4×3.

P_{ij} este legat de procesorul P_{kl} (cu $0 \leq i,k \leq m-1$ şi $0 \leq j,l \leq n-1$), una singură din condiţiile următoare fiind îndeplinită: $|k-i| = 1$ sau $|l-j| = 1$. Un procesor are cel mult patru vecini ($\Delta = 4$), numiţi stânga, dreapta, sus, jos sau vest, est, nord, sud; diametrul grafului este $D = m + n - 2$, iar numărul de arce $2mn - m - n$. Un exemplu de mare succes la mijlocul deceniului trecut este calculatorul Intel Paragon.

Torul de dimensiune 2 (figura 1.9b), este o grilă 2D, cu procesoarele de la margini legate între ele astfel încât fiecare linie sau coloană de procesoare să formeze un inel. Torul este regulat, cu $\Delta = 4$, $D = \lfloor m/2 \rfloor + \lfloor n/2 \rfloor$ (vezi diametrul inelului) şi număr de arce $2mn$. Ca şi la grilă, procesoarele vor fi notate P_{ij}; în afară de aceasta, se va folosi artificiul de notaţie de la inel, adică $P_{ij} \equiv P_{i \bmod m, j \bmod n}$. Un exemplu de calculator cu topologie tor 2D este Fujitsu AP3000. Grila şi torul se pot generaliza imediat în 3 dimensiuni. Calculatorul Cray T3E, unul din cele mai puternice actualmente, are o topologie de tor 3D.

Hipercubul (figura 1.10) a fost cea mai populară arhitectură în perioada 1985-1990; marele său succes a venit atât din calităţile intrinseci ale grafului, cât şi din numărul mare de algoritmi ce i se potrivesc (vom vedea mai târziu care e sensul acestei potriviri); calităţile algoritmice sunt cele care îl fac în continuare demn de un interes special. Notăm $\mathcal{H}_d$ hipercubul de dimensiune d, care are $p = 2^d$ procesoare. Fiecare procesor este legat de d vecini; P_i este conectat de P_j dacă reprezentările binare ale

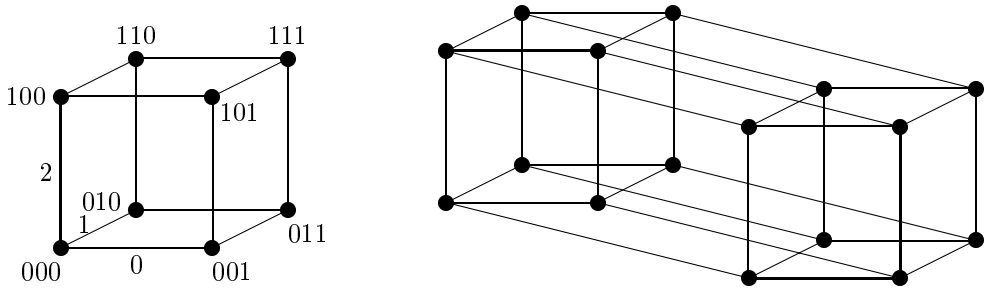


Figura 1.10: Două exemple de hipercub: (a) de dimensiune 3 (b) de dimensiune 4.

numerelor i şi j diferă printr-un singur bit[3]; de exemplu, în hipercubul de dimensiune 3, P_0 este conectat cu P_1, P_2 şi P_4 (vom mai spune, cu acelaşi înţeles, că 000 este conectat cu 001, 010 şi 100 sau că 0 este conectat cu 1,2 şi 4). Deci $\Delta = d = \log p$, numărul de arce $p(\log p)/2$ şi diametrul $D = \log p$ (vezi problema 1.2.3). Un exemplu de calculator cu topologie de hipercub este familia nCUBE.

Hipercubul poate fi definit şi recursiv; $\mathcal{H}_d$ poate fi obţinut luând două copii ale lui $\mathcal{H}_{d-1}$ şi adăugându-le arcele care unesc nodurile aflate în aceeaşi poziţie în cele două copii; la adresele nodurilor se va adăuga un bit în poziţia cea mai semnificativă: 0 pentru prima copie a lui $\mathcal{H}_{d-1}$, 1 pentru a doua.

Vom spune că arcul unind P_i şi P_j se găseşte în dimensiunea l dacă i şi j diferă în bitul din poziţia l; ne vom referi întotdeauna la reprezentări binare: dacă $i = i_{d-1}i_{d-2}\ldots i_l\ldots i_0$, atunci $j = i_{d-1}i_{d-2}\ldots\bar{i}_l\ldots i_0$; se mai spune că P_i şi P_j sunt vecini în dimensiunea l; vezi, de exemplu, figura 1.10, în care sunt precizate dimensiunile în care se află arcele plecând din P_0. Se vede deci că eliminând toate arcele dintr-o anume dimensiune a lui $\mathcal{H}_d$, se vor obţine două hipercuburi $\mathcal{H}_{d-1}$; eliminând apoi arcele dintr-o altă dimensiune, vor rezulta patru hipercuburi $\mathcal{H}_{d-2}$ etc. Notăm $\mathcal{H}_d^r$ familia de 2^{d-r} hipercuburi $\mathcal{H}_r$, obţinute dintr-un hipercub $\mathcal{H}_d$ prin eliminarea arcelor din dimensiunile $d-1, d-2, \ldots, d-r$; altfel zis, nodurile din fiecare din aceste hipercuburi au primii (i.e. cei mai semnificativi) $d-r$ biţi identici. Notăm $\mathcal{H}_d^r(q)$ un astfel de hipercub, anume pe cel care conţine procesorul P_q. În mod analog, notăm $\bar{\mathcal{H}}_d^r$ hipercuburile de dimensiune r, în fiecare nodurile având *ultimii* (i.e. cei mai puţin semnificativi) $d-r$ biţi identici. De exemplu, în $\mathcal{H}_3$, hipercubul $\mathcal{H}_3^2(0)$ are dimensiune 2 şi conţine nodurile 000, 001, 010, 011, iar hipercubul $\bar{\mathcal{H}}_3^2(0)$ conţine nodurile 000, 010, 100, 110.

Simetrie. Pentru inel, tor şi hipercub, simetria poate fi subliniată foarte simplu.

[3]Se mai spune că distanţa Hamming dintre cele două noduri este egală cu 1; distanţa Hamming este numărul poziţiilor în care reprezentările binare diferă: $\sum_{l=0}^{d-1} i_l \oplus j_l$; notăm cu i_l bitul din poziţia l, în reprezentarea binară a lui i.

Topologie	Δ	D	N
Inel	2	$O(p)$	$O(p)$
Grilă	4	$O(\sqrt{p})$	$O(p)$
Tor	4	$O(\sqrt{p})$	$O(p)$
Tor 3D	6	$O(\sqrt[3]{p})$	$O(p)$
Hipercub	$\log p$	$O(\log p)$	$O(p \log p)$

Tabelul 1.1: Gradul Δ, diametrul D şi numărul de arce N pentru cele mai importante topologii.

Notând un procesor oarecare cu P_0, şi pe celelalte în funcţie de acesta, se obţine aceeaşi construcţie în toate cazurile. Cu alte cuvinte, toate procesoarele au acelaşi context local, toate "văd" ansamblul în acelaşi fel; aceasta este o proprietate esenţială în dezvoltarea algoritmilor: un procesor nu cunoaşte decât adresa sa şi direcţiile pe care se află canalele sale de comunicaţie, deci vecinii săi; vom discuta aceste aspecte în capitolul următor. În plus, toate nodurile au aceeaşi excentricitate, egală cu diametrul. Din aceste motive, unii algoritmi vor fi prezentaţi în forma pentru P_0, pentru inel sau hipercub, sau P_{00}, pentru tor, fără ca particularizarea să fie simplificatoare; vom numi *central* acest procesor.

Produs cartezian. Topologiile de mai sus pot fi descrise alternativ sub forma unui produs cartezian. Notăm $\mathcal{I}(p) = 0 : p-1$; asociem elementelor lui $\mathcal{I}(p)$, văzute ca noduri ale unui graf, legături care formează un inel. Cu această convenţie, identificăm $\mathcal{I}(p)$ cu un inel. Se observă imediat că torul $m \times n$ este produsul cartezian $\mathcal{I}(m) \times \mathcal{I}(n)$. De asemenea, hipercubul este $\mathcal{I}^d(2) = \{0,1\}^d$.

Comparaţii. Recapitulăm principalele caracteristici ale topologiilor prezentate în tabelul 1.1, unde presupunem că grila şi torul au acelaşi număr de procesoare în orice dimensiune; lăsăm cititorului demonstraţia proprietăţilor torului 3D. În tabel, topologiile sunt ordonate crescător după grad şi descrescător după diametru.

Reţele configurabile. Reţelele de interconectare pot fi şi **configurabile**, de exemplu prin intermediul unor comutatoare în cruce (crossbar). Un astfel de comutator permite conectarea a $2n$ linii, putându-se realiza conectarea între orice linie dintr-un grup de n cu orice alta din celălalt grup de n. Se poate concepe o arhitectură ce permite oricăror două procesoare să fie vecine. Complexitatea reţelei este însă foarte mare, de aceea nu se construiesc crossbar cu mai mult de 32 de intrări; se folosesc în schimb diverse combinaţii de crossbar-uri, care permit realizare unui număr mare de topologii. Ca exemple, amintim calculatoarele din familia supernodurilor (din anii '90), în care procesoarele erau transputere şi care puteau realiza orice topologie cu grad mai mic sau egal cu patru (numărul de canale al transputerului); de asemenea, calculatoarele actuale Sun 10000 Starfire, HP 9000 Superdome şi SGI Origin 3000 (ultimele două sunt formate din clustere conectate prin comutatoare în cruce).

Alte topologii. Pornind de la constrângerile constructive legate de grad şi di-

ametru, se poate formula *problema* (Δ, D): care este numărul maxim de noduri al unui graf cu grad Δ şi diametru D ? Altfel spus, să se găsească topologia cu grad şi diametru impuse, dar cu cel mai mare număr de procesoare; problema este pusă oarecum pe dos: ar fi fost poate mai natural să se impună gradul şi numărul de procesoare şi să se minimizeze diametrul; în fond, este exact acelaşi lucru. Problema nu e decât parţial rezolvată. De exemplu, pentru $\Delta = 2$, inelul este cel care atinge optimul. Este curios că grafurile obţinute în mod aleator au de multe ori foarte bune proprietăţi din acest punct de vedere; ele sunt însă inutilizabile, datorită neregularităţii lor. Oricum, formularea problemei a condus la o multitudine de propuneri de posibile topologii, care, deşi nu au fost puse în practică la scară mare, dovedesc bune proprietăţi teoretice, nu numai din punctul de vedere (Δ, D), ci şi din cel al adecvării diverşilor algoritmi la aceste topologii. Menţionăm aici doar grafurile *de Bruijn* şi *Cube-Connected-Cycle*.

Tendinţe actuale. Până în 1990, hipercubul părea a fi topologia cea mai promiţătoare. După aceea, grila şi torul au beneficiat de avantajul ca li se pot adăuga foarte uşor procesoare, cu modificări minime în configuraţia celor existente; grila reprezintă un graf planar (se poate reprezenta în plan fără ca două arce să se intersecteze), iar torul se poate construi în două planuri, de aceea sunt uşor de realizat fizic. Torul 3D beneficiază de avantaje asemănătoare. În prezent, tendinţa este de a face topologia transparentă pentru utilizator; comunicaţia este lăsată sistemului de operare sau unor biblioteci specializate, care permit comunicarea între oricare două noduri. Totuşi, cunoaşterea topologiei este importantă din cel putin două motive:

- mulţi algoritmi folosesc în mod natural o topologie virtuală în care comunicaţiile se desfăşoară numai între vecinii în graf; conceperea algoritmilor se bazează deci pe o topologie de comunicaţie;

- performanţa obţinută cu un program paralel poate fi îmbunătăţită sensibil atunci când topologia virtuală a algoritmului corespunde topologiei reale a calculatorului.

1.2.4 Scufundări

Aşa cum am sugerat mai sus, în proiectarea de algoritmi paraleli se preferă o oarecare distanţare faţă de arhitectură, pentru a nu fi necesară rescrierea periodică a algoritmilor, la eventuale schimbări de arhitectură. O cale de a realiza portabilitatea este de a gândi algoritmi pe topologii simple (această simplitate poate fi văzută în două feluri: topologia în sine este simplă sau topologia este adecvată algoritmului, deci simplă din punctul de vedere al acestuia; avem în vedere ambele sensuri) şi de a găsi o metodă de "suprapunere" a acestei topologii cu cea a arhitecturii reale; sunt, aşadar, două nivele: unul al legăturilor logice, tipice algoritmului, altul al legăturilor fizice, specifice maşinii; se încearcă o potrivire a legăturilor logice cu cele fizice; vom reveni mai pe larg în capitolul 4.

Problema este interesantă şi din alte motive, de exemplu posibilitatea emulării unei arhitecturi prin alta şi determinarea efortului cerut de această emulare; să ne postăm deocamdată în acest punct de vedere. Această problemă este cunoscută sub numele de *scufundare a grafurilor*.

Vom prezenta întâi formularea matematică a problemei, explicând apoi semnificaţia noţiunilor, de altfel destul de intuitive. Un studiu mai general poate fi găsit [22].

Definiţii. 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 mulţime arcelor din G, cu valori în mulţimea căilor din H; prin S_f se asociază unui arc xy din G, o cale în H care uneşte $f(x)$ cu $f(y)$.

De obicei se vorbeşte despre o scufundare doar prin intermediul funcţiei f, lăsând S_f subînţeleasă (unui arc xy asociindu-i-se una din căile de lungime minimă între $f(x)$ şi $f(y)$). Pentru a măsura eficienţa unei scufundări se folosesc mai mulţi parametri, dintre care amintim pe cei mai importanţi:

- *dilatarea* unei scufundări, notată $dil(f)$, este lungimea maximă a unei căi $S_f(xy)$, cu $x, y \in V(G)$, unde $V(G)$ este mulţimea nodurilor grafului G. În cazul în care se consideră căile cele mai scurte—deci lungimea căii $S_f(xy)$ este $dist_H(f(x), f(y))$, adică distanţa pe H dintre $f(x)$ şi $f(y)$—atunci dilatarea se exprimă unic în funcţie de f prin (se notează cu $E(G)$ mulţimea arcelor din G):

$$dil(f) = \max_{xy \in E(G)} dist_H(f(x), f(y)).$$

- *congestia* unei scufundări f a lui G în H este maximul, după toate arcele $e \in E(H)$, a numărului de căi $S_f(xy) \in E(H)$, imagini de arce din G, care conţin e.

Putem privi H ca fiind arhitectura reală, iar G topologia emulată (fie că ea provine dintr-un algoritm, fie că este realmente a unei arhitecturi). Dilatarea exprimă distanţa reală 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 acest număr este mai mic, conflictele de utilizare a canalului vor fi mai reduse; valoarea ideală este 1; dacă dilatarea este egală cu 1, atunci şi congestia este tot 1.

Exemple. Vom prezenta doar câteva rezultate fundamentale, cu aspect intuitiv foarte pregnant, de aceea fără detalii. Ne vom referi la legăturile între topologiile cele mai curente, anume inel, grilă, tor şi hipercub.

O primă problema este de a afla dacă un inel poate fi scufundat cu dilatare 1 în topologiile de mai sus sau, cu alte cuvinte, dacă se poate găsi pentru acestea un *ciclu hamiltonian*.

O grilă 2D de dimensiuni $m \times n$ are ciclu hamiltonian dacă şi numai dacă m sau n sunt pare ($m, n > 2$). Un exemplu semnificativ în ce priveşte construirea ciclului este prezentat în figura 1.11a, pe o grilă 4×5. Într-o grilă cu ambele dimensiuni impare se poate scufunda un inel cu dilatare 2; se observă din figura 1.11b că linia oblică nu

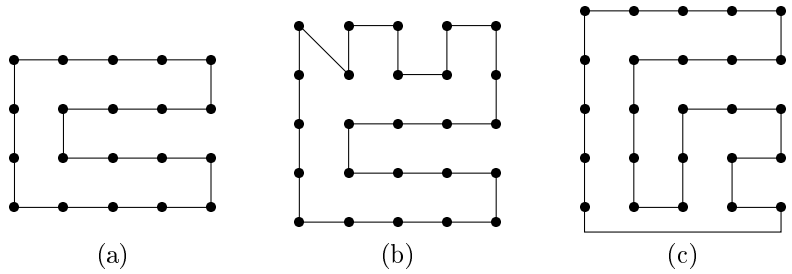


Figura 1.11: (a) Ciclu hamiltonian într-o grilă 2D 4×5; (b) scufundarea unui inel într-o grilă 5×5, cu dilatare 2; (c) ciclu hamiltonian într-un tor 5×5.

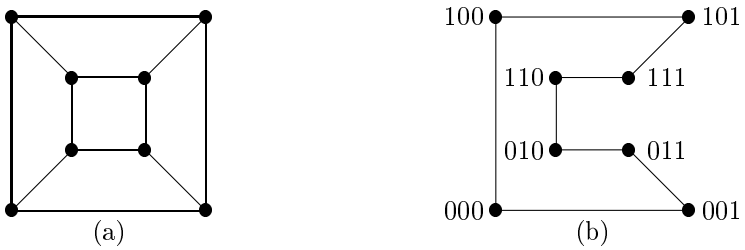


Figura 1.12: (a) un alt mod de desenare a unui hipercub de dimensiune 3 (b) ciclu hamiltonian în hipercubul de dimensiune 3.

corespunde unui arc al grilei; această unică conexiune trebuie realizată prin două arce ale grilei; în ambele variante posibile, dreapta-sus sau stânga-jos, congestia este 2.

Un tor 2D are întotdeauna ciclu hamiltonian; dacă m sau n sunt pare, se aplică rezultatul precedent; dacă ambele dimensiuni sunt impare, construcţia este (de exemplu) asemănătoare cu cea din figura 1.11c, pentru un tor 5×5.

De asemenea, un hipercub are întotdeauna un ciclu hamiltonian (de fapt se pot construi d cicluri disjuncte pentru un hipercub $2d$-dimensional, aşa cum se arată în [19]). Ordinea în care se parcurg nodurile este cea din codul Gray reflectat. Un cod Gray de dimensiune d este o permutare a celor 2^d numere binare pe d biţi, astfel încât două numere consecutive diferă printr-un singur bit (nodurile cu numerele vecine sunt vecine în hipercub). De exemplu, codul Gray reflectat se construieşte recursiv, după cum urmează: $G_1 = (0, 1)$, $G_d = (0G_{d-1}, 1G^R_{d-1})$, i.e codul de dimensiune d se obţine din codul de dimensiune $d-1$, căruia i se adaugă cifra 0 pe poziţia cea mai semnificativă, urmat de acelaşi cod de dimensiune $d-1$, dar în ordine inversă, cu cifra 1 adăugată pe poziţia cea mai semnificativă. Deci, $G_2 = (00, 01, 11, 10)$, iar G_3 poate fi văzut, în ciclul hamiltonian al unui hipercub de dimensiune 3, în figura 1.12.

O grilă 2D de dimensiuni $2^{n/2} \times 2^{n/2}$ poate fi scufundată cu dilatare 1 într-un

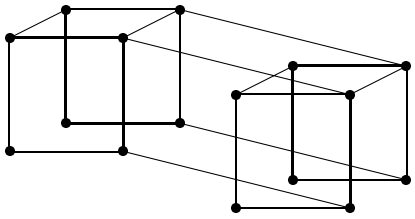


Figura 1.13: Grilă 4×4 mapată într-un hipercub de dimensiune 4; un tor 4×4 este chiar hipercubul !

hipercub n-dimensional (cu n par), ca şi, de altfel, orice grilă de dimensiuni $2^{n_1} \times 2^{n_2}$, cu $n_1 + n_2 = n$; un exemplu este prezentat în figura 1.13. Şi pentru torul 2D este valabil acest rezultat.

Probleme

P 1.2.1 Să se arate că numărul de arce dintr-un arbore de ordin p este $p - 1$.

P 1.2.2 Care este numărul de arce într-un graf regulat de ordin p şi grad Δ ?

P 1.2.3 Să se arate că distanţa între două noduri oarecare cu adresele x, y din hipercubul $\mathcal{H}_d$ este egală cu numărul de poziţii în care cifrele din x şi y diferă (distanţa Hamming între x şi y).

P 1.2.4 Câte noduri se află la distanţă x de P_0, într-un hipercub $\mathcal{H}_d$?

P 1.2.5 Care sunt vecinii procesorului P_{ij} într-un ciclu hamiltonian pe o grilă $n \times n$, cu n par ?

P 1.2.6 Cum se face conversia între reprezentarea binară a unui număr şi codul său Gray (reflectat) ? Care sunt vecinii procesorului P_g într-un ciclu hamiltonian în hipercubul $\mathcal{H}_d$? Caz particular: vecinii lui P_4 într-un ciclu hamiltonian în $\mathcal{H}_4$.

1.3 Criterii de performanţă

Vom prezenta pe scurt câteva criterii prin care se apreciază posibilităţile hardware ale unui calculator paralel:

- *Viteza de calcul* maximă, măsurată în Mflop/s (Million FLoating point OPeration per Second) sau, mai rar, în MIPS (Million Instructions Per Second). În primul caz se măsoară numărul de operaţii în virgulă mobilă efectuate într-o secundă (alte prefixe: G pentru giga—10^9 operaţii, sau T pentru tera—10^{12} operaţii), în al doilea numărul de instrucţiuni. În descrierea vitezelor de calcul, se folosesc două valori: una de vârf, care presupune că procesoarele lucrează la încarcare maximă (ignorând

conflictele de acces la memorie, comunicaţia, sincronizările, timpii morţi), şi alta practică, evaluată cu programe speciale de test (benchmark), care rezolvă probleme des întâlnite în calculele ştiinţifice. Unul dintre cele mai utilizate teste este LINPACK, care conţine rezolvarea unor probleme de algebră matriceală, ca de exemplu rezolvarea de sisteme de ecuaţii liniare. Recordul mondial actual este de peste 7 Tflop/s, realizat de calculatorul IBM ASCI White de la Lawrence Livermore National Laboratory, în anul 2000, şi măsurat la rezolvarea unui sistem de 518000 ecuaţii liniare.

- *Numărul de procesoare*. Un calculator paralel, în special în clasa MIMD cu memorie distribuită, nu are un număr fixat definitiv de procesoare. Datorită puterii de cumpărare şi necesităţilor diferite ale potenţialilor clienţi, se proiectează variante arhitecturale cu număr de procesoare de la câteva zeci la câteva mii, diferenţele conceptuale între acestea fiind minime.
- *Dimensiunea memoriei* locale a unui procesor, sau a memoriei comune, dacă există. Acesta e un parametru de mare importanţă, deoarece dictează dimensiunea problemelor care pot fi rezolvate. Pentru calculatoarele MIMD cu memorie distribuită actuale, memoria locală a unui procesor este de ordinul Go.
- *Frecvenţa* ceasului unui procesor, care dă informaţii despre puterea unui nod. În prezent, procesoarele utilizate în calculatoarele paralele lucrează la frecvenţe de sub 1 GHz; totuşi, datorită paralelismului lor funcţional, ele pot furniza viteze de calcul de peste 1 Gflop/s.
- Caracteristici legate de *viteza de comunicaţie* şi în special banda de trecere a unei legături, măsurată în număr de octeţi transferaţi pe secundă. Valorile curente sunt de ordinul 500 Mo/s. Totuşi, ca şi la viteza de calcul, trebuie să diferenţiem între viteza de vârf, valabilă doar pentru mesaje foarte lungi (când nu se ia în considerare timpul de amorsare a comunicaţiei etc.) şi cea atinsă efectiv în aplicaţii reale. Vom discuta despre caracteristicile comunicaţiei în capitolul 3.
- *Viteza de acces la memorie* a fiecărui procesor, esenţială pentru calculatoarele cu memorie comună. Valorile actuale ale acestei viteze sunt de ordinul sutelor de Mo/s.
- Viteza de comunicaţie cu exteriorul, care poate produce un efect de strangulare a vitezei efective de calcul, prin introducerea unor timpi morţi datoraţi nealimentării cu programe şi date a calculatorului paralel.
- Suportul software oferit o dată cu calculatorul. În ultimii ani, se constată un efort intens de standardizare, materializat în apariţia de biblioteci implementate eficient pe toate arhitecturile (sau pe clase de arhitecturi). Aceasta permite portabilitatea programelor paralele, ceea ce acum 10 ani părea un ţel îndepărtat. Vom prezenta câteva biblioteci importante în această carte.
- Raportul între costul calculatorului şi viteza sa de calcul sau, mai popular, *preţul* unui Mflop/s. Acesta, ca peste tot, e de multe ori hotărâtor. Tendinţa actuală este de a menţine cât mai scăzut acest raport, chiar dacă nu se obţine o viteză de calcul deosebit de ridicată.

Este clar că trebuie să existe o armonie între valorile acestor parametri. De exemplu, nu are sens ca viteza de comunicaţie să fie mult mai mare decât viteza de calcul:

un element în virgulă mobilă să poată fi mult mai repede transmis decât calculat. Evident că nici situaţia inversă nu e de dorit. Există, de asemenea, corelaţii între viteza de calcul şi capacitatea memoriei; aceasta din urmă trebuie gândită după dimensiunea problemelor pe care calculatorul le poate rezolva în timp rezonabil. Criteriul principal de apreciere a unui calculator paralel rămâne totuşi viteza de calcul.

Lista celor mai rapide 500 de calculatoare din lume, actualizată semestrial, poate fi găsită la `http://www.top500.org`. Pe primele locuri se găsesc calculatoare de construcţie specială, în majoritate unicate, cu mai mult de 1000 de procesoare. Pentru ilustrarea criteriilor de mai sus, prezentăm fişa tehnică a două calculatoare comerciale.

Cray T3E este un calculator MIMD cu memorie distribuită, cu o topologie de tor 3D; nodurile sunt construite cu procesoare DEC Alpha 21164, cu o frecvenţă a ceasului de 675 MHz şi o performanţă de vârf de 1.35 Gflop/s (procesorul are două unităţi de calcul în virgulă mobilă), şi cu o memorie de 2Go. Numărul de procesoare poate varia între 40 şi 2176. Viteza maximă de comunicaţie pe fiecare legătură între noduri este de 325 MB/s, bidirectional. Viteza maximă de calcul atinsă este de 1.127 Tflop, pentru rezolvarea unui sistem liniar de dimensiune 148800, folosind 1488 procesoare. Mai multe informaţii se pot obţine la `http://www.cray.com`.

Sun Enterprise E10000 este un calculator MIMD cu memorie comună, având 4 până la 64 procesoare UltraSPARC, funcţionând la o frecvenţă de 400 MHz şi având o performanţă de vârf de 800 Mflop/s. Capacitatea memoriei comune este de până la 64 Go, iar viteza de acces la memorie de 200-400 Mo/s pe procesor (valoarea minimă este pentru numărul maxim de procesoare). Viteza maximă raportată este 43.8 Gflop/s (test LINPACK). Mai multe informaţii se pot obţine la `http://www.sun.com`, iar pentru exemplarul instalat in Universitatea Politehnica Bucureşti, la `http://hpc.ss.pub.ro`.

Capitolul 2

Modele de programare

În acest capitol vom prezenta modele de programare pentru calculatoarele MIMD, cu memorie distribuită sau comună. Aceste modele ne vor permite descrierea precisă a algoritmilor într-un pseudocod care poate fi translatat cu uşurinţă în formă implementabilă pe un calculator paralel.

Paradigma unică de programare folosită în această carte şi în calculatoarele MIMD actuale este SPMD (Single Program Multiple Data): toate procesoarele execută acelaşi program, dar fiecare utilizează un set propriu de date. Acest mod de lucru combină versatilitatea arhitecturilor MIMD cu facilitatea programării.

Organizarea memoriei în arhitectura paralelă determină caracteristicile secundare ale modelului de programare, referitoare la modul în care cooperează procesoarele. În cazul memoriei distribuite se lucrează cu modelul de comunicaţie prin mesaje; mai precis, procesoarele comunică numai prin intermediul unor mesaje transmise de la un procesor sursă la unul destinaţie; aşadar, comunicaţia este explicită. În cazul memoriei partajate, comunicaţia se face implicit prin memoria comună; modelul de programare se bazează pe arhitectura ideală PRAM (Parallel Random Access Machine).

Toate aceste modele vor fi detaliate în continuare. De asemenea, dedicăm o secţiune consistentă standardului de comunicaţie prin mesaje MPI (Message Passing Interface).

2.1 Descrierea algoritmilor SPMD

Deşi în stilul SPMD toate procesoarele execută acelaşi program, trebuie făcute două precizări importante:

- Execuţia instrucţiunilor nu este sincronă, adică fiecare procesor execută instrucţiunile în ritmul său propriu.
- Nu toate procesoarele execută aceleaşi instrucţiuni, adică deşi programele încărcate în memoriile procesoarelor sunt identice, totuşi pot exista diferenţe la execuţie.

Prima precizare este justificată de însăşi arhitectura MIMD, în care nu există sincronism. A doua necesită detaliere, deoarece pare în contradicţie cu definiţia modelului SPMD.

Adresa unui procesor. Un program se execută pe p procesoare ale unui calculator paralel; în general, valoarea lui p este mai mică decât numărul total de procesoare al calculatorului, în funcţie de dorinţa utilizatorului, care ia în seamă dimensiunile problemei pe care o are de rezolvat şi gradul de încărcare a calculatorului. (În general, noi nu vom face distincţie între p şi numărul total de procesoare, deoarece nu este important din punct de vedere algoritmic.) Procesoarele sunt numerotate de la 0 la $p-1$, dar această numerotare nu este statică, ci se face în momentul lansării în execuţie. Totuşi, procesoarele pot să-şi afle adresa proprie, printr-o primitivă a sistemului de operare sau de bibliotecă. În descrierea algoritmilor, vom preciza că, de exemplu, adresa este conţinută de variabila k, spunând că algoritmul e valabil pentru procesorul P_k; implicit, variabila conţinând adresa procesorului curent va fi numită id. Procesoarele pot executa instrucţiuni diferite în funcţie de adresa lor. De exemplu, dacă programul unic al procesoarelor arată astfel

1. **dacă** id $= 3$ **atunci**
    1. $a_{\mathrm{id}} \leftarrow 1$
2. **altfel**
    1. $a_{\mathrm{id}} \leftarrow 0$

atunci doar procesorul P_3 execută instrucţiunea $a_{\mathrm{id}} \leftarrow 1$, în timp ce toate celelalte execută $a_{\mathrm{id}} \leftarrow 0$. În consecinţă, în final, vectorul $a \in \mathbb{R}^p$ are elementele $a_k = 0$, $k \neq 3$, şi $a_3 = 1$.

Variabile. Indici locali şi indici globali. În descrierile algoritmilor, variabilele nu vor fi declarate; utilizarea unei variabile înseamnă implicit declararea ei; tipul variabilei (întreg, real etc.) va rezulta din context. În privinţa locului în care se află o variabilă în memorie şi a modului în care ne referim la ea, trebuie să facem o distincţie importantă.

- În cazul memoriei *comune*, fiecare variabilă se află în aceasta, iar referirea la variabile se face la fel ca în programele secvenţiale. Micul exemplu de mai sus este elocvent, elementele vectorului a fiind referite cu indicii lor reali.
- În cazul memoriei *distribuite*, un procesor execută programul unic utilizând date din memoria sa *locală*. Aşadar, o variabilă a unui program SPMD este multiplicată: fiecare procesor deţine un exemplar, asupra căruia are control complet. Un procesor nu poate modifica variabile din memoria altui procesor. Prin urmare, în exemplul de mai sus, *fiecare* procesor deţine doar un element al vectorului $a \in \mathbb{R}^p$, anume elementul al cărui indice este egal cu adresa procesorului. În mod natural, vom nota acest element tot cu a, însă indicii vor dispărea, iar programul de mai sus va fi transformat astfel:

    1. **dacă** id $= 3$ **atunci**

1. $a \leftarrow 1$
2. **altfel**
   1. $a \leftarrow 0$

De unde în primul caz a era un vector aflat într-o singura zonă de memorie, acum vectorul este distribuit în memoriile locale ale procesoarelor.

Dacă pentru memoria comună lucrurile sunt foarte clare, în cazul memoriei distribuite avem de ales între cele două variante de mai sus, în *prezentarea* algoritmilor. Varianta didactică este de a folosi indici *globali*, ca şi cum vectorii s-ar afla în întregime în memoriile procesoarelor, dar fiecare procesor efectuează numai adunarea sa; în acest caz scriem că P_k execută $a_k \leftarrow 0$, la fel ca pentru memorie comună. Dacă ţinem seama că fiecare procesor are de fapt în memorie un singur element al vectorului, atunci scriem că P_k execută $a \leftarrow 0$; variabila este scalară; indicele a dispărut. Dacă vectorul a ar avea dimensiune mai mare decât p, atunci fiecare procesor ar avea mai multe elemente în memoria locală, stocate contiguu; accesul la ele se va face cu indici *locali*.

În general, vom folosi indicii globali, pentru că permit o descriere mai clară a algoritmilor. Când indicii sunt locali, adică se referă la partea dintr-un vector (sau matrice) aflată în memoria locală a unui procesor, vom menţiona explicit acest lucru; acesta este modul de lucru în programele reale, de aceea vom da suficiente exemple de programe în ambele forme.

Pseudocod. Limbajul în care vor fi descrişi algoritmii va fi mai mult sau mai puţin formal. În general vom folosi un pseudocod format din câteva instrucţiuni obişnuite în limbajele de programare de nivel înalt, aşa cum am şi făcut-o mai sus. Rezumăm aici aceste instrucţiuni, presupunând cititorul familiarizat cu semnificaţia lor.

- Atribuirea se va simboliza cu $\leftarrow$; deci $a \leftarrow 1$ înseamnă că variabila a capătă valoarea 1 (a nu se confunda cu $=$, operatorul semnificând egalitatea). Semnul $\leftrightarrow$ se va folosi pentru interschimbarea valorilor a două variabile.
- Instrucţiunea de decizie va avea forma **dacă** *expresie logică* **atunci** *instrucţiuni*; instrucţiunile se execută dacă condiţia este adevărată. Poate exista şi o ramură care se execută atunci când condiţia este falsă, ramură care va urma cuvântului cheie **altfel**.
- Instrucţiunea **pentru** $i = vi : pas : vf$, *instrucţiuni* reprezintă ciclul cu număr cunoscut de paşi; contorul i ia valori de la valoarea iniţială vi la valoarea finală vf, la fiecare iteraţia adăugându-i-se valoarea pas. Pasul poate lipsi, fiind implicit 1, ca de exemplu în $i = 1 : n$.
- Instrucţiunea **cât timp** *expresie logică, instrucţiuni* este ciclul cu număr necunoscut de paşi şi test iniţial. Instrucţiunile se execută atât timp cât expresia logică este adevărată.
- Instrucţiunea **stop** termină execuţia programului; ea nu e neapărat necesară, dar ne permite să scurtăm descrierile algoritmilor.

Pentru descrierea expresiilor logice sau a celor aritmetice vom folosi semnele matematice uzuale.

Algoritmii vor fi numerotaţi, eventual denumiţi după autorul lor sau cu un nume consacrat, şi vor avea câteva cuvinte explicative la început, rezumând problema pe care o rezolvă şi condiţiile de funcţionare, incluzând, dacă e cazul, adresa procesorului pentru care este valabil algoritmul.

O instrucţiune va ocupa, în general, o singură linie; pentru cele care pot ocupa mai multe linii, adică **dacă**, **pentru** şi **cât timp**, liniile următoare primeia vor fi indentate (aliniate mai spre dreapta), prin această metodă putând fi determinat sfârşitul instrucţiunii; algoritmii vor avea descrieri suficient de scurte pentru a nu mai introduce un cuvânt cheie special pentru terminarea unei instrucţiuni de acest tip. Liniile algoritmului sunt numerotate după o regulă evidentă în exemplul de mai sus: liniile aliniate la acelaşi nivel sunt numerotate în ordine, începând de la 1, până când se întâlneşte o instrucţiune de nivel superior (adică aliniată mai la stânga). Pentru referire se foloseşte numărul liniilor tuturor instrucţiunilor în interiorul cărora se află instrucţiunea curentă, la care se concatenează numărul liniei instrucţiunii. Aşadar, în ultimul exempl de mai sus, instrucţiunea $a \leftarrow 0$ are numărul 2.1.

Descrierea paralelismului. O problemă oarecum delicată este modul în care se descriu acţiuni ce au loc în paralel. Modelul SPMD descrie implicit paralelismul; pentru facilitarea studiului unui algoritm, putem presupune că toate cele p procesoare încep simultan execuţia programelor lor, ceea ce în realitate nu se întâmplă de obicei; a vedea însă în ce măsură operaţiile se desfăşoară într-adevăr în paralel ţine de buna înţelegere a algoritmului; vom încerca să explicăm cât mai des ce anume se întâmplă în paralel şi ce acţiuni sunt condiţionate de altele (capitolul 4 va oferi unele instrumente suplimentare). În afara paralelismului implicit, vom folosi totuşi şi instrucţiunea **în paralel**, în două contexte destul de diferite:

1. Primul, coerent cu cele spuse până acum, este cel al unor acţiuni care se desfăşoară în paralel pe un acelaşi procesor—de exemplu mai multe comunicaţii şi, eventual, calcul.

2. Al doilea va fi cel al unor formulări la nivel global ale algoritmilor (mai puţin riguroase), descriindu-se acţiuni paralele desfăşurate pe procesoare diferite; va fi utilizat mult mai rar, în special atunci când lucrurile vor fi suficient de clare.

În primul caz, instrucţiunea **în paralel** nu înseamnă obligatoriu execuţie paralelă, ci numai potenţialul pentru aceasta. Instrucţiunea va fi utilizată mai ales pentru noduri ale unui calculator cu memorie distribuită (vezi figura 1.7), acolo unde comunicaţia pe diverse canale poate decurge în bună măsură în paralel cu activitatea (de calcul) a procesorului (şi unde este destul de greu de decis în ce măsură este vorba despre paralelism sau concurenţă). Execuţia unei instrucţiuni **în paralel** se termină o dată cu terminarea execuţiei ultimei instrucţiuni componente.

Posibile ambiguităţi. Combinaţia instrucţiunilor **în paralel** şi **pentru** poate da naştere la ambiguităţi: trebuie precizat dacă instrucţiunile din bucla **pentru** se

execută secvenţial sau paralel. În secvenţa de instrucţiuni

1. **în paralel**
   1. **pentru** $i = 0 : n - 1$
      1. *instrucţiuni 1*
   2. *instrucţiuni 2*

instrucţiunea **pentru** 1.1 şi grupul *instrucţiuni 2* din linia 1.2 se execută în paralel; în interiorul instrucţiunii **pentru**, iteraţiile după contorul i se execută secvenţial: se execută grupul *instrucţiuni 1* pentru $i = 0$, apoi aceleaşi instrucţiuni pentru $i = 1$ etc. În schimb, în secvenţa

1. **pentru** $i = 0 : n - 1$, **în paralel**
   1. *instrucţiuni(i)*

se execută în paralel grupul *instrucţiuni(i)* pentru toate valorile lui i între 0 şi $n - 1$.

Confuzii cu cazul precedent pot fi în următoarea situaţie:

1. **pentru** $i = 0 : n - 1$
   1. **în paralel**
      1. *instrucţiuni 1*
      2. *instrucţiuni 2*

când, pentru fiecare iteraţie în i a buclei **pentru**, instrucţiunile 1.1.1 şi 1.1.2 se execută în paralel, dar iteraţiile se execută secvenţial.

2.2 Modelul de comunicaţie prin mesaje

Discutăm în această secţiune numai despre programarea calculatoarelor MIMD cu memorie distribuită. Deoarece fiecare procesor are memoria sa proprie, singura modalitate de comunicare între procesoare este transmiterea de mesaje. Operaţia de bază este cea în care un procesor sursă P_s transmite un mesaj M conţinând date din memoria sa M_s unui procesor destinaţie P_a, care stochează datele în memoria sa M_a. În general, vom presupune că procesoarele sursă şi destinaţie sunt vecine, adică există o legătură fizică între ele, pe care circulă mesajul. (Cazurile în care această regulă nu este respectată vor fi menţionate explicit.)

Primitive de bază. În modelul de programare descris mai sus, numit *model de comunicaţie prin mesaje*, orice operaţie de comunicaţie poate fi descrisă utilizând doar două primitive de comunicaţie: una de transmitere executată de procesorul sursă, pe care o vom numi **send**, alta de recepţie executată de destinatar şi numită **recv**. Sintaxa lor este următoarea

send(*date, dest*)
recv(*date, sursă*)

Pentru ambele moduri de apel, *date* este un nume (o variabilă) din care să rezulte clar locul în memoria locală în care se află datele de transmis (sau în care se stochează datele recepţionate), precum şi lungimea mesajului. Al doilea parametru este un nume ce identifică vecinul cu care se comunică şi deci portul care va efectua comunicaţia; pentru simplitate, atunci când va fi posibil, vom folosi drept nume al vecinului direcţia pe care se găseşte acesta: stânga, dreapta, sus, jos, vest, est, nord, sud; aceste denumiri sunt potrivite pentru inel, grilă sau tor; pentru hipercub va fi folosită dimensiunea pe care se comunică, un număr între 0 şi $d-1$. Aşadar,

send($A(i,:)$, dreapta)

înseamnă operaţia de transmitere a liniei i a matricei A (folosim notaţia în stil Matlab), de dimensiune presupusă cunoscută, către vecinul din dreapta al procesorului ce execută operaţia.

Vom presupune întotdeauna că datele formează o zonă contiguă în memorie, ceea ce nu este neapărat adevărat. Aceasta reprezintă o problemă practică de cea mai mare importanţă în eficienţa comunicaţiei; necontiguitatea datelor în memorie implică fie transmiterea mai multor mesaje, ceea ce înseamnă de regulă un consum mai mare de timp, fie copierea datelor într-o zonă contiguă, deci folosind operaţii suplimentare. De aceea, modul în care sunt organizate matricele în memorie—pe linii sau pe coloane—nu poate fi neglijat.

Corectitudinea comunicaţiei. Este evident că primitivele **send** şi **recv** trebuie să apară în perechi, pe ansamblul procesoarelor; orice operaţie de transmisie a unui procesor este însoţită de una de recepţie a unui vecin al său. Acesta este un prim criteriu de corectitudine a comunicaţiei într-un algoritm. Iată un singur exemplu, foarte simplu; să presupunem că procesorul P_k trebuie să transmită un acelaşi mesaj M vecinilor săi pe un inel, P_{k-1} şi P_{k+1}. Algoritmul are forma următoare

1. **dacă** id $= k$ **atunci**
   1. **send**(M, dreapta)
   2. **send**(M, stânga)
2. **altfel dacă** id $= (k-1) \bmod p$ **atunci** **recv**(M, dreapta)
3. **altfel dacă** id $= (k+1) \bmod p$ **atunci** **recv**(M, stânga)

Sunt în total două apeluri **send** şi două **recv**, deci totalul este corect. Primul **send** este efectuat în instrucţiunea 1.1 de către P_k, spre dreapta, adică spre procesorul $P_{(k+1) \bmod p}$; la rândul său, acest procesor, în linia 3, execută **recv** din stânga, adică recepţionează de la P_k. Se verifică în mod analog că a două pereche **send-recv**este corectă.

Terminarea locală a comunicaţiei. Să presupunem că procesorul P_s transmite un mesaj procesorului P_a. Deoarece fiecare procesor are fluxul său de instrucţiuni, momentele în care P_s apelează primitiva **send**, iar P_a **recv**, sunt în general diferite. Transmiterea efectivă a mesajului se face doar după ce ambele procesoare au apelat primitivele respective. Se pune întrebarea: ce face procesorul care a apelat primul

primitiva sa de comunicaţie, din momentul apelului şi până când celălalt procesor apelează primitiva pereche ? Sunt două raspunsuri posibile:

1. aşteaptă (fără să facă nimic); este cazul comunicaţiei *blocante*;

2. poate executa alte operaţii; comunicaţia este *non-blocantă*.

Comunicaţie blocantă. Ilustrăm acest tip de comunicaţie în figurile 2.1 şi 2.2, presupunând că procesorul sursă este primul care apelează primitiva de comunicaţie (cazul în care destinatarul face primul apel este obţinut prin simetrie). În figura 2.1, la apelul primitivei **send** de către sursă zona de memorie numită generic "mesaj" conţine datele care trebuie transmise (ceea ce este marcat printr-un dreptunghi haşurat); până în momentul apelării **recv** de către destinatar, procesorul sursă nu execută nici o operaţie, deci este blocat; după apelul **recv** are loc comunicaţia efectivă (marcată în figură cu o săgeată), după care zona "mesaj" devine liberă, adică poate fi utilizată în alte scopuri. Deoarece unul din procesoare aşteaptă fără a face altceva, iar rutinele **send** şi **recv** se termină doar după transferul mesajului, comunicaţia se numeşte blocantă şi *sincronă*.

În figura 2.2, comunicaţia blocantă are loc printr-un *buffer*. Primitiva **send** mută mesajul din zona sa originală într-un buffer; după aceasta, execuţia **send** se termină, zona de memorie a mesajului putând fi refolosită. Mesajul trece din buffer la destinaţie atunci când se execută **recv**; de comunicaţia efectivă se ocupă o rutină de bibliotecă sau a sistemului de operare (deci la nivel ierarhic inferior), care lucrează în paralel (sau concurent) cu programul principal, utilizând facilităţile arhitecturale pentru comunicaţie, de exemplu DMA în figura 1.7. Procesorul sursă este blocat doar în timpul în care mesajul este copiat în buffer. Observăm că terminarea rutinei **send** la sursă nu este condiţionată de apelul **recv** la destinaţie.

Atât în varianta sincronă cât şi în cea prin buffer, la terminarea execuţiei primitivei **send**, zona de memorie în care era memorat mesajul poate fi refolosită, iar la terminarea execuţiei **recv**, mesajul este recepţionat în zona de memorie alocată acestui scop.

Comunicaţie non-blocantă. În modul non-blocant, primitivele **send** şi **recv** se termină *imediat* după apel, fără ca zona de memorie a mesajului să fie liberă la **send** sau să conţină mesajul la **recv**. Primitivele au doar rolul de a iniţia comunicaţia, transferul efectiv fiind realizat mai târziu la un nivel inferior. După terminarea execuţiei **send** sau **recv**, procesorul poate executa alte operaţii, în paralel sau concurent cu comunicaţia. Şi în modul non-blocant comunicaţia se poate desfăşura sincron sau prin buffer; prima variantă este ilustrată în figura 2.3; cealaltă este lăsată ca exerciţiu cititorului.

Este evident că primitivele de comunicaţie non-blocante trebuie să fie însoţite de posibilitatea de a afla când se termină comunicaţia efectivă, mai exact de a afla când zona de memorie a mesajului poate fi refolosită de procesorul sursă sau poate fi utilizată de procesorul destinaţie. În acest scop se introduce primitiva **wait**(pe care o

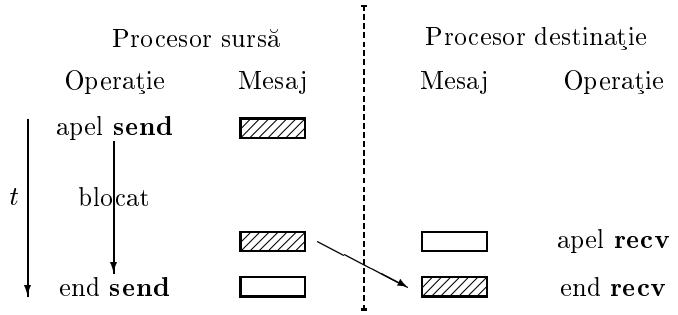


Figura 2.1: Comunicaţie blocantă între două procesoare, modul sincron.

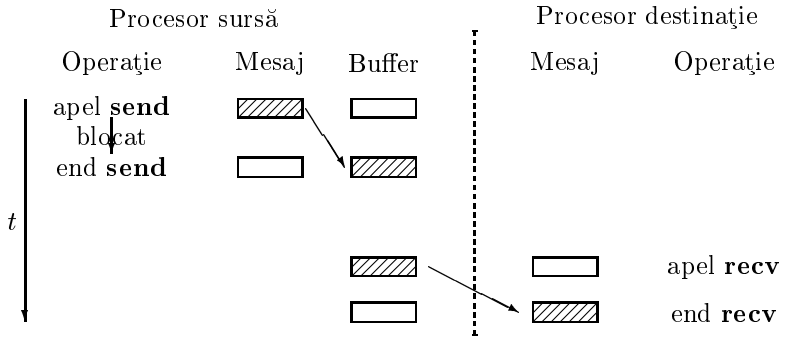


Figura 2.2: Comunicaţie blocantă prin buffer.

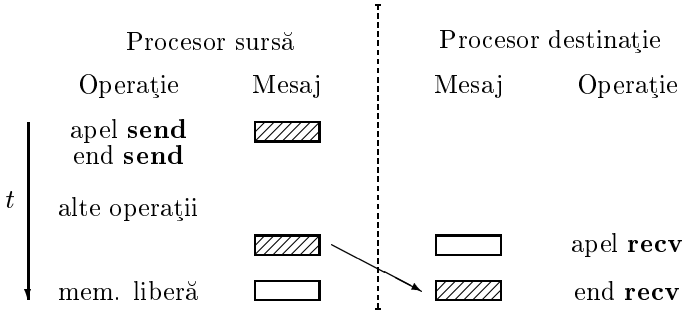


Figura 2.3: Comunicaţie non-blocantă sincronă.

vom folosi mai ales în traducerea **aşteaptă**), de aşteptare a terminării comunicaţiei. Aşadar, utilizarea tipică a rutinelor non-blocante este

1. **send**(*date*, dest)
2. execută operaţii care nu modifică *date*
3. **aşteaptă** terminarea **send**
4. modifică *date*

În mod evident, dacă **aşteaptă** este executată imediat dupa **send** sau **recv**, efectul este acelaşi ca în cazul primitivelor blocante.

Comparaţii. Comunicaţia non-blocantă este mai flexibilă şi este mai folosită, în special în cazul tipic când există coprocesoare specializate pentru comunicaţie; ea are avantajul de a permite execuţia altor operaţii, de către procesor, în timpul comunicaţiei, şi deci economisirea timpului unităţii centrale. Desigur, programarea concurentă a comunicaţiei cu alte operaţii este uneori mai dificilă; pot apărea erori de gestionare a zonelor de date, cu manifestări diferite în funcţie de desfăşurarea în timp a comunicaţiei.

Convenţii de apel. Păstrăm un singur nume pentru rutinele de comunicaţie, indiferent de modul blocant sau non-blocant. Contextul va decide ce mod este folosit. De asemenea, vom folosi relativ rar primitiva **aşteaptă**; în general, vom presupune, pentru claritatea algoritmilor, că operaţiunile de comunicaţie nu se desfăşoară în paralel cu calculele. Atunci când vom dori să evidenţiem acest paralelism o vom face explicit. Să presupunem, de exemplu, că un procesor dintr-un hipercub cu $p = 2^d$ noduri transmite un mesaj tuturor vecinilor săi. În cazul comunicaţiei blocante vom scrie (numai pentru procesorul sursă)

1. **pentru** $i = 0 : d - 1$
   1. **send**(*mesaj*, i)

(Reamintim că i reprezintă dimensiunea pe care se transmite.) Pentru comunicaţia non-blocantă, vom evidenţia desfăşurarea în paralel a mai multor transmisii, scriind

1. **pentru** $i = 0 : d - 1$, **în paralel**
   1. **send**(*mesaj*, i)
2. **aşteaptă** terminarea tuturor operaţiilor **send**

În general, pentru economie de spaţiu, vom *ignora* apelul primitivei **aşteaptă**, presupunându-l implicit.

Erori frecvente în comunicaţia blocantă. Să presupunem că, pe un inel de procesoare, se efectuează următoarea operaţie de comunicaţie colectivă: fiecare procesor transmite vecinului său din dreapta un mesaj. Echivalent, fiecare procesor trebuie să recepţioneze un mesaj de la vecinul său din stânga. Notăm generic cu $d1$ şi $d2$ variabilele locale alocate mesajului propriu, respectiv celui recepţionat. În modul non-blocant, cu convenţiile de mai sus, programul unui procesor poate fi scris în oricare din cele două forme de mai jos.

1. **în paralel**
   1. **send**($d1$, dreapta)
   2. **recv**($d2$, stânga)

1. **send**($d1$, dreapta)
2. **recv**($d2$, stânga)
3. **aşteaptă**terminarea **send** şi **recv**

Vom prefera scrierea din stânga.

În modul blocant, banala secvenţă

1. **send**($d1$, dreapta)
2. **recv**($d2$, stânga)

reprezintă un program incorect, dacă transmisia se face sincron. În acest caz, fiecare procesor începe prin a transmite şi apoi aşteaptă procesorul din dreapta să execute recepţia, ceea ce nu se întâmplă: toate procesoarele se blochează. În schimb, dacă transmisia se face prin buffer, atunci operaţia **send** se termină local la transferul mesajului în buffer (independent de acţiunile procesorului destinaţie), deci recepţia poate să înceapă.

O soluţie simplă de a evita blocarea în cazul blocant sincron este ca parte din procesoare să înceapă prin a transmite, iar celelalte prin a recepţiona; este interesant că oricum se face împărţirea procesoarelor în două mulţimi nevide, comunicaţia va funcţiona. Cel mai rapid algoritm este următorul:

1. **dacă** id este par
   1. **send**($d1$, dreapta)
   2. **recv**($d2$, stânga)
2. **altfel**
   1. **recv**($d2$, stânga)
   2. **send**($d1$, dreapta)

Dacă numărul de procesoare p este par, atunci au loc efectiv două etape de comunicaţie; în prima transmit procesoarele cu adresă pară şi recepţionează cele cu adresă impară, în a doua lucrurile se petrec invers. În schimb, dacă p este impar, au loc trei etape; de exemplu, când $p = 5$, în prima etapă, transmisiile sunt $P_0 \longrightarrow P_1$ şi $P_2 \longrightarrow P_3$; observăm că P_4 nu poate transmite, deoarece vecinul său din dreapta este P_0, care este ocupat. În a doua etapă, au loc transmisiile $P_4 \longrightarrow P_0$ şi $P_1 \longrightarrow P_2$; procesorul P_3 nu poate transmite, deoarece P_4 este ocupat. În fine, în ultima etapă, rămâne o singură transmisie de efectuat, indiferent de numărul de procesoare; în exemplul nostru este vorba despre $P_3 \longrightarrow P_4$.

Pentru a evita complicaţii de tipul celor de mai sus, vom presupune că în modul blocant transmisia se face prin buffer.

Probleme

P 2.2.1 Descrieţi în pseudocod următoarea operaţie de comunicaţie într-un hipercub: un (unic) procesor P_k transmite un mesaj M tuturor vecinilor săi pe hipercub. Aceeasî problemă pentru tor.

P 2.2.2 Descrieţi în pseudocod operaţia de comunicaţie în care, pe un hipercub, fiecare procesor transmite un acelaşi mesaj M tuturor vecinilor săi. Scrieţi un algoritm general, precum şi unul care să funcţioneze în cazul blocant sincron. Idem pentru tor.

2.3 Standardul MPI

Message Passing Interface (MPI) este un standard descriind primitive de comunicatie—în contextul unui model de programare SPMD cu comunicatie prin mesaje—apărut la mijlocul anilor '90, ca urmare al unui efort comun al cercetătorilor din mediul academic şi din industrie. La ora actuală, MPI este disponibil pe toate calculatoarele MIMD cu memorie distribuită, sub formă de bibliotecă de rutine optimizate pe calculatoarele respective. Scrierea unui program folosind rutinele de comunicaţie MPI asigură portabilitatea; eficienţa comunicaţiei este garantată de implementările performante disponibile ale rutinelor MPI. Vom prezenta aici doar câteva rutine, împreună cu concepte specifice MPI; cititorul interesat poate gasi o documentaţie completă la `http://www.mpi-forum.org`; în afara implementărilor pe calculatoare paralele, MPI este disponibil şi pentru alte medii, de exemplu Linux sau Windows NT; astfel, scrierea şi testarea unui program utilizând MPI se pot face extrem de uşor pe o reţea de câteva PC-uri—desigur, pe probleme de dimensiuni reduse. Standardul MPI este descris pentru limbajele FORTRAN şi C. În cele ce urmează, vom folosi numai convenţiile de apelare a funcţiilor C. Numim program MPI un program care utilizează MPI pentru comunicaţie.

Generalităţi. O rutină MPI se execută în cadrul unui grup de procesoare, numit *comunicator*. Iniţial, există un singur comunicator, numit `MPI_COMM_WORLD`, conţinând toate procesoarele pe care se execută programul. Pentru multe aplicaţii, acest comunicator este suficient; există rutine MPI care permit crearea dinamică a grupurilor de procesoare. În cadrul unui comunicator, procesoarele sunt numerotate de la 0 la $p-1$, unde p este numărul de procesoare din comunicator. Un procesor poate afla numărul de procesoare din comunicator şi adresa proprie (numită *rang* în MPI) apelând următoarele rutine:

```
int MPI_Comm_size(MPI_Comm com, int *p)
int MPI_Comm_rank(MPI_Comm com, int *my_id)
```

În ambele rutine, `MPI_Comm` este tipul predefinit pentru comunicatoare; variabila `com` este comunicatorul în care se află procesorul, în cazul cel mai general `MPI_COMM_WORLD`. În variabilele `p` şi `my_id`, rutinele returnează valorile numărului de procesoare, respectiv a adresei procesorului apelant. Toate rutinele MPI (care reprezintă funcţii C) întorc un întreg caracterizând succesul execuţiei: `MPI_SUCCES` inseamnă execuţie corectă, alte valori reprezintă coduri de eroare predefinite.

Structura unui program MPI. Încadrându-se în modelul SPMD, un program MPI are forma obişnuită a unui program secvenţial. Rutinele `MPI_Init` şi `MPI_Finalize` trebuie apelate înaintea, şi respectiv după, orice alte rutine MPI (menţionăm ca `MPI_Init` primeşte argumentele `argc` şi `argv`, acestea având sau nu o

Comunicaţie	blocantă	non-blocantă
standard	MPI_Send MPI_Recv	MPI_Isend MPI_Irecv
prin buffer	MPI_Bsend	MPI_Ibsend
sincronă	MPI_Ssend	MPI_Issend

Tabelul 2.1: Numele rutinelor MPI, pentru diverse tipuri de comunicaţie.

semnificaţie în funcţie de implementarea MPI). Aceasta este singura constrângere, în rest programatorul este liber sa folosească rutinele MPI potrivite algoritmului pe care îl implementează.

Rutine de comunicaţie. În modelulul MPI, oricare două procesoare îşi pot transmite mesaje. Aşadar, topologia virtuală este graful complet conectat. Calea pe care o parcurge efectiv mesajul este transparentă pentru utilizator. Principalele rutine de comunicaţie sunt `MPI_Send` şi `MPI_Recv`, a căror sintaxă completă este următoarea

```
MPI_Send(void *mes, int lung, MPI_Datatype tipdate, int dest,
         int eticheta, MPI_Comm com)
MPI_Recv(void *mes, int lung, MPI_Datatype tipdate, int sursa,
         int eticheta, MPI_Comm com, MPI_Status *stare)
```

La execuţia `MPI_Send` de către un procesor, mesajul aflat în memoria locală la adresa `mes`, de lungime `lung` şi având tipul `tipdate` (MPI are identificatori predefiniţi pentru practic toate tipurile uzuale de date; alte tipuri se pot construi cu rutine speciale) este transmis procesorului `dest`. Mesajului îi este asociată o etichetă (tag), care îl personalizează. Transmisia mesajului se face în cadrul comunicatorului `com`, care trebuie să conţină şi procesorul sursă, şi cel destinaţie. Argumentele rutinei `MPI_Recv` au semnificaţii similare, cu următoarele observaţii: la execuţie, se recepţionează un mesaj de lungime cel mult `lung`, trunchiind eventual mesajul transmis; tipul de date poate diferi la sursă şi destinaţie; în plus, după recepţie, variabila `stare` dă informaţii suplimentare despre mesajul primit, ca de exemplu lungimea efectivă a mesajului.

Moduri de comunicaţie. MPI prevede rutine pentru toate tipurile de comunicaţie descrise în secţiunea 2.2; ne referim la comunicaţie blocantă, respectiv non-blocantă, precum şi la modurile de transmisie prin buffer, respectiv sincron. Numele rutinelor sunt prezentate în tabelul 2.1, regulile de formare a numelor fiind evidente. Trebuie subliniat că modul de implementare a rutinelor de bază (standard) `MPI_Send`, `MPI_Recv` nu este precizat; de exemplu, este posibil ca mesajele scurte să fie transmise prin buffer, iar cele lungi sincron; de aceea programatorul trebuie să-şi ia precauţiile necesare, discutate anterior. Pentru comunicaţia non-blocantă, rutina `MPI_Wait` aşteaptă terminarea transmisiei sau recepţiei, iar `MPI_Test` verifică dacă transmisia sau recepţia s-au terminat sau nu.

Exemple. Vom prezenta aici două exemple de programe MPI, implementând

două operaţii simple de comunicaţie pe inel discutate în secţiunea 2.2. Prima este transmiterea unui mesaj de către procesorul P_0 vecinilor săi, P_1 şi P_{p-1}; programul este prezentat în figura 2.4. A doua operaţie este transmiterea unui mesaj de către fiecare procesor vecinilor săi, realizată de programul din figura 2.5. În ambele cazuri, am prezentat un program MPI complet pentru a ilustra structura acestuia.

Programele sunt suficient de simple pentru a fi clare, aşadar prezentăm numai scurte comentarii. În programul din figura 2.4, lungimea mesajului transmis este aleasă astfel încât să fie transmis şi zeroul de terminare a unui şir de caractere; astfel, procesoarele care recepţionează pot afişa mesajul fără a şti lungimea lui exactă (ci doar că aceasta este mai mică decât 50); o altă variantă ar putea fi transmiterea şirului de caractere fără terminator şi aflarea lungimii la receptori prin intermediul variabilei `stare`. Eticheta asociată mesajului este 99, aceeaşi la sursă şi destinaţie. În cazul e.g. $p = 6$, programul afisează următorul text:

```
0: am trimis:   mesaj de la 0
1: am primit:   mesaj de la 0
5: am primit:   mesaj de la 0
```

Atragem atenţia că ordinea în care aceste linii sunt afişate este mai mult sau mai puţin aleatoare. Chiar dacă P_0 apelează `printf` înaintea celorlalte două procesoare, nu există garanţia că şi afişarea se face înainte. Cu atât mai mult între P_1 şi P_5 nu poate exista o ordine fixată de afişare. Mai multe execuţii ale programului pot produce ordini diferite la afişare.

În programul din figura 2.5, observăm folosirea rutinei de transmisie prin buffer, care garantează corectitudinea programului; în general, pentru mesaje atât de scurte, `MPI_Send` ar fi putut înlocui cu succes `MPI_Bsend` în practică; recomandăm totuşi utilizarea variantei de program a cărei corectitudine nu depinde de implementarea rutinelor MPI. Rutina `MPI_Buffer_attach` este destinată alocării unei zone de memorie folosită pentru buffere; utilizatorul alege dimensiunea acestei zone în funcţie de o estimare a numărului de mesaje care se pot găsi simultan în buffer (şi a lungimii lor); în exemplul nostru, lungimea 100 ar fi fost suficientă. Calculul adresei procesorului din stanga se face ca în program, adică adunând p, deoarece operaţia modulo din C diferă de cea matematică pentru numere negative.

Alte categorii de rutine. Standardul MPI conţine numeroase alte rutine. Enumerăm aici câteva categorii interesante.

- **Comunicaţie globală.** MPI conţine rutine pentru diverse operaţii de comunicaţie implicând toate procesoarele dintr-un grup (comunicator). Este vorba despre sincronizare, difuzare, distribuţie etc. Aceste operaţii vor fi discutate în capitolul următor, de aceea ne mărginim să semnalăm existenţa lor.

- **Alte operaţii globale**, ca de exemplu calculul maximului unor valori distribuite tuturor procesoarelor unui grup, sau calculul sumei prefixelor. Vom prezenta în capitolele 4 şi 5 algoritmi pentru aceste operaţii.

```
#include <stdlib.h>
#include <mpi.h>

void main(int argc, char ** argv)
{
  int my_id, p;
  MPI_Status stare;
  char mesaj[100];

  MPI_Init(&argc, &argv);                // initializare MPI
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id); // afla adresa proprie
  MPI_Comm_size(MPI_COMM_WORLD, &p);     // afla numar de procesoare

  if ( p < 3 )
  { if ( my_id == 0 )          // doar proc. 0 afiseaza mesaj eroare
      printf("Sunt necesare cel putin 3 procesoare !\n");
    exit(0);
  }

  if ( my_id == 0 )                       // proc. 0 transmite mesajul
  { sprintf(mesaj, "mesaj de la %d", my_id);
    MPI_Send(mesaj, strlen(mesaj)+1, MPI_CHAR,   1, 99, MPI_COMM_WORLD);
    MPI_Send(mesaj, strlen(mesaj)+1, MPI_CHAR, p-1, 99, MPI_COMM_WORLD);
    printf("%d: am trimis:   %s\n", my_id, mesaj);
  }
  else if ( my_id == 1 || my_id == p-1 )  // proc. 1 si p-1 receptioneaza
  {
    MPI_Recv(mesaj, 50, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &stare);
    printf("%d: am primit:   %s\n", my_id, mesaj);
  }
  MPI_Finalize();                         // terminare MPI
}
```

Figura 2.4: Program MPI pentru operaţia: procesorul P_0 transmite un mesaj procesoarelor P_1 şi P_{p-1}.

```
#include <stdlib.h>
#include <mpi.h>

void main(int argc, char ** argv)
{
  int my_id, p, dreapta, stanga;
  MPI_Status stare;
  char msend[100], mrecv[100], buffer[1000];

  MPI_Init(&argc, &argv);                // initializare MPI
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id); // afla adresa proprie
  MPI_Comm_size(MPI_COMM_WORLD, &p);     // afla numar de procesoare

  dreapta = (my_id + 1) % p;             // adresa procesor dreapta
  stanga = (my_id - 1 + p) % p;          // adresa procesor stanga
  sprintf(msend, "mesaj de la %d", my_id);
  MPI_Buffer_attach(buffer, 1000);
  MPI_Bsend(msend, strlen(msend)+1, MPI_CHAR, dreapta, 99, MPI_COMM_WORLD);
  MPI_Recv(mrecv, 100, MPI_CHAR, stanga, 99, MPI_COMM_WORLD, &stare);
  printf("%d: am primit:   %s\n", my_id, mrecv);

  MPI_Finalize();                         // terminare MPI
}
```

Figura 2.5: Program MPI pentru operaţia: fiecare procesor transmite un mesaj procesorului din dreapta sa, pe un inel.

- **Topologii virtuale.** Pentru creşterea performanţelor unui program MPI pe un anume calculator, programatorul poate descrie o topologie virtuală care să corespundă—exact, sau în sensul unei dilatări minime—topologiei reale a calculatorului. Rutinele MPI dedicate descrierii topologiei asociază procesoarelor adresele corespunzătoare poziţiei în topologia reală. MPI permite descrierea unei topologii arbitrare, modelată printr-un graf, aşa cum am văzut în capitolul 1. Rutine speciale sunt dedicate topologiilor ce pot fi descrise printr-un produs cartezian, aici incluzându-se toate topologiile curente: inel, tor (2D sau 3D), hipercub. Programele astfel scrise sunt în continuare portabile, chiar dacă eventual au performanţe mai slabe pe arhitecturi diferite de cea pentru care au fost concepute.

- **Procese.** În versiunea 1 a MPI, fiecare procesor executa un singur proces, reprezentând programul MPI; aşadar, procesele erau create static, de către sistemul de operare, la încărcarea programului în procesoare. Versiunea 2 a adus ca principală noutate posibilitatea de creare dinamică a proceselor; se permite astfel o mai mare flexibilitate în descrierea paralelismului, precum şi execuţia concurentă a mai multor procese pe acelaşi procesor.

2.4 Modelul PRAM

Modelul teoretic al arhitecturii MIMD cu memorie partajată este PRAM (Parallel Random Access Machine), o extensie a modelului RAM pentru arhitectura secvenţială; pentru amănunte, vezi [16]. În acest model, într-o unitate de timp, fiecare procesor poate să citească o dată din memoria comună, să execute o operaţie şi să scrie într-o locaţie de memorie. Se trece deci peste implicaţiile pe care le au structura şi viteza de acces ale memoriei, presupunându-se că timpul de acces la memorie este constant. De asemenea, se presupune că procesoarele încep sincron execuţia fiecărei instrucţiuni a lor; aceasta este o abatere de la modelul SPMD, dar permite descrierea simplă a algoritmilor; vom reveni asupra sincronizării cu exemple.

Modelul PRAM poate avea mai multe variante, în funcţie de posibilitatea accesului simultan la o locaţie de memorie:

- EREW PRAM (Exclusive Read Exclusive Write)—în care o locaţie poate fi citită sau scrisă de către un singur procesor la un moment dat.

- CREW PRAM (Concurrent Read Exclusive Write)—în care, în plus, este permisă citirea unei locaţii de către mai multe procesoare simultan.

- CRCW PRAM (Concurrent Read Concurrent Write)—în care, în plus, mai multe procesoare pot scrie simultan în aceeaşi locaţie; aceasta se poate realiza în mai multe feluri:

  – toate procesoarele scriu aceeaşi valoare—modelul COMMON.

- procesoarele au asociate o prioritate; scrie procesorul cu prioritate maximă; acesta este modelul PRIORITY.
- procesorul care scrie este ales aleator—modelul ARBITRARY.

Din punct de vedere teoretic, modelul este foarte puternic, el permiţând descrierea simplă a algoritmilor şi evidenţierea paralelismului în sine al problemelor, făcând abstracţie de detaliile arhitecturii ţintă (cea pe care se urmăreşte implementarea). Trecerea de la forma PRAM a algoritmilor la implementări practice este relativ simplă, de aceea arhitectura PRAM va fi folosită intens în această lucrare.

Exemplul 1: suma a doi vectori. Să presupunem că vrem să efectuăm adunarea $z \leftarrow x + y$, unde variabilele x, y, z sunt vectori de lungime p. Această operaţie este perfect paralelă, fiecare sumă a două elemente putând fi efectuată independent de câte un procesor. În general, vom scrie că procesorul P_k efectuează operaţia $z_k \leftarrow x_k + y_k$. Vectorii se află în memoria comună, deci fiecare procesor trebuie să acceseze locaţia adecvată, aşa că indicii sunt obligatorii. Cu totul excepţional, putem scrie

1. **pentru** $k = 0 : p - 1$, **în paralel**
   1. $z_k \leftarrow x_k + y_k$

Aici este vorba despre o descriere globală a paralelismului, în care nu se precizează adunarea efectuată de fiecare procesor. Acest mod mai vag de descriere este util doar când evidenţiem paralelismul unor operaţii; va fi utilizat foarte rar în această lucrare.

Exemplul 2. Pentru ilustrarea sincronizării presupuse de modelul PRAM, considerăm calculul de către două procesoare a sumei a patru numere, u_k, $k = 0 : 3$. Un mod rapid de a face calculele este ca procesorul P_0 să efectueze suma $u_0 + u_1$ şi, în paralel, P_1 suma $u_2 + u_3$, după care P_0 să adune cele două sume parţiale. Notând id $= i$, programul corespunzător acestei idei este

1. $v_i \leftarrow u_{2i} + u_{2i+1}$
2. **dacă** $i = 0$
   1. $s \leftarrow v_0 + v_1$

Algoritmii scrişi pentru modelul PRAM presupun (tacit) că procesoarele execută simultan instrucţiunile, adică lucrează sincron; în cazul nostru, avem garanţia că ambele sume parţiale sunt calculate în instrucţiunea 1, înainte de a se trece mai departe la instrucţiunea 2. În lipsa sincroniei, este posibil ca P_0 să utilizeze v_1 înainte ca P_1 să pună în această variabilă suma $u_3 + u_4$. Astfel de sincronizări vor fi presupuse implicit întotdeauna, chiar dacă ele nu sunt strict necesare.

Observăm că, în ambele exemple de mai sus, modelul este de fapt EREW PRAM, deoarece nu există citiri sau scrieri simultane în aceeaşi locaţie de memorie. Vom da mai târziu exemple de algoritmi şi pentru alte modele.

Probleme

P 2.4.1 Se poate simula cu un PRAM un calculator MIMD cu memorie distribuită ?

P 2.4.2 Scrieţi un program de care calculează suma numerelor u_k, $k = 0 : 3$, fără a utiliza variabile suplimentare.

Capitolul 3

Algoritmi de comunicaţie

În calculatoarele paralele, comunicaţia este o problemă ce are o cu totul altă natură decât la calculatoarele secvenţiale; la acestea din urmă, comunicaţia se face îndeosebi între un proces (unic, de obicei) ce ocupă unitatea centrală şi exterior. La primele, comunicaţia are loc între procese care se execută pe procesoare diferite şi care colaborează la rezolvarea aceleiaşi probleme; în acelaşi timp există posibilitatea dialogului între procesoare şi exterior, în mod analog cu cazul secvenţial, dar de acest tip de comunicaţie nu ne vom ocupa. Vom studia deci comunicaţia doar atunci când aceasta este parte intrinsecă a algoritmului paralel, transferul de informaţie (cooperarea între procesoare) fiind absolut necesar rezolvării problemei şi oferind cheia eficienţei, adică a execuţiei rapide pe un calculator paralel.

În acest capitol, ne vom ocupa de cazul arhitecturilor MIMD cu memorie distribuită, la care există linii de comunicaţie directe între anumite procesoare, după o topologie specifică fiecărui calculator. Pentru arhitecturile MIMD cu memorie partajată, comunicaţia se face prin intermediul memoriei comune; problemele corespunzătoare acestui caz sunt mai vechi şi au apărut o dată cu calculatoarele secvenţiale cu sistem de operare multitasking; există multă literatură pe această temă, de exemplu [**?**].

Vom detalia o serie de algoritmi, cu precădere în domeniul comunicaţiilor globale—i.e. atunci când mai multe procesoare participă la realizarea comunicaţiei. După cum veţi vedea, tehnicile folosite depind fundamental nu numai de topologia reţelei de interconectare, ceea ce era clar de la început, dar şi de modelul de comunicaţie adoptat; de aceea alegerea unui model adecvat calculatorului ţintă şi sistemului său de operare este o etapă extrem de importantă în proiectarea sau alegerea unui algoritm de comunicaţie. Vom încerca, în acest capitol, să oferim o serie de idei utile nu numai atunci când este vorba exclusiv de comunicaţie, ci şi în alte cazuri, în probleme în care comunicaţia este totuşi preponderentă iar calculele se pot grefa în mod natural pe structura algoritmului de comunicaţie.

3.1 Modele de comunicaţie

Aşa cum am sugerat în figura 1.7, fiecare nod al unei arhitecturi MIMD cu memorie distribuită are unităţi specializate în comunicaţie. În cele ce urmează nu ne vom preocupa prea mult de ceea ce se întâmplă la nivelul acestor unităţi: modul în care sunt legate fizic unităţi aparţinând unor procesoare vecine, protocolul de comunicaţie folosit, etc. Ne vom mulţumi cu faptul că există legături între anumite procesoare, după topologii de genul celor descrise în capitolul anterior şi vom prezenta doar modele generale simple, dar care sunt suficiente în proiectarea algoritmilor.

Toate modelele prezentate în această secţiune se referă la arhitecturi cu memorie distribuită. Principalele aspecte abordate sunt: modul de transmitere a mesajului între procesoare, modul de folosire a canalelor de către un procesor, modelarea timpului necesar pentru comunicarea unui mesaj de lungime dată. Reamintim că o clasificare a comunicaţiei după modul de execuţie local—blocant sau non-blocant—a fost discutată în capitolul anterior.

Modul de transmitere a unui mesaj între procesoare. Problema de bază în comunicaţie este de a transmite un mesaj, de la un procesor oarecare la un altul. În general, pentru a ajunge de la procesorul sursă la cel destinaţie, mesajul trebuie să parcurgă mai multe legături între procesoare vecine (în afara cazului simplu când sursa şi destinaţia sunt vecine). Nu ne punem deocamdată problema cum se găseşte o cale între cele două procesoare—o presupunem cunoscută—ci doar a modului în care este transmis mesajul de către procesoarele de pe această cale.

1. O primă posibilitate are la bază doar comunicaţia între fiecare procesor şi vecinii săi direcţi, pe liniile de legătură cu aceştia. Fiecare mesaj este transmis integral de către posesorul său şi recepţionat de către vecin într-o zonă de memorie înainte de a fi prelucrat sau, eventual, trimis mai departe către destinatar; acest mod de transmitere este numit *store and forward* (memorează şi apoi trimite mai departe) sau *comutare de pachete*. Ilustrăm în figura 3.1 această tehnică; succesiunea etapelor de comunicaţie este reprezentată de sus în jos; mesajul M—având un antet h, care conţine adresa destinatarului şi alte informaţii referitoare la mesaj, de exemplu lungimea acestuia—este transmis de către sursă primului procesor de pe calea spre destinaţie; după recepţie, acesta îl retransmite celui de-al doilea, ş.a.m.d. până la destinaţie. O analogie simplă este cea cu un lanţ de oameni care transmite din mână în mână o cărămidă.

2. Un alt model posibil, frecvent întâlnit în calculatoarele actuale este cel cu *comutare de circuit* (*circuit switched*), în care întâi se stabileşte fizic o cale între sursă şi destinatar, indiferent de distanţa dintre aceştia, şi apoi se transmite direct mesajul destinatarului, după cum se sugerează în figura 3.2. Deci, dacă în prima variantă mesajul se transmitea în mai multe etape identice, în fiecare între două procesoare vecine, fiecare procesor memorând pe rând mesajul, acum mesajul va fi transmis în două etape principial diferite; întâi este trimis antetul h (ca un deschizător de drum) care parcurge calea între sursă şi destinaţie, rezervând legăturile de pe această cale; apoi, procesorul sursă este informat de rezervarea întregii căi, şi deci că poate începe

transmisia; apoi, mesajul este transmis direct destinatarului, pe calea acum rezervată, ca şi cum acesta ar fi vecin cu expeditorul; procesoarele intermediare nu memorează mesajul. Această tehnică este utilizată în telefonie, de exemplu.

3. În fine, o a treia variantă posibilă, asemănătoare ca idee cu comutarea de circuit, este cea *wormhole*. Mesajul M este împărţit în mai multe pachete (*flit*), notate M^i în figura 3.3, care au fiecare lungimea egală cu cea a cozii (zonei de memorie, bufferului) ataşate unui canal; primul flit, pe care-l vom nota h conţine şi adresa destinatarului. Antetul mesajului avansează spre destinaţie de câte ori aceasta este posibil, iar restul mesajului îl urmează; ultimul flit, M^f, eliberează coada canalului pe care o ocupa. Mesajul se deplasează deci ca un vierme, care nu poate fi "tăiat" de alte mesaje, deci ocupă mai multe canale la un moment dat. Avantajul acestui mod de lucru este că un flit nu trebuie stocat în întregime de un procesor; acesta, dacă canalul de ieşire este liber, introduce doar o mică întârziere—de câţiva octeţi—ceea ce conduce la o deplasare rapidă a trupului viermelui, dacă primul flit, capul, se deplasează el însuşi rapid. Dacă, în schimb, canalul de ieşire este ocupat, fiecare flit este memorat de către procesorul la care a ajuns şi transmis mai departe atunci când este posibil. Faţă de comutarea de circuit, se elimină deci etapa de confirmare a rezervării căii de comunicaţie. Din nou comunicaţia este transparentă pentru procesoarele de pe traseu, ele nememorând întregul mesaj.

Modul de utilizare a canalelor. Un procesor are mai multe canale de comunicaţie; el poate folosi un singur canal la un moment dat, sau pe toate în paralel; primul model se numeşte *1-port*, al doilea *n-port* sau *multiport* (sau link-bound, adică numărul maxim de comunicaţii este mărginit numai de numărul de canale fizice ale procesorului). Cele două modele sunt extreme: în primul canalele sunt folosite pur secvenţial, în al doilea în perfect paralelism. În modelul 1-port se presupune de obicei că un procesor poate emite şi recepţiona simultan pe câte un canal, nu neapărat acelaşi. Modelul multiport este idealizat; de exemplu, pentru un procesor cu arhitectură de genul celei din figura 1.7, la care canalele pot transmite sau recepţiona în paralel, modelul nu este totuşi perfect adecvat: programarea canalelor (cu adresa mesajului, lungimea sa) se face în mod secvenţial, de-abia apoi ele lucrând în paralel, dar concurând în accesul la memorie, ceea ce poate eventual întârzia comunicaţia. Cel mai realist, dar şi mai greu de manipulat algoritmic, ar fi un model în care k porturi pot fi folosite simultan, unde $k > 1$, dar mai mic decât numărul de canale al unui procesor.

Canalele vor fi presupuse bidirecţionale, ceea ce este cazul pentru toate calculatoarele actuale; dacă transmisia se poate face în ambele sensuri simultan, modelul va fi numit *full duplex*; dacă se poate folosi un singur sens la un moment dat, modelul este *half duplex*. În modelul full duplex se presupune că transmisia simultană în cele două sensuri ale unei legături se face cu aceeaşi viteză ca în cazul folosirii unui singur sens, ceea ce nu este întotdeauna adevărat.

Ca unitate de măsură pentru lungimea mesajelor se alege de obicei octetul. Totuşi, deoarece în această lucrare ne vom ocupa în special de probleme de calcul numeric, în care datele sunt numere reale reprezentate întotdeauna în virgulă mobilă la imple-

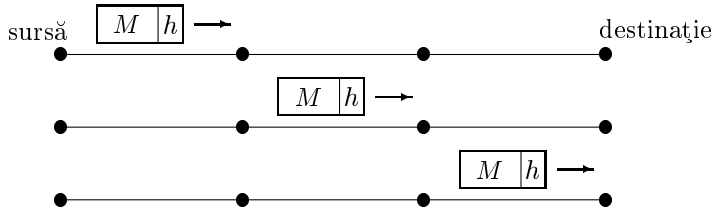


Figura 3.1: Comunicaţie store and forward (comutare de pachete).

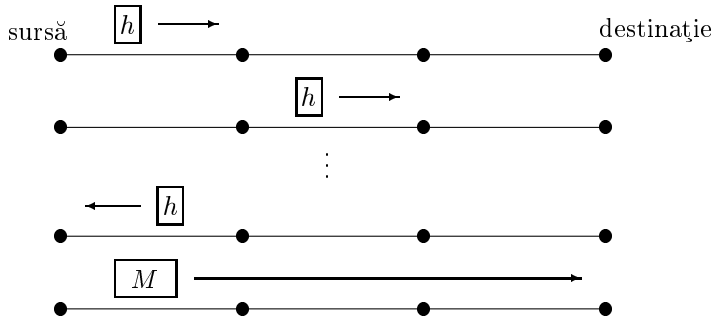


Figura 3.2: Comunicaţie cu comutare de circuit.

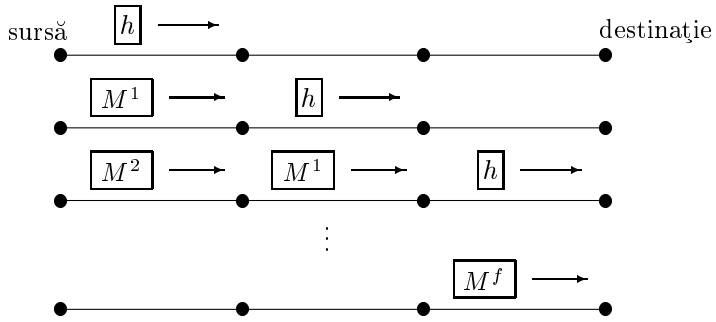


Figura 3.3: Comunicaţie wormhole.

mentare, vom presupune în general dimensiunea mesajelor măsurată nu în octeţi, ci în elemente în virgulă mobilă, făcând abstracţie de numărul de octeţi necesar pentru reprezentarea unui element (care este, pentru multe calculatoare şi limbaje de programare, de 4 octeţi pentru reprezentarea în simplă precizie şi de 8 octeţi pentru reprezentarea în dublă precizie). De asemenea, în alte probleme, vom alege ca unitate de măsură lungimea obiectului fundamental pentru acel algoritm (şir de caractere, de exemplu, la sortare).

Modelarea timpului de comunicaţie. În ce priveşte timpul necesar pentru transmiterea unui mesaj între două procesoare vecine (implicit, deci, presupunem modul de transmisie store and forward) sunt mai multe modele posibile, fiecare adecvat unor anume tipuri de mesaje şi de arhitecturi. Cel mai simplu model este acela în care durata de transmisie a unui mesaj este *constantă*, indiferent de lungimea mesajului; modelul este adecvat cazului în care mesajele se transmit în pachete de lungime fixă—dacă mesajul efectiv este mai scurt, el se completează până la lungimea cerută cu elemente nesemnificative. De asemenea, modelul este bun dacă mesajele sunt relativ scurte (zeci de octeţi, să zicem).

Mai aproape de realitate este modelul *proporţional*, în care timpul de transmisie a unui mesaj de lungime m este

$$t_c(m) = m\beta,$$

unde β este timpul cât durează transmiterea unui element; valoarea $1/\beta$ este denumită *rată de transmisie*.

În fine, cel mai potrivit model, şi deci cel mai folosit, este cel *liniar*:

$$t_c(m) = \sigma + m\beta, \tag{3.1}$$

în care σ este timpul de iniţializare (start-up) a transmisiei; acesta apare, de exemplu, datorită necesităţii de a programa canalul de comunicaţie, a procedurilor de începere a comunicaţiei, a confirmării recepţiei etc. În cazul în care mesajele au lungime mare, timpul de iniţializare poate fi neglijat, folosindu-se modelul proporţional; de asemenea, dacă σ este mai mic sau de acelaşi ordin de mărime cu β, atunci σ poate fi neglijat chiar pentru mesaje relativ scurte. Pe de altă parte, dacă σ este relativ mare şi m mic, modelul liniar este apropiat de cel cu timp constant. Se remarcă aşadar că modelul liniar este o generalizare a modelelor precedente. Pentru majoritatea calculatoarelor actuale, relaţia între constantele din (3.1) este $\sigma \gg \beta$; ca ordin de mărime, valori ca $\sigma = 50\mu$s şi $\beta = 0.03\mu$s sunt caracteristice (amintim că valoarea timpului de initializare σ depinde şi de suportul software pentru comunicaţie).

În cazul modurilor de transmisie prin comutare de circuit sau wormhole, timpul de transmisie are forma

$$t_c(m) = \sigma + d\tau + m\beta. \tag{3.2}$$

În acest caz procesoarele nu sunt neapărat vecine, ci se află la distanţă d; termenul τ reflectă timpul necesar antetului să străbată legătura între doi vecini. Se observă

că factorul ce înmulţeşte dimensiunea m a mesajului nu depinde de lungimea căii de comunicaţie; cum τ este relativ mic, aceasta reflectă faptul că modul wormhole este, în general, mai eficient decât cel store and forward. Expresia (3.2) este valabilă în ipoteza că nici una dintre legăturile de pe calea dintre sursă şi destinaţie nu este ocupată.

Alte observaţii. Alegerea modelului pentru timpul de transmisie a unui mesaj este foarte importantă; există probleme de comunicaţie care se rezolvă optim, pe anumite arhitecturi, în mod destul de diferit în funcţie de modelul adoptat. În această lucrare vom folosi îndeosebi modelul liniar şi transmisia store and forward; de asemenea, modelul multiport va fi cel mai des adoptat. În ipoteza utilizării unui alt model, acesta va fi specificat în mod explicit acolo unde este cazul.

3.2 Operaţii de comunicaţie globală

În algoritmii de calcul numeric (în care datele se reprezintă sub formă matriceală) dar nu numai în aceştia, sunt deseori întâlnite operaţii de comunicare globală, în care sunt implicate mai mult de două procesoare. Unele dintre operaţii apar frecvent, de aceea au beneficiat de un dublu interes. În primul rând, ele au fost studiate intens pentru a se găsi algoritmi care să le execute rapid, pe fiecare dintre arhitecturile curente; amintim aici doar câteva din articolele de pionierat, care au propus algoritmi valabili şi astăzi, ale căror autori sunt Saad & Schultz [23], Johnsson & Ho [19], Stout & Wagar [25], Bertsekas *et al* [3]. În al doilea rând, aceste operaţii de comunicaţie au fost standardizate, în prezent toate bibliotecile de comunicaţie oferind implementări ale lor.

Cele mai uzuale operaţii de schimbare a datelor între procesoare sunt enumerate mai jos. Operaţiile sunt ilustrate în figura 3.4.

1. Transmiterea unui mesaj de la un procesor la altul (one-to-one). Includem această operaţie de comunicaţie în categoria celor globale pentru că, în general, procesoarele sursă şi destinaţie nu sunt vecine, deci mesajul transmis circulă pe la mai multe procesoare în drumul său.

2. Transmiterea aceluiaşi mesaj de către un procesor tuturor celorlalte procesoare, denumită *difuzare* (broadcast, one-to-all).

3. Transmiterea câte unui mesaj de către fiecare procesor tuturor celorlalte procesoare, denumită *difuzare generală* (all-to-all broadcast, multinode broadcast, gossiping[1]). Aşadar este vorba de câte o operaţie de difuzare efectuată de fiecare nod.

4. Transmiterea de către un procesor a câte unui mesaj pentru fiecare din celelalte procesoare, denumită *difuzare personalizată* sau *distribuţie* (distributing,

[1] Dacă vă pasionează engleza şi nu cunoaşteţi cuvântul, daţi fuga la dicţionar.

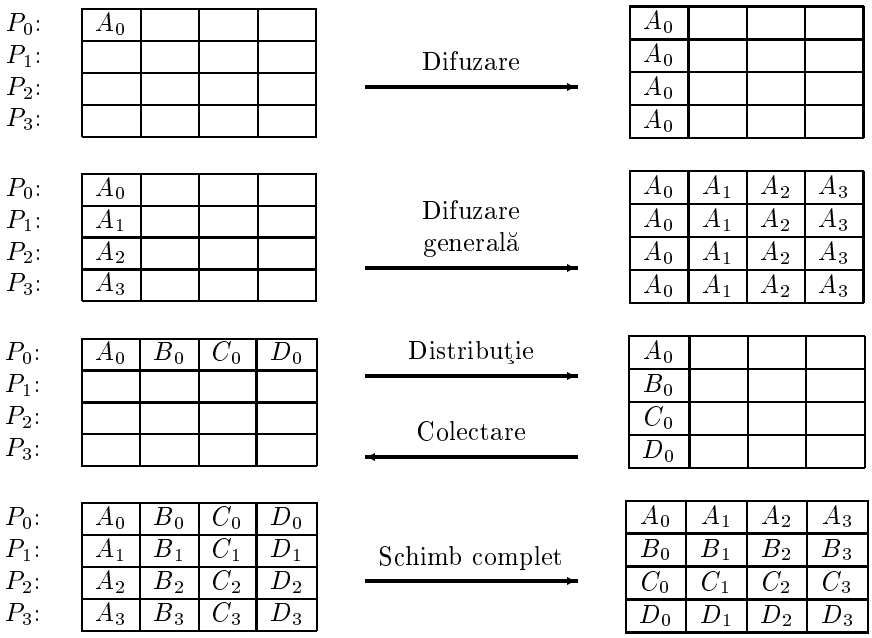


Figura 3.4: Operaţii de comunicaţie colectivă, exemplificate pentru $p = 4$ procesoare participante. Un procesor deţine datele de pe orizontala sa: în stânga cele dinainte de comunicaţie, în dreapta cele de după.

scattering, personalized one-to-all). Operaţia inversă, de transmitere de către fiecare procesor a câte unui mesaj pentru un anume procesor, denumită *colectare* (collecting, gathering), este similară difuzării personalizate şi nu trebuie tratată separat; inversând sensul căilor pe care se efectuează transmisia într-un algoritm de difuzare personalizată, se obţine un algoritm de colectare. Ambele operaţii pot apărea în cazul în care unul dintre procesoare are o poziţie privilegiată: de exemplu, el împarte seturi de date celorlalte procesoare, printr-o distribuţie, şi apoi culege rezultatele printr-o colectare.

5. Transmiterea de către fiecare procesor a câte unui mesaj pentru fiecare dintre celelalte procesoare, denumită *schimb complet*, *multidistribuţie* sau *transpunere* (multiscattering, personalized all-to-all, complete exchange, data transposition). Această operaţie consistă din difuzări personalizate efectuate de către fiecare procesor. Dacă fiecare procesor deţine iniţial o linie a unei matrice $p \times p$, el va avea în final coloana cu acelaşi indice, operaţia efectuată fiind aşadar transpunerea matricei respective.

O altă operaţie extrem de utilă ce poate fi inclusă în categoria comunicaţiilor

globale este **sincronizarea** (termenul englezesc echivalent este **barrier**). Ea este echivalentă cu o barieră în calea execuţiei programului unui procesor, care se opreşte la întâlnirea ei şi nu trece mai departe decât împreună cu toate celelalte procesoare. Este evident că, dacă este utilizată, primitiva **sincronizare** trebuie să apară în programul fiecărui procesor; după execuţia ei, toate procesoarelor vor începe, în paralel, execuţia instrucţiunii următoare. Operaţia de sincronizare poate apărea atât în calculatoarele MIMD cu memorie comună, cât şi în cele cu memorie distribuită. Multe calculatoare oferă suport hardware pentru execuţia ei, reducând semnificativ timpul necesar sincronizării. Să remarcăm însă că este vorba de timpul efectiv, scurs între momentul în care ultimul procesor apelează local primitiva de sincronizare şi momentul în care procesoarele încep execuţia primei instrucţiuni ulterioare sincronizării. În sfârşit, să mai observăm că în arhitecturile MIMD cu memorie distribuită nu este neapărat necesară o sincronizare globală a procesoarelor; sincronizarea se face implicit prin transmisia-recepţia de mesaje.

3.3 Comunicaţii globale

În această secţiune vor fi prezentaţi algoritmi pentru operaţiile de comunicaţie globală. Deşi în bibliotecile de comunicaţie actuale există rutine implementând toate operaţiile de comunicaţie globală, algoritmii descrişi în continuare sunt interesanţi deoarece oferă idei ce pot fi aplicate în algoritmii de calcul; în plus, studiul algoritmilor de comunicaţie permite o mai bună cunoaştere a modului în care se poate profita de o topologie dată. Topologiile considerate vor fi inelul, hipercubul şi torul de dimensiuni egale, deci $\sqrt{p} \times \sqrt{p}$; pentru grilă vom particulariza unii dintre algoritmii pentru tor; de asemenea, vom ţine seama că unii algoritmi de la inel se pot generaliza uşor pentru tor (deşi există algoritmi specifici mai performanţi). Să precizăm de la bun început că, de exemplu, orice algoritm valabil pentru inel va funcţiona şi pe un hipercub, deoarece un inel poate fi scufundat în hipercub; totuşi, pentru hipercub se pot găsi algoritmi specifici mult mai performanţi, care folosesc conectivitatea mai bună a acestuia.

Se va presupune că lungimea tuturor mesajelor este egală, anume m, ceea ce, în general, este cazul în practică, atunci când există mai multe mesaje de transmis, adică în difuzarea generală, distribuţie şi schimb complet. În algoritmii care implică situaţia specială a unui procesor, adică difuzarea şi distribuţia, se va presupune P_0 acest procesor (sau P_{00} pentru tor), ceea ce nu este o restrângere, arhitecturile considerate fiind simetrice, după cum am văzut în primul capitol. Modelul de comunicaţie va fi cel *store and forward*, pentru comutarea de circuit rezervând o secţiune specială.

Timpii de execuţie ai algoritmilor vor fi notaţi cu doi până la cinci indici. Primul este numărul de ordine al tipului de comunicaţie, după numerotarea din secţiunea precedentă. Al doilea reprezintă tipul arhitecturii: R pentru inel (ring), T pentru tor, G pentru grilă, H pentru hipercub. Al treilea reprezintă felul comunicaţiei: H pentru half duplex, F pentru full duplex; acest indice va fi omis dacă timpii de execuţie sunt aceiaşi pentru ambele modele. Al patrulea precizează numărul canalelor

de comunicaţie ce pot transmite simultan; valorile pot fi 1 pentru 1-port şi $*$ pentru multiport (acesta din urmă fiind cazul implicit). Al cincilea se referă la modelul timpului de comunicaţie: L pentru timp liniar, C pentru timp constant, P pentru timp proporţional; dacă lipseşte, se presupune modelul liniar. Primii doi indici vor fi prezenţi întotdeauna; ultimii trei, având valori diferite, vor putea apare în orice combinaţie. Deci $T_{2,H,F}$ reprezintă timpul de execuţie pentru difuzare, pe un hipercub, model full duplex, multiport, timp liniar; la fel şi $T_{2,H,F,C}$, dar pentru timp constant; nu apare nici o ambiguitate, chiar dacă indicele C se află în a patra poziţie, şi nu într-a cincea.

Un mesaj va fi notate M_a, unde a este procesorul destinaţie; indicele a poate lipsi, dacă destinaţia este evidentă din context. Dacă mesajul este împărţit în pachete, notăm M_a^i pachetul cu numărul de ordine i. Dacă este necesară precizarea procesorului sursă, atunci un pachet va fi notat cu $M_{s,a}^i$, unde s este procesorul sursă. În loc de s şi a pot apărea liste de procesoare, semnificând concatenarea unor mesaje provenind de la, sau cu destinaţia procesoarele din lista respectivă. Prin aceste notaţii lăsăm a se înţelege că un mesaj poate fi separat în mai multe submesaje, sau, la fel de bine, mai multe mesaje pot fi concatenate; aceste operaţii vor fi presupuse a se efectua în timp neglijabil.

3.3.1 Comunicaţie de la un procesor la altul

Deşi această operaţie poate părea banală, este totuşi interesantă tratarea ei pentru a pune în evidenţă mijloacele prin care se poate creşte viteza de comunicaţie a unui mesaj, pentru fiecare din modelele de timp expuse. Motivaţia includerii ei între operaţiile de comunicaţie globală, deşi doar două procesoare sunt beneficiarele efective ale comunicaţiei, se justifică prin aceea că multe alte procesoare, eventual toate celelalte, participă la transmisia între sursă şi destinaţie.

Presupunem că procesorul sursă P_s transmite un mesaj procesorului destinaţie P_a şi că aceasta este singura operaţie de comunicaţie efectuată pe ansamblul procesoarelor.

Comunicaţie punct-la-punct pe inel

Comunicaţie pe calea cea mai scurtă. În cazul inelului, distanţa între sursă şi destinaţie este $\delta = \min(|a-s|, p-|a-s|)$. Cea mai simplă metodă este de a transmite mesajul pe cea mai scurtă cale dintre P_s şi P_a, din procesor în procesor, după algoritmul următor.

ALGORITM 3.1 (transmiterea mesajului M de la P_s la P_a, pe calea cea mai scurtă, pe inel)

1. $\delta = \min(|a-s|, p-|a-s|)$
2. **dacă** $dist(\mathrm{id}, s) + dist(\mathrm{id}, a) = \delta$ **atunci**
    1. **dacă** $\mathrm{id} = s$

Figura 3.5: Poziţiile relative posibile pentru sursă şi destinaţie, pe un inel: (a) P_a la dreapta lui P_s (b) P_a la stânga lui P_s.

1. **dacă** $(a - s) \bmod p = \delta$ **atunci** **send**(M, dreapta)
   2. **altfel** **send**(M, stânga)
2. **altfel dacă** $(\mathrm{id} - s) \bmod p \leq \delta$ **atunci**
   1. **recv**(M, stânga)
   2. **dacă** $\mathrm{id} \neq a$ **atunci** **send**(M, dreapta)
3. **altfel dacă** $\delta < p/2$
   1. **recv**(M, dreapta)
   2. **dacă** $\mathrm{id} \neq a$ **atunci** **send**(M, stânga)

Aşadar, un procesor determină în primul rând, în instrucţiunile 1 şi 2, dacă se află sau nu pe calea cea mai scurtă între sursă şi destinaţie. Apoi, procesoarele de pe calea cea mai scurtă trebuie să determine în ce sens se efectuează transmisia; condiţia din instrucţiunea 2.2 arată când un astfel de procesor se găseşte la dreapta lui P_s; dacă $\mathrm{id} \geq s$, atunci $(\mathrm{id} - s) \bmod p = \mathrm{id} - s \leq \delta$; dacă $\mathrm{id} < s$, atunci $(\mathrm{id} - s) \bmod p = p - (s - \mathrm{id}) < \delta$, pentru că $s - \mathrm{id} > p - \delta$; aceste două cazuri sunt cele din figura 3.5a; în celelalte două cazuri, P_a se găseşte în stânga lui P_s, deci şi procesoarele de pe calea cea mai scurtă sunt în stânga. Să mai observăm că atunci când ambele căi între P_s şi P_a au aceeaşi lungime, atunci este preferat convenţional sensul de la stânga la dreapta. În cazul în care sensul de transmisie este de la stânga la dreapta, comunicaţia este iniţiată de P_s în instrucţiunea 2.1.1. Celelalte procesoare participante realizează retransmiterea mesajului, în instrucţiunile 2.2.1 şi 2.2.2; evident, procesorul destinaţie doar recepţionează. Circulaţia în sensul celălalt este realizată de 2.1.2, 2.3.1 şi 2.3.2 (testul din 2.3 interzice procesoarelor să transmită de la dreapta spre stânga în cazul în care există două căi de lungime minimă, ceea ce e posibil doar când p este par şi $\delta = p/2$). Poate că am detaliat excesiv acest algoritm, însă am făcut-o pentru a arăta drumul de la simpla enunţare a ideii, până la o implementare; de aici înainte vor fi prezentaţi algoritmi doar în cazurile mai complicate.

Timpul necesar execuţiei algoritmului 3.1 este:

$$T_{1,R,C} = \delta; \qquad T_{1,R,P} = \delta m\beta; \qquad T_{1,R,L} = \delta(\sigma + m\beta).$$

Pentru modelul timp constant nu se poate face mai bine, mesajul trebuind să parcurgă oricum δ canale. Pentru celelalte două modele, două idei conduc la creşterea vitezei comunicaţiei.

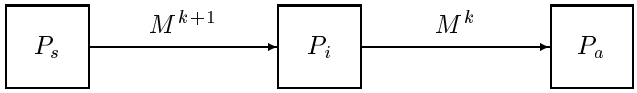


Figura 3.6: Transmisie pipeline între procesoarele P_s şi P_a, prin intermediul procesorului P_i.

Utilizarea mai multor căi. În primul rând, mesajul se poate împărţi în două şi transmite pe cele două căi distincte dintre P_s şi P_a, una de lungime δ, cealaltă de lungime $p - \delta$; se străbate deci inelul în cele două sensuri, pornind de la P_s. Dacă mesajul se împarte într-un pachet de dimensiune γm trimis pe calea cea mai scurtă şi unul de lungime $(1-\gamma)m$ trimis pe calea mai lungă, cu $\gamma \in (0,1)$, timpul optim este

$$T'_{1,R,L} = \delta(\sigma + \gamma m\beta),$$

atunci când ambele pachete ajung simultan la P_a. Se obţine

$$\gamma = \frac{p-\delta}{p} + \frac{p-2\delta}{p}\frac{\sigma}{m\beta}.$$

Pentru modelul timp proporţional se ia $\sigma = 0$ în relaţiile de mai sus.

Transmisie pipeline. În al doilea rând, pentru a fi folosite cât mai multe canale de comunicaţie în paralel (spre deosebire de simpla transmitere a mesajului, în care o singură legătură este ocupată la un moment dat), un mesaj transmis pe o cale poate fi împărţit în mai multe pachete (submesaje), care sunt transmise succesiv, unul câte unul. Un exemplu este prezentat în figura 3.6, în care între procesoarele P_s şi P_a se află un singur alt procesor P_i; după ce P_i primeşte primul pachet din mesaj, va trimite spre P_a acest pachet, în paralel cu recepţia celui de-al doilea pachet de la P_s. De fapt se efectuează o transmisie *pipeline* a mesajului. În cazul unor mesaje foarte lungi, timpul pentru transmiterea mesajului între P_s şi P_a poate fi aproximat cu cel din cazul în care procesoarele ar fi fost vecine; diagrama ocupării legăturilor dintre procesoare din figura 3.7 arată că, în transmisia pipeline, legăturile sunt aproape permanent ocupate, mai puţin unii mici timpi morţi iniţiali şi/sau finali. Se poate observa că aceasta este o abordare software a modului de comunicare wormhole, cu diferenţa, totuşi, că acolo pachetele nu erau memorate în nodurile intermediare decât dacă era absolut necesar.

Numărul optim de pachete. Aplicăm metoda pipeline pentru transmisia mesajului de la sursă la destinaţie, pe calea cea mai scurtă; considerăm mesajul împărţit în ν pachete notate M^0, M^1, ..., $M^{\nu-1}$, fiecare de lungime m/ν; presupunem, pentru simplitate, că ν divide m. Se constată că primul pachet ajunge la P_a după δ paşi, iar ultimul după $\nu - 1 + \delta$ paşi; deci timpul total este, pentru modelul timp proporţional:

$$T''_{1,R,P} = (\nu - 1 + \delta)\frac{m}{\nu}\beta = m\beta + (j-1)\frac{m}{\nu}\beta.$$

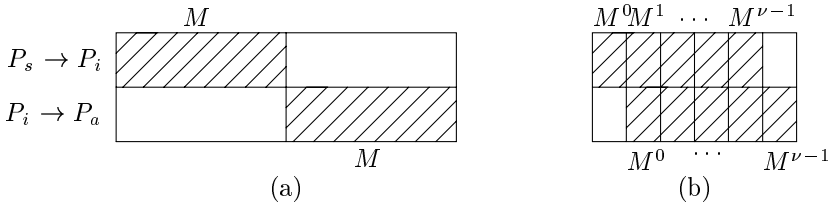


Figura 3.7: Ocuparea legăturilor pentru comunicarea din figura anterioară, în cazurile: (a) transmisie integrală a mesajului; (b) transmisie pipeline. Dreptunghiurile haşurate reprezintă timpii necesari transmiterii unui pachet; zonele albe reprezintă timpii morţi.

Pentru a minimiza acest timp se observă că ν trebuie să fie cât mai mare; deci, $\nu = m$, adică fiecare pachet este format dintr-un singur octet (element de mesaj).

Pentru modelul timp liniar, timpul de comunicaţie este

$$T''_{1,R,L} = (\nu - 1 + \delta)(\sigma + \frac{m}{\nu}\beta).$$

Numărul de pachete care minimizează acest timp este

$$\nu_{\text{opt}} = \sqrt{\frac{m\beta(\delta-1)}{\sigma}},$$

iar timpul minim corespunzător este

$$T''_{1,R,L\min} = \left(\sqrt{m\beta} + \sqrt{(\delta-1)\sigma}\right)^2. \tag{3.3}$$

Acest minim se poate încadra astfel:

$$(\delta-1)\sigma + m\beta < T''_{1,R,L\min} \leq 2(\delta-1)\sigma + 2m\beta$$

şi el se apropie de termenul din stânga pe măsură ce m creşte.

Pe de altă parte, pentru a se atinge optimul, este necesar să fie îndeplinită condiţia $1 \leq \nu_{\text{opt}} \leq m$, ceea ce se întâmplă dacă:

$$\frac{\sigma}{m} \leq (\delta-1)\beta \leq m\sigma.$$

Pentru valori suficient de mari ale lungimii mesajului m, această condiţie este satisfăcută. Parametrul σ variază, pentru diverse calculatoare, de la valori de ordinul lui β până la valori de sute de ori mai mari; în plus, distanţa între sursă şi destinaţie δ, fiind mai mică decât diametrul grafului reprezentat de topologia reţelei de interconectare, are o valoare de ordinul zecilor, să spunem; deci m trebuie să fie de cel puţin câteva zeci, ceea ce este cazul aproape întotdeauna; practic, nu se folosesc mesaje prea scurte

datorită ineficienţei comunicaţiei; oricum, se poate concluziona că tehnica pipeline nu se poate aplica eficient mesajelor scurte.

Pipeline pe mai multe căi. Metoda pipeline se poate aplica tuturor algoritmilor de comunicaţie de la un procesor la altul, combinată cu folosirea tuturor căilor de comunicaţie *arc-disjuncte* (care nu au nici un arc comun) între sursă şi destinaţie; căile pot avea noduri comune, deoarece lucrăm în modelul multiport, deci se poate transmite permanent în mod simultan prin acele noduri. De exemplu, în cazul inelului, transmiţând pipeline în ambele sensuri, de la sursă spre destinaţie, timpul de transmisie are o expresie similară cu (3.3), anume

$$T'''_{1,R,L} = \left(\sqrt{\frac{m}{2}}\beta + \sqrt{(p-\delta-1)\sigma}\right)^2. \tag{3.4}$$

Acest timp este de aproape două ori mai mic decât $T''_{1,R,L\min}$, atunci când m este suficient de mare. El poate fi încă micşorat dacă se folosesc pachete de dimensiuni diferite pe cele două căi, dar nu cu mult. Expresia (3.4) se poate imediat generaliza pentru un graf oarecare.

PROPOZIŢIA 3.1 *Presupunem că, pe un graf oarecare, există c căi arc-disjuncte între sursă şi destinaţie, cea mai lungă dintre ele având lungime h. Atunci timpul de comunicaţie a unui mesaj de lungime m, folosind toate aceste căi în regim pipeline, este*

$$T_1 = \left(\sqrt{\frac{m}{c}}\beta + \sqrt{(h-1)\sigma}\right)^2. \tag{3.5}$$

Se poate observa că sunt valabile relaţiile $c \leq \Delta$, unde Δ este gradul grafului, i.e. numărul de canale al unui procesor, şi $h \geq \delta$.

Concluzii. Pentru modelul timp constant cea mai eficientă transmitere a unui mesaj are loc pe calea cea mai scurtă între sursă şi destinaţie. Pentru modelul timp liniar (şi m suficient de mare), trebuie găsite cât mai multe căi arc-disjuncte între sursă şi destinaţie, lungimea lor fiind mai puţin importantă; transmisia eficientă pe aceste căi are loc în modul pipeline. În continuare, pentru tor şi hipercub, nu vom mai preciza timpii obţinuţi în urma aplicării tehnicii pipeline, ei rezultând direct din formula (3.5), ci ne vom rezuma la prezentarea căilor pe care se poate face comunicaţia.

Mai observăm că timpul (3.5) este acelaşi pentru modelele half duplex şi full duplex, canalele fiind folosite întotdeauna într-un singur sens.

Comunicaţie punct-la-punct pe tor

Pe un tor există cel mult patru căi arc-disjuncte între sursă şi destinaţie, dat fiind gradul acestui graf. În această secţiune vom presupune că $\sqrt{p}$ este par, deci diametrul torului este chiar $D = \sqrt{p}$. Să considerăm întâi cazul în care procesoarele P_s şi P_a se află pe linii şi coloane diferite. O construcţie imediată este cea din figura 3.8a, în care cele patru căi sunt construite folosind pentru fiecare o singură linie şi o singură

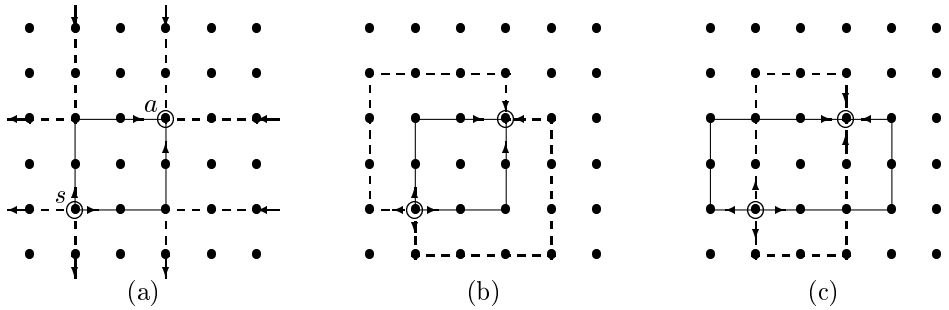


Figura 3.8: Trei variante de construcţie a patru căi arc-disjuncte între două procesoare aflate pe linii şi coloane diferite, într-un tor.

coloană; pentru prima se merge întâi la dreapta, pe linia lui P_s, până se întâlneşte coloana lui P_a, apoi în sus; pe a doua, întâi în sus, apoi la dreapta; celelalte două sunt construite folosind direcţiile stânga şi jos, în cele două ordini posibile. Se observă că două căi, cele reprezentate cu linie continuă, au lungimea egală cu distanţa δ dintre noduri. Celelalte, reprezentate cu linie întreruptă, au în general lungimea mai mare, anume $2\sqrt{p} - \delta$. Aceste patru căi pot fi privite ca o generalizare a celor două căi posibile pe un inel.

Această variantă are în avantajul său simplitatea, dar e posibil ca lungimea căilor să fie prea mare; de asemenea, nu este posibilă o generalizare pentru grilă. Se pot imagina două alte soluţii, care încearcă minimizarea lungimii maxime a celor patru căi. În figura 3.8b, celor două căi de lungime δ li se adaugă două de lungime $\delta + 4$, construite în jurul dreptunghiului format de primele două căi. Se poate demonstra optimalitea celei de-a treia variante, cea din figura 3.8c, în care cele patru căi au toate lungime $\delta + 2$; ele sunt construite folosind una din cele două căi cele mai scurte între fiecare dintre vecinii din aceeaşi poziţie ai lui P_s, respectiv P_a (vecinul din dreapta al lui P_s cu cel din dreapta al lui P_a, etc.); toate aceste căi au lungime δ, la care se adaugă cele două arce între P_s şi vecin, respectiv între P_a şi vecinul corespunzător.

Când cele două procesoare, P_s şi P_a se află pe aceeaşi linie sau aceeaşi coloană, în oricare dintre variantele din figura 3.8, două căi se suprapun, deci trebuie găsită o altă soluţie. O posibilitate este cea din figura 3.9, în care fiecare cale are lungime $\delta + 4$; căile au fost desenate câte două, pentru mai multă claritate; închipuiţi-vă că torul este ceva mai mare, altfel există o variantă mai bună, dar particulară, P_a fiind la aceeaşi distanţă de P_s pe orizontală, şi prin stânga şi prin dreapta.

Şi această variantă cade atunci când procesoarele sunt vecine. Vă rămâne să găsiţi d-voastră soluţia (problema 3.3.4).

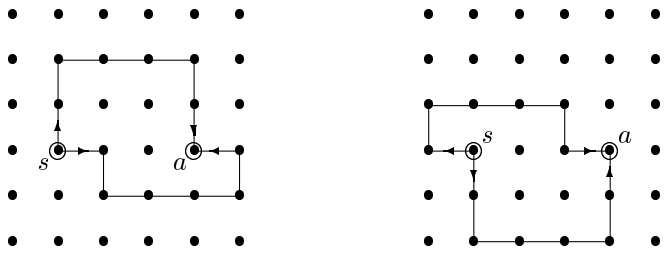


Figura 3.9: Patru căi arc-disjuncte între procesoare aflate pe aceeşi linie.

Comunicaţie punct-la-punct pe hipercub

Pe un hipercub, distanţa dintre P_s şi P_a este egală cu numărul de biţi de 1 din $s \oplus d$, adică distanţa Hamming dintre s şi a, pe care o notăm cu $H(s,a) = \delta$. O cale de lungime minimă între P_s şi P_a se obţine complementând unul câte unul biţii diferiţi din reprezentările binare ale adreselor s şi a. Numim *principală* calea obţinută prin complementarea biţilor diferiţi în ordine *de la dreapta la stânga* (de la cel mai puţin la cel mai semnificativ).

Pentru simplitate, descriem comunicarea între procesoarele 0 şi $2^\delta - 1 = 0 \ldots 01 \ldots 11$ (număr ce are 1 pe ultimele δ poziţii). Generalizarea la adrese s şi a oarecare este imediată, vezi problema **P**3.3.5.

Comunicaţie între colţuri opuse. Vom începe cu cazul particular al comunicării între două colţuri opuse ale hipercubului, deci $s = 0$ şi $a = 2^d - 1$. În acest caz, e.g. pentru $\mathcal{H}_3$, calea principală este formată din nodurile 000, 001, 011, 111, şi are lungimea $d = 3$. Pentru obţinerea altor $d-1$ căi, complementăm biţii adresei $s = 0$ tot de la dreapta la stânga, dar începând cu biţii din poziţiile $1, \ldots, d-1$; după bitul $d-1$, se continuă ciclic, adică se complementează bitul 0. De exemplu, pentru $\mathcal{H}_3$, cele două noi căi sunt 000, 010, 110, 111 şi 000, 100, 101, 111, după cum se vede şi în figura 3.10a. Aceste căi pot fi obţinute din adresele nodurilor de pe calea principală, prin rotiri (deplasări ciclice) la stânga cu $1, \ldots, d-1$ poziţii. Numerotăm căile după poziţia primului bit care este modificat (deci calea principală are numărul 0). Introducem notaţia $\mathrm{Ro}(c,l)$, care semnifică numărul obţinut prin rotaţia numărului binar c cu l biţi la stânga. Aşadar, dacă c este adresa unui nod de pe calea principală, atunci $\mathrm{Ro}(c,l)$ este adresa nodului de pe calea l, aflat la aceeaşi distanţă de nodul 0 ca şi c.

PROPOZIŢIA 3.2 *Cele d căi astfel construite între nodurile 0 şi $2^d - 1$ sunt arc-disjuncte.*

Demonstraţie. Notăm cu C_k mulţimea nodurilor aflate la distanţă k de nodul 0 (şi deci având k biţi de 1). Prin construcţie, fiecare cale are exact un arc între C_k şi

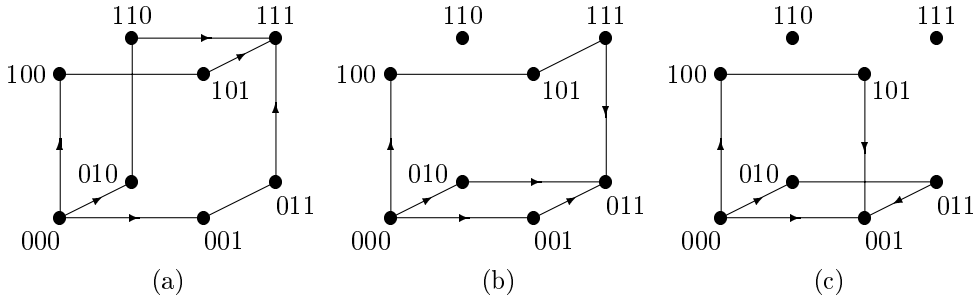


Figura 3.10: Cele trei căi arc-disjuncte într-un hipercub de dimensiune 3, între nodurile: (a) 000 şi 111; (b) 000 şi 011; (c) 000 şi 001.

C_{k+1}; aceste d arce sunt în dimensiuni diferite, mai precis arcul de pe calea l se află în dimensiunea $(k+l)$ mod d. Deci căile sunt arc-disjuncte. ■

Cazul general. Trecem acum la cazul în care $\delta < d$. Calea principală între 0 şi $2^\delta - 1$ are lungime δ. Rotind acum numai grupul primilor δ biţi ale adreselor nodurilor de pe calea principală, obţinem δ căi arc-disjuncte de lungime δ; sunt, de fapt, căile între colţurile opuse ale unui hipercub de dimensiune δ, construite toate cu arce din acel hipercub. Mai multe căi de lungime δ nu se pot obţine, deoarece modificând biţi din poziţii mai mari decât $\delta - 1$ ne îndepărtăm de nodul $2^\delta - 1$. Scopul nostru este însă să construim d căi, maximul ce se poate spera într-un hipercub, acesta fiind gradul grafului. Celelalte $d-\delta$ căi se construiesc astfel; în calea l, primul arc se obţine prin negarea bitului l, deci se află în dimensiunea l; este deci arcul între 0 şi 2^l; apoi se construieşte calea principală de la nodul 2^l la nodul $2^\delta - 1$. În figura 3.10b,c sunt ilustrate astfel de construcţii pentru noduri aflate la distanţă 2, respectiv 1, în $\mathcal{H}_3$.

PROPOZIŢIA 3.3 *Cele d căi între nodurile 0 şi $2^\delta - 1$ sunt arc-disjuncte.*

Demonstraţie. Primele δ căi sunt arc-disjuncte conform Propoziţiei 3.2. Celelalte—în afara primului şi ultimului arc, aflate pe dimensiunea l pentru calea l—unesc noduri ale căror adrese au bitul din poziţia l egal cu 1, deci nu se pot suprapune cu nici una dintre celelalte căi, care conţin noduri având acest bit egal cu 0. ■

Putem sintetiza toate aceste consideraţii într-un algoritm. Notăm cu y numărul format din ultimii δ biţi din adresa procesorului pentru care scriem algoritmul (adică id), şi cu x pe cel format din primii $d - \delta$ biţi; cu u_k notăm numărul reprezentat pe δ biţi, având ultimii k biţi egali cu 1, iar pe ceilalţi 0 (de exemplu, dacă $\delta = 4$, $u_3 = 0111$). În întreg algoritmul, pachetul M^i, de lungime m/d, va circula pe calea i, folosind numerotarea descrisă mai sus. Algoritmul este valabil şi pentru cazul $\delta = d$.

ALGORITM 3.2 (comunicaţie de la P_0 la $P_{2^\delta-1}$ a mesajului M, pe d căi arc-disjuncte, în hipercubul $\mathcal{H}_d$)

1. $b \leftarrow$ numărul de biţi de 1 din y
2. $e \leftarrow$ numărul de biţi de 1 din x
3. **dacă** id $= 0$ **atunci** {procesorul sursă}
   1. **pentru** $i = 0 : d-1$, **în paralel** {emite pe toate căile}
      1. **send**(M^i, i)
4. **altfel dacă** id $= 2^\delta - 1$ **atunci** {procesorul destinaţie}
   1. **pentru** $i = 0 : \delta - 1$, **în paralel** {primele δ căi}
      1. **recv**$(M^i, (i-1) \bmod \delta)$
   2. **pentru** $i = \delta : d-1$, **în paralel** {celelalte $d - \delta$ căi}
      1. **recv**(M^i, i)
5. **altfel dacă** $e = 0$ **atunci** {proc. poate fi într-una din primele δ căi}
   1. **dacă** $y = \mathrm{Ro}(u_k, i)$ **atunci** {implicit $0 < k < \delta$, $0 \le i < \delta$}
      1. **recv**$(M^i, (k+i-1) \bmod \delta)$
      2. **send**$(M^i, (k+i) \bmod \delta)$
6. **altfel dacă** $e = 1$ **atunci** {proc. poate fi într-una din ultimele $d - \delta$ căi}
   1. $i \leftarrow$ poziţia bitului de 1 din x, în id
   2. **dacă** $y = u_k$ **atunci**
      1. **dacă** $k = 0$ **atunci** **recv**(M^i, i) {de la P_0}
      2. **altfel** **recv**$(M^i, k-1)$
      3. **dacă** $k = \delta$ **atunci** **send**(M^i, i) {spre $P_{2^\delta - 1}$}
      4. **altfel** **send**(M^i, k)

În loc de suplimentarea explicaţiilor oferite în comentarii, vă propunem ca exerciţiu aplicarea algoritmului pentru fiecare din nodurile hipercubului de dimensiune 3 pentru comunicaţia între nodurile 000 şi 011. Se obţin într-adevăr căile din figura 3.10b ?

3.3.2 Difuzare

Dacă în comunicarea între două procesoare nu am avut nevoie de a preciza sensul în care se transmit mesajele pe o legătură, acesta fiind evident, acum şi mai departe sensul devine important. De aceea e necesar să considerăm grafurile orientate, între fiecare două noduri vecine existând două arce, câte unul în fiecare sens. Vom da în continuare câteva noi definiţii referitoare la arbori.

Arbori. Reamintim că un arbore este un graf conex fără bucle. Într-un arbore, nodurile unite de un arc se numesc *tată* şi *fiu*, sensul arcului fiind de la tată spre fiu; un tată poate avea mai mulţi fii; dacă are cel mult doi, arborele se numeşte *binar*; în schimb, un fiu are întotdeauna un singur tată. Nodul care nu are tată se numeşte *rădăcină*; nodurile care nu au fii se numesc *frunze*. *Înălţimea* (numită şi *adâncime*) unui arbore este distanţa cea mai mare între rădăcină şi o frunză. *Nivelul* unui nod este distanţa dintre acesta şi rădăcină; aşadar, înălţimea grafului este nivelul maxim al unui nod; nivelul unui arc va fi considerat egal cu nivelul nodului tată.

Un *arbore de acoperire* (spanning tree) al unui graf X este un arbore care conţine toate nodurile din X şi arce din X (este arbore şi graf parţial al lui X). O *familie* de

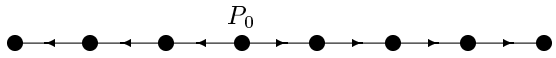


Figura 3.11: Un arbore de acoperire de înălţime $\lfloor p/2 \rfloor$, pe un inel.

arbori de acoperire este formată din mai mulţi arbori de acoperire cu aceeaşi rădăcină (care pot fi arc-disjuncţi sau nu).

Difuzare pe arbori de acoperire. Legătura dintre difuzare şi arborii de acoperire este imediată. Procesorul care începe difuzarea se află în rădăcină. El îşi transmite mesajul către fii săi, care transmit la rândul lor fiilor lor, ş.a.m.d până când mesajul ajunge la toate frunzele; evident, fiecare nod păstrează pentru sine o copie a mesajului. Pe o familie de arbori de acoperire, mesajul este împărţit în mai multe pachete, fiecare difuzat pe unul dintre arbori. Pe un singur arbore de acoperire se poate utiliza întotdeauna tehnica pipeline. Cum, la un moment dat, este posibil ca toate arcele unui astfel de arbore să fie ocupate, se poate face pipeline eficient pe o familie de arbori de acoperire numai dacă arborii sunt arc-disjuncţi. Pentru timpul de difuzare este deci valabil următorul rezultat analog Propoziţiei 3.1.

PROPOZIŢIA 3.4 *Utilizând transmisia pipeline cu dimensiune optimă a pachetelor pentru difuzare pe o familie de c arbori de acoperire arc-disjuncţi, având înălţimea maximă h, se obţine un timp de difuzare care respectă formula (3.5).*

Atragem atenţia că este vorba despre modelul full duplex, deoarece ambele arce orientate corespunzând unui canal de comunicaţie pot aparţine unor arbori (distincţi) din familie. Fiindcă arborii de acoperire din familie au aceeaşi rădăcină (nodul care difuzează), avem $c \leq \Delta$. În plus, înălţimea arborilor este limitată inferior de diametrul grafului, i.e. $h \geq D$. Deci, în cel mai bun caz, timpul de difuzare este egal cu timpul de comunicaţie punct-la-punct între două noduri situate la distanţă maximă (D) în graf. Aşadar, în cele ce urmează vom evidenţia numărul de arbori de acoperire şi înălţimea lor maximă, timpul de difuzare rezultând din (3.5).

Difuzare pe inel

Un arbore de înălţime minimă. Presupunem fără pierderea generalităţii că P_0 este procesorul care difuzează. Un arbore de acoperire evident util pentru difuzare este cel din figura 3.11, având înălţimea egală cu diametrul inelului, adică $\lfloor p/2 \rfloor$. Algoritmul de difuzare este intuitiv: P_0 trimite pe ambele canale mesajul, vecinii săi îl retransmit, etc. Orice alt arbore de acoperire conduce la un timp mai mare de difuzare, deoarece înălţimea arborelui este mai mare decât diametrul inelului.

Doi arbori arc-disjuncţi. Putem încerca construirea unei familii formată din doi arbori de acoperire arc-disjuncţi; mai mult de atât nu se poate, deoarece din rădăcină pornesc doar două arce. Singura soluţie este cea din figura 3.12, în care primul arbore conţine $p-1$ arce orientate spre stânga, iar al doilea $p-1$ arce orientate spre dreapta;

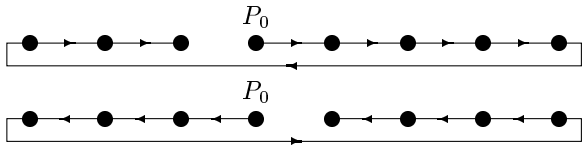


Figura 3.12: O familie formată din doi arbori de acoperire arc-disjuncţi de înălţime $p-1$, pe un inel.

înăltimea arborilor este $p-1$; singurele arce nefolosite din inel sunt cele orientate spre P_0. Se împarte aşadar mesajul M în două pachete de dimensiuni egale, trimise de P_0 în cele două sensuri, şi retransmise de-a lungul întregului inel, conform următorului algoritm:

ALGORITM 3.3 (P_0 difuzează mesajul M pe inel, pe doi arbori de acoperire arc-disjuncţi, model multiport, full duplex)

1. **dacă** id $= 0$ **atunci**
   1. **în paralel send**(M^0, dreapta), **send**(M^1, stânga)
2. **altfel**, **în paralel**
   1. **recv**(M^0, stânga), **dacă** id $\neq p-1$ **atunci send**(M^0, dreapta)
   2. **recv**(M^1, dreapta), **dacă** id $\neq 1$ **atunci send**(M^1, stânga)

Aşadar, pachetul M^0 este trimis pe primul arbore de acoperire din figura 3.12, iar M^1 pe al doilea. Dacă urmărim puţin în timp "traiectoria" celor două pachete, observăm că instrucţiunile 2.1 şi 2.2 nu se execută în general în paralel, ci numai în cazul unui singur procesor (cu id $= p/2$, când p este par); acesta nu e un motiv pentru a nu le programa în paralel; dimpotrivă, nefiind nici o corelaţie între operaţiile din cele două instrucţiuni, algoritmul 3.3 este general şi simplu. Mai mult, algoritmul se poate generaliza pentru un cazul transmisiei pipeline. Fiecare procesor va executa de ν ori instrucţiunile 1.1 (sursa), respectiv 2.1 şi 2.2 (celelalte procesoare), unde ν este numărul de pachete.

Pentru modelul half duplex, timpul de difuzare este de două ori mai mare decât pentru full duplex, deci nu se obţine nici un câştig prin utilizarea a doi arbori de acoperire.

Translaţia unui arbore. Ce se întâmplă dacă nodul care face difuzarea nu este P_0, ci un P_s oarecare ? Este clar că arborii de acoperire se construiesc în acelaşi mod ca pentru P_0. Vom numi *translaţie* operaţia prin care dintr-un arbore de acoperire $\mathcal{A}(0)$ cu rădăcina în P_0 se obţine un altul, $\mathcal{A}(s)$, cu aceeaşi formă, dar cu rădăcina în P_s. Pentru inel, un nod oarecare $x \in \mathcal{A}(0)$ se transformă simplu în $x' \in \mathcal{A}(s)$ prin relaţia $t(x) = (x+s) \bmod p$; un arc unind nodurile x şi y în $\mathcal{A}(0)$, se transformă în arcul ce uneşte nodurile $t(x)$ şi $t(y)$; de acum înainte vom vorbi doar despre transformările aplicate nodurilor, rămânând implicit că un arc ce uneşte două noduri se transformă în arcul ce uneşte transformatele nodurilor.

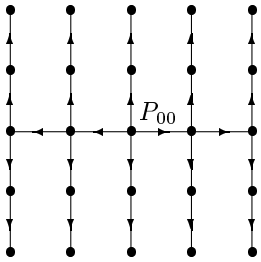


Figura 3.13: Un arbore de acoperire pentru tor.

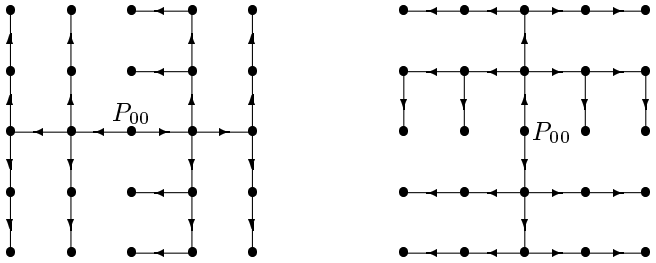


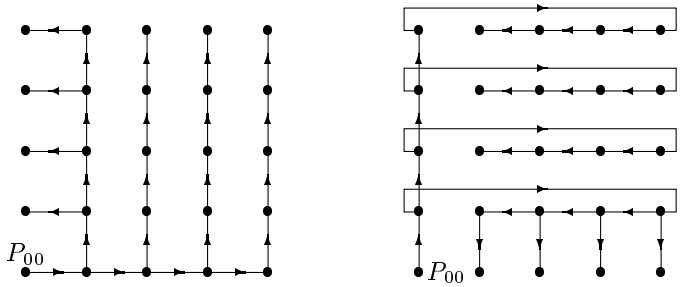
Figura 3.14: O familie formată din doi arbori de acoperire arc-disjuncţi, pe un tor.

Difuzare pe tor

Vom trece repede peste tor; P_{00} este procesorul care difuzează. În figura 3.13 este prezentat cel mai evident arbore de acoperire, de înălţime egală cu diametrul torului. Distanţa, pe acest arbore, între P_{00} şi orice nod, este cea mai mică posibil (egală cu distanţa pe tor între cele două noduri). Algoritmul de difuzare pe acest arbore este prezentat în rezolvarea problemei 3.3.12.

Cum el conţine toate cele patru arce care pornesc din rădăcină, este de presupus că se poate mai bine. În figura 3.14 sunt prezentaţi doi arbori de acoperire arc-disjuncţi; privind cu puţină atenţie, se observă că cel de-al doilea este obţinut din primul printr-o rotaţie cu 90 de grade în sens trigonometric, în jurul procesorului P_{00}, pe care-l identificăm cu punctul de coordonate (0,0); o astfel de rotaţie transformă punctul (x, y) în $(-y, x)$ (pentru verificare, consideraţi un exemplu, să zicem (2,1)); evident, aşa cum am mai spus-o, toate coordonatele trebuie gândite modulo $\sqrt{p}$. Înălţimea arborilor este egală cu diametrul torului. Construcţia decurge aproape imediat din cea din figura 3.13; arcele de pe verticala nodului P_{00} sunt singurele stânjenitoare: după rotaţie, s-ar suprapune cu cele de pe orizontala lui P_{00}; ele sunt eliminate şi înlocuite cu arce orizontale.

Cum ideea rotaţiei pare benefică şi este şi deosebit de simplă, să o aplicăm pentru



(a) (b)

Figura 3.15: (a) Arbore de acoperire pe tor, din care se poate genera prin rotaţii o familie de patru arbori arc-disjuncţi; (b) arbore de acoperire obţinut prin rotaţia cu 90 de grade în sens trigonometric a arborelui de la (a).

a construi patru arbori de acoperire arc disjuncţi. Este evident că fiecare va folosi un singur arc plecând din P_{00}. Primul arbore este cel construit în figura 3.15a, gândind aproximativ după cum urmează; nodurile de pe orizontala lui P_{00} sunt toate unite prin arce mergând spre dreapta; din fiecare se construiesc arce în sus, pe toată verticala; în acest fel se acoperă toate nodurile torului, mai puţin cele de pe verticală lui P_{00}; se vede imediat că, între arborii obţinuţi prin rotaţie, nu există conflicte de utilizare a arcelor. În fine, nodurile de pe verticala lui P_{00} sunt legate de cele de pe verticala lui P_{01}, deci de la dreapta spre stânga, din nou fără conflicte la rotaţie. Pentru edificare, prezentăm şi arborele obţinut prin rotaţie cu 90 de grade în sens trigonometric, în figura 3.15b. Înălţimea acestor arbori este $2(\sqrt{p}-1)$, deci de aproximativ două ori mai mare decât a arborilor din figura 3.14.

Trebuie menţionat că se pot construi patru arbori de acoperire arc disjuncţi de înălţime $D+1$, unde $D = 2\lfloor\sqrt{p}/2\rfloor$ este diametrul torului; acesta este optimul, dar construcţia este relativ complicată.

Pentru tor, translaţia se face cu aceeaşi operaţie ca pentru inel, însă aplicată pentru ambele coordonate ale adresei unui procesor.

Difuzare pe hipercub

Arbore de acoperire binomial. În figura 3.16a este prezentat un arbore de acoperire binomial; construcţia sa poate fi descrisă în mai multe feluri; de exemplu, în mod recursiv: arcul între nodurile 0 şi 1 face parte din arbore; se continuă analog construcţia în cele două hipercuburi de dimensiune $d-1$, obţinute după eliminarea ultimului bit al adreselor nodurilor, adică $\bar{\mathcal{H}}_d^{d-1}(0)$ şi $\bar{\mathcal{H}}_d^{d-1}(1)$; o observaţia imediată este că fiecare subarbore al unui arbore binomial este la rândul său un arbore binomial. Sau, dacă i este poziţia pe care se află cel mai semnificativ bit de 1 (-1, pentru nodul 0), atunci fiii acestui nod sunt vecinii în hipercub pe dimensiunile $i+1, \ldots,$

$d-1$. Arborele se numeşte binomial deoarece numărul nodurilor pe nivelul k este $\binom{d}{k}$ (totuşi, nu e singurul arbore de acoperire cu această proprietate).

O formă a algoritmului de difuzare pe acest arbore este sugerată într-un mod mai intuitiv în figura 3.17. La fiecare pas, se transmite pe o singură dimensiune a hipercubului, de la toate procesoarele aflate în posesia mesajului M, către vecinii din acea dimensiune; fiecare procesor foloseşte un singur canal la un moment dat. După k paşi, difuzarea într-un hipercub k-dimensional conţinând P_0 este încheiată. Algoritmul este formulat mai riguros în continuare:

ALGORITM 3.4 (P_0 difuzează mesajul M pe hipercub, după un arbore binomial, model 1-port)

1. $l \leftarrow$ poziţia celui mai semnificativ bit de 1 din id (-1 pentru P_0)
2. **dacă** id $\neq 0$ **atunci**
   1. **recv**(M, l) {recepţionează de la tată}
3. **pentru** $i = l + 1 : d - 1$
   1. **send**(M, i) {trimite tuturor fiilor}

Cu acest algoritm, procesoarele nu primesc mesajul în cel mai scurt timp fiecare. Aceasta se poate realiza în modelul multiport, adăugând o instrucţiune **în paralel** instrucţiunii 3; astfel, fiecare procesor va expedia imediat mai departe mesajul tuturor fiilor. Timpul global de difuzare nu se va modifica însă, el fiind cel după care procesorul $2^d - 1$ primeşte mesajul.

Familie de arbori binomiali. Deoarece în algoritmul 3.4 la fiecare pas se comunică pe o singură dimensiune, o idee de îmbunătăţire a timpului de comunicare este de a folosi toate dimensiunile simultan. Se împarte mesajul în d pachete M^0, M^1, ..., M^{d-1}; la pasul 0, procesorul P_0 trimite câte un pachet fiecărui vecin, i.e. pachetul M^i vecinului din dimensiunea i; la pasul k, pachetul M^i va fi trimis pe direcţia $(i + k) \bmod d$, de către toate procesoarele care îl deţin. Funcţionarea algoritmului este ilustrată în figura 3.18.

Aceasta revine la a utiliza o familie de arbori de acoperire construiţi din cel din figura 3.16a, pe care-l vom nota $\mathcal{A}_0(0)$. Ceilalţi $d-1$ arbori se construiesc prin rotaţii: arborele $\mathcal{A}_i(0)$, cu $1 \leq i \leq d-1$, se obţine transformând orice nod $x \in \mathcal{A}_0(0)$ în $\mathrm{Ro}(x, i)$. Un exemplu de astfel de arbori este prezentat în figura 3.16b,c, în care se află $\mathcal{A}_1(0)$, respectiv $\mathcal{A}_2(0)$, pentru un hipercub 3-dimensional. Faptul că arborii sunt obţinuţi prin rotaţii a condus la denumirea algoritmului, numit *rotativ*. Aceşti arbori nu sunt arc-disjuncţi, dar mecanismul de difuzare descris mai sus evită conflictele în ocuparea arcelor. Totuşi, deşi timpul de difuzare este de aproximativ de d ori mai mic decât al algoritmului 3.4, nu se poate utiliza transmisia pipeline. Acest defect este remediat în continuare.

O familie de arbori de acoperire arc-disjuncţi. Pornim de la următoarea idee algoritmică intuitivă: P_0 trimite mesajul tuturor vecinilor săi; apoi, aceştia difuzează mesajul după câte un arbore de acoperire binomial, cu singurul amendament că nu îl mai retrimit lui P_0.

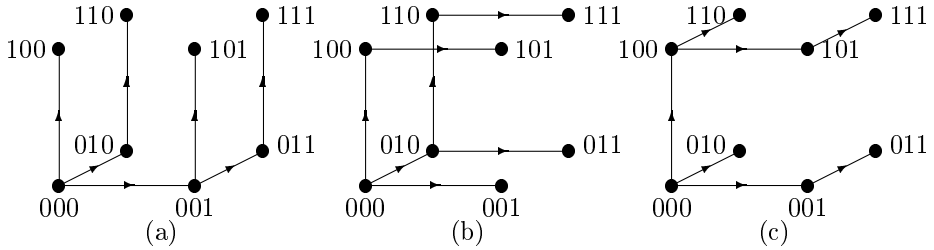


Figura 3.16: (a) Arbore de acoperire binomial într-un hipercub de dimensiune 3 (b), (c) arbori binomiali obţinuţi prin rotaţia primului cu una, respectiv două poziţii la stânga.

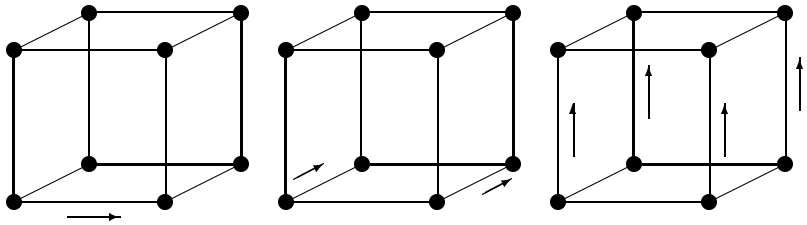


Figura 3.17: Difuzare după un arbore binomial, la fiecare pas pe o direcţie.

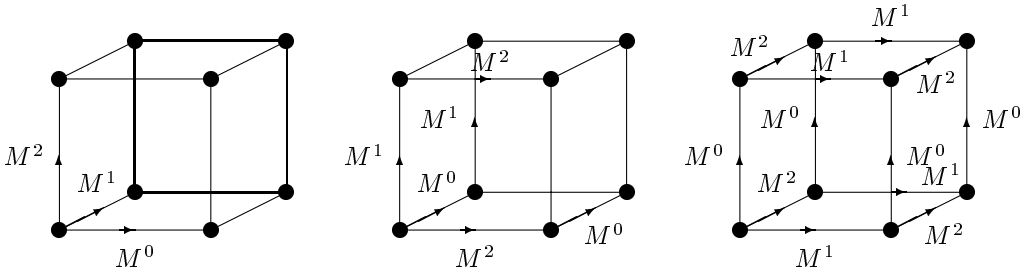


Figura 3.18: Difuzare într-un hipercub de dimensiune 3, algoritmul rotativ (pe o familie de arbori de acoperire binomiali).

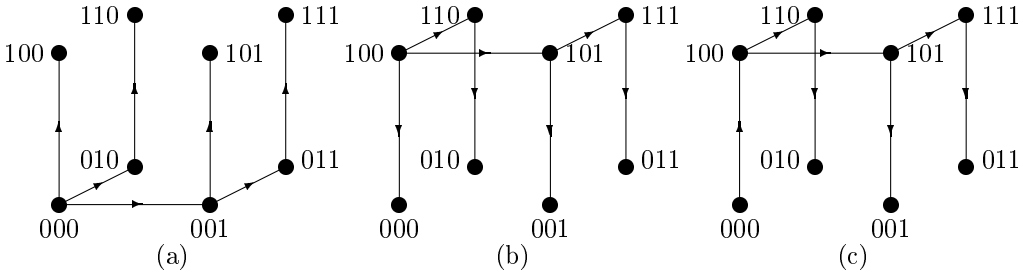


Figura 3.19: (a) Arbore de acoperire binomial; (b) Arbore obţinut prin translatarea primului în nodul 100; (c) Arborele translatat, cu sensul arcului din P_0 schimbat.

Definim întâi operaţia de *translaţie pe hipercub*. Un arbore oarecare $\mathcal{A}(0)$ este translatat cu rădăcina în nodul cu adresa s prin transformarea fiecărui nod $x \in \mathcal{A}(0)$ în $x \oplus s \in \mathcal{A}(s)$ (şi transformarea corespunzătoare a arcelor). Un exemplu este prezentat în figura 3.19, în care desenul (a) prezintă arborele de acoperire binomial $\mathcal{A}_0(000)$, iar desenul (b), arborele $\mathcal{A}_0(100)$. Să observăm că, după translaţie, orice arc rămâne în aceeaşi dimensiune (dacă nodurile x şi y sunt vecine în arbore, sunt şi în hipercub, deci adresele lor diferă într-un singur bit; atunci şi $x \oplus s$ şi $y \oplus s$ diferă în acelaşi bit).

Acum putem construi câte un arbore de acoperire binomial în fiecare dintre vecinii nodului 0. Pe care însă, dintre cei d arbori posibili, rădăcina fiind fixată şi scopul obţinerea unei familii arc-disjuncte ? Vom încerca să translatăm câte un arbore de acoperire binomial cu rădăcina în nodul 0, în fiecare din vecini. Ne bazăm pe faptul că în orice arbore de acoperire binomial există o singură frunză vecină cu rădăcina; de exemplu, pentru $\mathcal{A}_0(0)$, aceasta este 100 (în $\mathcal{H}_3$); vom translata arborele în această frunză; rezultatul a fost deja prezentat în figura 3.19b. Cum, la difuzare, comunicaţia între nodurile 0 şi 100 se desfăşoară dinspre 0, vom schimba sensul arcului dintre aceste două noduri, obţinând un arbore cu rădăcina în nodul 0; rezultatul este prezentat în figura 3.19c; acest arbore va fi notat $\mathcal{A}_0^D(0)$. Să generalizăm: în $\mathcal{A}_i(0)$, frunza vecină cu rădăcina este $2^{(i-1) \bmod d}$; atunci $\mathcal{A}_i^D(0)$ se obţine din $\mathcal{A}_i(2^{(i-1) \bmod d})$ schimbând sensul unicului arc adiacent nodului 0. Cei d arbori $\mathcal{A}_i^D(0)$, cu $0 \leq i \leq d-1$, sunt arc-disjuncţi (lăsăm fără demonstraţie această afirmaţie) şi au fiecare înălţime $d+1$. Pentru $\mathcal{H}_3$, această familie este prezentată în figura 3.20.

3.3.3 Difuzare generală

Actul de naştere al difuzării generale l-a constituit următoarea problemă frivolă: p persoane ştiu fiecare câte o parte dintr-o istorie mondenă; de câte telefoane este nevoie pentru ca toate persoanele să afle întreaga istorie ? Se presupune că, într-o comunicare telefonică, fiecare persoană spune tot ce ştie în momentul respectiv. Recunoaştem imediat modelul: 1-port, timp constant, full duplex (deşi nu se comunică simultan,

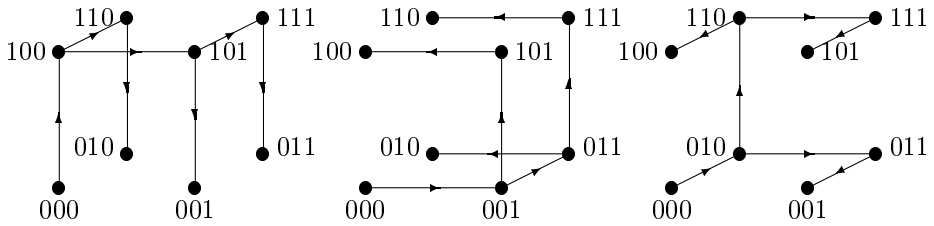


Figura 3.20: Familie de arbori de acoperire arc-disjuncţi într-un hipercub de dimensiune 3.

totuşi se folosesc ambele sensuri în *aceeaşi* convorbire); comunicarea se desfăşoară pe un graf complet; pentru aprofundare, vezi problema 3.3.19. Şi acum să trecem la formulări serioase.

O abordare naivă a problemei ar fi următoarea: un procesor colectează toate mesajele, apoi le difuzează celorlalte procesoare; după ce vom fi discutat despre colectare se va vedea că, de multe ori, aceasta durează cât o difuzare generală; în plus, difuzarea ar fi făcută cu un mesaj imens, de dimensiune pm. Deci, sunt slabe speranţe de performanţă.

Idee eficientă generală. O modalitate naturală şi eficientă de a obţine algoritmi de difuzare generală este de a aplica algoritmi de difuzare din fiecare nod, aceştia executându-se în paralel. Dacă, la un moment dat, un procesor are de trimis mai multe mesaje pe un canal, atunci le va concatena într-un mesaj mai lung, procesorului care recepţionează acest mesaj revenindu-i implicit sarcina de a extrage mesajele iniţiale şi de a le retransmite pe căile corespunzătoare. Dacă graful este simetric, nu vor exista dezechilibre de încărcare a canalelor. Se va vedea în continuare că algoritmi simpli de difuzare duc la bune performanţe în difuzarea generală; de asemenea, deoarece multe arce (eventual toate) vor fi ocupate simultan, nu mai este necesară utilizarea tehnicii pipeline.

Difuzare generală pe inel

Pe un inel, difuzarea generală durează tot atât ca şi o difuzare; dacă fiecare procesor îşi transmite mesajul pe ambele canale, apoi retransmite mesajele primite de la vecini, în ambele sensuri, toate canalele sunt folosite în paralelism complet, iar situaţia de a transmite două mesaje pe acelaşi canal în aceeaşi etapă nu apare. Practic, fiecare procesor îşi difuzează mesajul M_i după câte un arbore de acoperire de tipul celui din figura 3.11, procesorul fiind situat în rădăcină. Sunt $\lfloor p/2 \rfloor$ etape de comunicare, cât este înălţimea arborelui, după algoritmul următor:

ALGORITM 3.5 (difuzare generală pe inel, fiecare procesor pe un arbore de acoperire de înălţime minimă)

1. **pentru** $k = 0 : \lfloor p/2 \rfloor - 1$

1. **în paralel**
   1. **send**($M_{(\mathrm{id}-k) \bmod p}$, dreapta)
   2. **recv**($M_{(\mathrm{id}-k-1) \bmod p}$, stânga)
   3. **dacă** p este impar sau $k \neq \lfloor p/2 \rfloor - 1$ **atunci**, **în paralel**
      1. **send**($M_{(\mathrm{id}+k) \bmod p}$, stânga)
      2. **recv**($M_{(\mathrm{id}+k+1) \bmod p}$, dreapta)

În etapa k, un procesor retransmite vecinilor mesajele celor două procesoare aflate la distanţă k de el (propriile mesaje când $k = 0$) şi recepţionează de la vecini mesajele procesoarelor aflate la distanţă $k + 1$; evident, procesorul memorează separat aceste mesaje. Condiţia din linia 1.3 se explică prin faptul că atunci când p este par, cele două ramuri ale arborelui din figura 3.11 nu au lungimi egale; în acest algoritm am presupus că ramura din dreapta are lungime $p/2$, iar cea din stânga $p/2 - 1$, de aceea există o etapă în plus de comunicaţie spre dreapta. În regim full duplex, timpul de execuţie este

$$T_{3,R,F} = \lfloor p/2 \rfloor (\sigma + m\beta).$$

În modelul half duplex, circulaţia mesajelor se face într-un singur sens, de exemplu fiecare procesor transmite spre dreapta şi recepţionează simultan din stânga. Difuzarea generală se termină atunci când mesajul fiecărui procesor a ajuns la vecinul său din stânga. Sunt deci $p - 1$ etape de comunicaţie.

Difuzare generală pe tor

Pe tor, se poate generaliza algoritmul 3.5 de la inel. Într-o primă fază, se face difuzare generală pe fiecare inel orizontal în timp $(\sqrt{p}/2)(\sigma + m\beta)$; apoi, se face din nou o difuzare generală, dar acum pe fiecare inel vertical, fiecare procesor comunicând toate mesajele procesoarelor de pe linia sa; timpul necesar este $(\sqrt{p}/2)(\sigma + \sqrt{p}m\beta)$.

Se observă că, în orice moment, nu se folosesc decât jumătate dintre arce, cele orizontale în prima fază, cele verticale în a doua. Pentru a le utiliza pe toate recurgem la un truc simplu şi deja cunoscut; fiecare procesor îşi împarte mesajul în două pachete; primul este difuzat cum am spus mai sus, iar al doilea, după aceeaşi idee, dar întâi pe verticală şi apoi pe orizontală. În acest fel toate arcele sunt ocupate tot timpul, iar timpul de difuzare generală este

$$T_{3,T,F} = \sqrt{p}\,\sigma + \left(\frac{mp}{4} + \frac{m\sqrt{p}}{4}\right)\beta. \tag{3.6}$$

Difuzare generală pe hipercub

Pe un hipercub, simetrizarea algoritmului de difuzare pe un arbore de acoperire binomial conduce la o difuzare generală ce se desfăşoară după cum se sugerează în figura 3.21. La fiecare pas se comunică pe o singură dimensiune, în ordinea $0, 1, \ldots, d-1$, pe toate arcele posibile; fiecare procesor transmite vecinului său din acea dimensiune

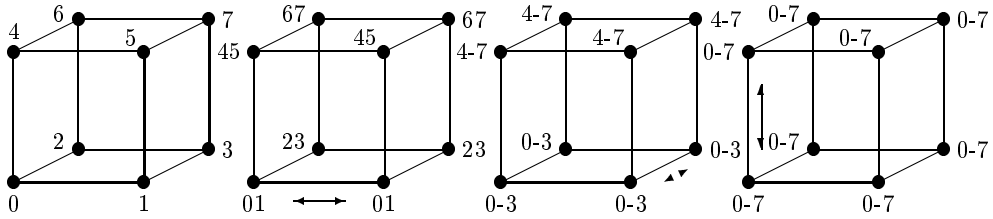


Figura 3.21: Difuzare generală într-un hipercub de dimensiune 3, la fiecare pas pe o direcţie (în noduri sunt notate mesajele ajunse acolo după fiecare etapă).

toate mesajele pe care le posedă. După fiecare pas dimensiunea mesajelor transmise se dublează; în figură, fiecare nod are asociate numerele mesajelor deja primite; de exemplu, după doi paşi, nodul 1 posedă mesajele 0, 1, 2, 3. La pasul k este încheiată difuzarea generală în cele 2^{d-k+1} hipercuburi formate cu arce din dimensiunile $0, 1, \ldots, k-1$ ($\tilde{\mathcal{H}}_d^{d-k+1}$, în notaţia de până acum), ceea ce arată natura recursivă a algoritmului (care poate fi descris scurt astfel: se face difuzare generală în două subhipercuburi componente, apoi se schimbă toate datele locale între procesoare aflate pe aceeaşi poziţie în cele două subhipercuburi).

Ca şi la difuzare, defectul acestui algoritm este că nu se folosesc, la un moment dat, decât arcele dintr-o singură dimensiune. De aceea încercăm simetrizarea algoritmului de difuzare rotativ, care conduce la folosirea în paralel a tuturor canalelor de comunicaţie. Procesorul P_j împarte mesajul propriu în d pachete M_j^i, cu $0 \leq i \leq d-1$. La pasul k, procesorul P_j transmite pe direcţia $l = (i+k) \bmod d$ pachetul (modificat prin concatenare) M_j^i; de asemenea, concatenează pachetele primite pe direcţia l cu M_j^i; în final, toate pachetele M_j^i, pentru $0 \leq j \leq p-1$, au acelaşi conţinut; pentru procesorul P_j, pachetele M_j^i, cu $0 \leq i \leq d-1$, conţin mesajele tuturor procesoarelor, adică întreaga cantitate de informaţie pe care trebuie să o primească P_j. Deşi principial algoritmul e destul de simplu, concatenarea pachetelor conduce la amestecarea lor; de aceea, în final (sau, mai bine, după fiecare etapă de comunicaţie) fiecare procesor trebuie să pună în ordine pachetele primite; aceasta se poate face simplu dacă se ataşează fiecărui pachet M_j^i numerele j (procesorul sursă) şi i (numărul de ordine) ca antet. Pentru a exemplifica amestecarea, să ne mai uităm la figura 3.21; presupunem că se concatenează mesajul primit la finalul celui local; procesorul 5 are după prima etapă mesajele 5,4, după a doua 5,4,7,6, şi după a treia 5,4,7,6,1,0,3,2; aici ordonarea se face numai după numărul procesorului sursă; evident, când fiecare mesaj e împărţit în pachete, lucrurile se complică şi mai mult. Timpul de difuzare generală este

$$T_{3,H,F} = (\log p)\sigma + \frac{(p-1)m}{\log p}\beta.$$

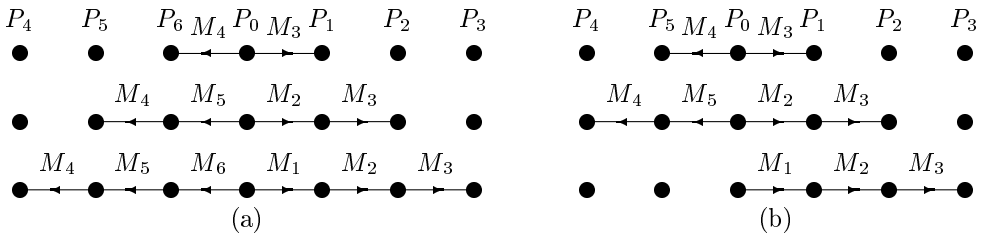


Figura 3.22: Distribuţie în trei etape, pe un inel: (a) cu 7 procesoare; (b) cu 6 procesoare.

3.3.4 Difuzare personalizată (distribuţie)

Deoarece procesorul P_0 trebuie să trimită mesaje tuturor celorlalte procesoare, utilizarea unui arbore de acoperire sau a unei familii de arbori de acoperire arc-disjuncţi este naturală. Principiul de utilizare a unui arbore de acoperire pentru distribuţie este ca P_0 să trimită mesajele *întâi celor mai îndepărtate* procesoare. Astfel, în timp ce primele mesaje se deplasează mai departe prin arbore spre destinaţiile lor, procesorul P_0 poate trimite mesaje procesoarelor mai apropiate; se poate spera la utilizarea în paralel a multor arce ale arborelui. Vom vedea, în cazul hipercubului, ce calităţi este bine să aibă arborii de acoperire.

Distribuţie pe inel

Pe un inel, procesorul sursă P_0 transmite spre stânga şi spre dreapta simultan, deci folosind arborele de acoperire din figura 3.11, întâi celor mai îndepărtate procesoare, i.e. cele cu adresele $\lfloor p/2 \rfloor$, prin dreapta, şi respectiv $\lfloor p/2 \rfloor + 1$, prin stânga. La pasul următor, P_0 trimite mesajele pentru procesoarele aflate cu un arc mai aproape etc. Toate celelalte procesoare retransmit mesajele primite, până în momentul în care îl primesc pe cel destinat lor. Algoritmul este prezentat în continuare şi exemplificat în figura 3.22.

ALGORITM 3.6 (P_0 distribuie pe inel)

1. $j \leftarrow \lfloor p/2 \rfloor - dist(\mathrm{id}, 0) + 1$
2. **dacă** p este par şi $\mathrm{id} > \lfloor p/2 \rfloor$ **atunci** $j \leftarrow j - 1$
3. **pentru** $k = 0 : j - 1$
   1. **dacă** $\mathrm{id} = 0$ **atunci**, **în paralel**
      1. **send**($M_{\lfloor p/2 \rfloor - k}$, dreapta)
      2. **dacă** p este impar sau $k < j - 1$ **atunci** **send**($M_{\lfloor p/2 \rfloor + 1 + k}$, stânga)
   2. **altfel dacă** $\mathrm{id} \leq \lfloor p/2 \rfloor$ **atunci**, **în paralel**
      1. **recv**($M_{\lfloor p/2 \rfloor - k}$, stânga)
      2. **dacă** $k > 0$ **atunci** **send**($M_{\lfloor p/2 \rfloor - k - 1}$, dreapta)
   3. **altfel**, **în paralel** $\{\mathrm{id} > \lfloor p/2 \rfloor\}$

1. **recv**($M_{\lfloor p/2 \rfloor+1+k}$, dreapta)
2. **dacă** $k > 0$ **atunci** **send**($M_{\lfloor p/2 \rfloor+k}$, stânga)

Numărul de etape la care participă un procesor este calculat în instrucţiunile 1 şi 2; el este cu atât mai mic cu cât distanţa de procesorul care distribuie este mai mare; când p este par, arborele de acoperire din figura 3.11 are ramurile de lungimi inegale, deci procesoarele din stânga lui P_0 participă la o etapă de comunicaţie mai puţin, după cum se vede şi în figura 3.22b. Procesorul P_0 transmite în ambele sensuri simultan în fiecare etapă, mai puţin în ultima, dacă p este par, după cum se vede din 3.1.2. În fine, celelalte procesoare doar recepţionează în prima lor etapă de comunicaţie, apoi recepţionează şi transmit simultan. Instrucţiunile 3.2.1 şi 3.2.2 sunt identice cu 3.3.1 şi respectiv 3.3.2, doar sensul de comunicaţie este schimbat; este descrisă comunicaţia pentru procesoarele din dreapta, respectiv stânga, lui P_0. Remarcăm că un procesor poate folosi doar două zone de memorie, alternativ pentru recepţie şi transmisie; ultimul mesaj primit este cel destinat lui. Complexitatea acestui algoritm este

$$T_{4,R} = \lfloor p/2 \rfloor \, (\sigma + m\beta).$$

Aplicarea tehnicii pipeline în distribuţie nu face decât să înrăutăţească performanţele. La inel, aceasta se vede imediat, deoarece procesorul P_0 este în permanenţă ocupat. Spargerea mesajelor în pachete n-ar modifica timpul de propagare, dar ar mări coeficientul lui σ, deci şi timpul total; aşadar, nu se utilizează pipeline.

Distribuţie pe tor

Pe tor, putem să ne inspirăm din algoritmul de la inel, folosind implicit arborele de acoperire din figura 3.13 după cum urmează. În prima fază, nodul P_{00} distribuie fiecărui nod de pe orizontala sa toate cele $\sqrt{p}$ mesaje ce trebuie trimise pe verticala acelui nod; timpul necesar este (aproximativ) $(\sqrt{p}/2)(\sigma + m\sqrt{p}\beta)$. În a doua fază, fiecare procesor de pe linia 0 distribuie pe inelul vertical pe care se află; de data aceasta talia mesajelor este m, deci timpul necesar este $(\sqrt{p}/2)(\sigma + m\beta)$.

Pentru utilizarea simultană a mai multor canale, se împarte fiecare mesaj în două pachete; primul pachet se distribuie ca mai sus; al doilea, după aceeaşi idee, dar întâi pe verticală şi apoi pe orizontală, adică după un arbore de acoperire obţinut prin rotaţia celui din figura 3.13 cu 90 de grade în sens trigonometric. Timpul de distribuţie este identic cu cel de difuzare generală dat de (3.6).

Distribuţie pe hipercub

Distribuţie pe un arbore de acoperire binomial. Pe un hipercub, o primă idee este de a transmite folosind un arbore de acoperire binomial; să încercăm, pentru a vedea care sunt dezavantajele. În figura 3.23 este prezentată evoluţia comunicaţiilor, dacă în fiecare etapă se transmite pe câte o dimensiune; în prima etapă, P_0 transmite (concatenate) toate mesajele pentru procesoarele din subhipercubul $\bar{\mathcal{H}}_d^{d-1}(1)$, în a

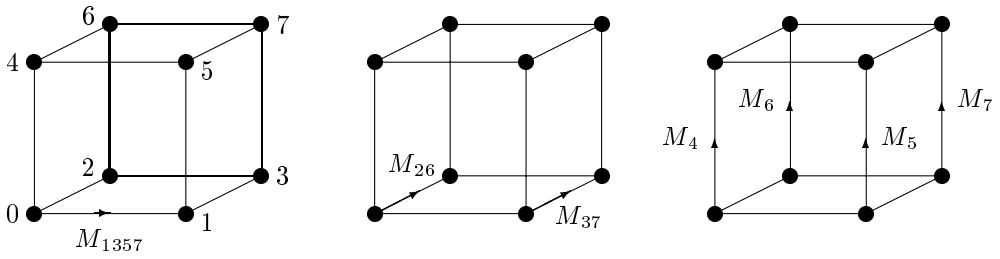


Figura 3.23: Difuzare personalizată, într-un hipercub de dimensiune 3, pe un arbore de acoperire binomial, la fiecare pas pe o direcţie.

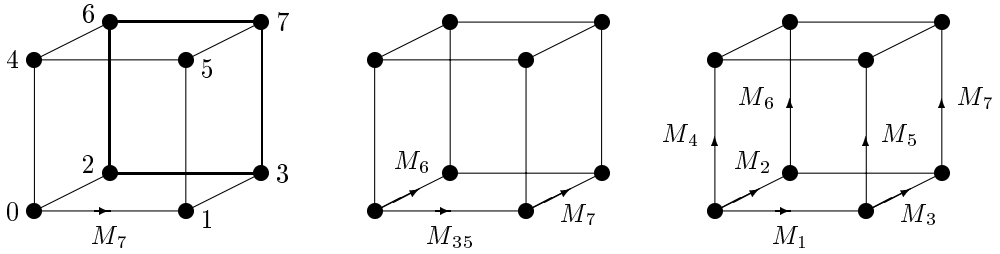


Figura 3.24: Difuzare personalizată pe un arbore de acoperire binomial, după principiul "întâi celui mai îndepărtat".

doua, toate mesajele pentru $\bar{\mathcal{H}}_d^{d-2}(2)$; în general, în etapa k, cu $0 \leq k \leq d-1$, trimite pe dimensiunea k toate mesajele pentru subhipercubul $\bar{\mathcal{H}}_d^{d-1-k}(2^k)$; fiecare procesor care primeşte un mesaj păstrează porţiunea destinată lui şi transmite restul mai departe în subarborele a cărui rădăcină este, într-un mod asemănător cu P_0. Se observă că dimensiunea mesajelor se înjumătăţeşte la fiecare pas, iniţial fiind $2^{d-1}m$. Complexitatea acestui algoritm 1-port este:

$$T_{4,H,1} = (\log p)\sigma + (p-1)m\beta.$$

Tot pe un arbore de acoperire binomial, se poate proceda trimiţând mesajele după principiul "întâi celui mai îndepărtat", aşa cum este sugerat în figura 3.24. În etapa k $(0 \leq k \leq d-1)$, P_0 trimite, simultan pe toate canalele sale, mesajele pentru nodurile aflate la distanţă $d-k$, mesajele pentru nodurile din acelaşi subarbore fiind concatenate; timpul de transmisie este mai bun de maximum două ori decât precedentul:

$$T'_{4,H} = (\log p)\sigma + \frac{p}{2}m\beta.$$

El se justifică prin faptul că, într-un arbore binomial, unul dintre subarborii rădăcinii, cel mai "stufos", are jumătate din noduri; timpul de distribuţie este cel necesar comunicărilor de la rădăcină spre acest subarbore; nu mai sunt introduse întârzieri

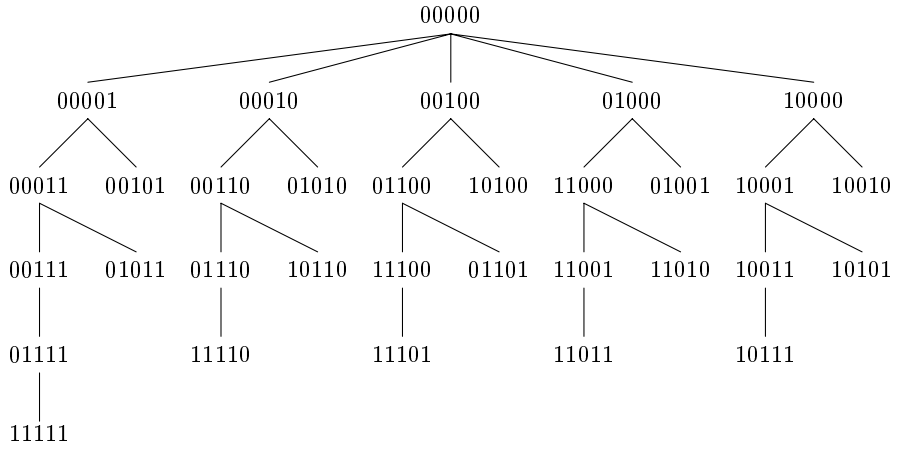


Figura 3.25: Arbore de acoperire echilibrat pentru un hipercub de dimensiune 5.

suplimentare, arborele binomial având proprietatea că, în orice subarbore, un nod oarecare are mai puţini fii decât rădăcina subarborelui (şi, în general, are mai puţini descendenţi la distanţă x decât are rădăcina subarborelui, tot la distanţă x). Figura 3.24 e puţin eronată, din exces de didacticism; cum M_6 şi M_7 sunt de două ori mai scurte decât M_{35}, ele ajung mai repede la destinaţie, în al doilea desen, şi pornesc imediat mai departe; deci, în al treilea desen, M_6 şi M_7 sunt aproape transmise, când începe transmisia celorlalte mesaje. Totuşi, timpul total rămâne cel de mai sus.

Arbore de acoperire echilibrat. Se observă că timpul de execuţie ar descreşte dacă P_0 ar trimite mesaje, în fiecare etapă, pe toate canalele sale; pe de altă parte, căile de transmitere trebuie să aparţină unui arbore de acoperire. Aceasta se poate realiza numai dacă arborele de acoperire este *echilibrat*, adică toţi subarborii rădăcinii au număr de noduri "aproape" egal (diferind cel mult prin 1). Un astfel de arbore de acoperire echilibrat este prezentat în figura 3.25; nu detaliem construcţia sa; cu puţină atenţie se poate observa că, la fel ca la arborele binomial, pe nivelul k sunt $\binom{d}{k}$ noduri, adică maximul posibil. Menţionăm totuşi că arborele este într-adevăr echilibrat doar când d este prim; oricum, în celelalte cazuri, diferenţele între subarbori sunt neglijabile.

Deoarece acest arbore este echilibrat, timpul de execuţie al algoritmului în care se transmit întâi mesajele destinate nodurilor cel mai depărtate din subarbori este

$$T''_{4,H} = (\log p)\sigma + \left\lceil \frac{p-1}{\log p} \right\rceil m\beta.$$

Un timp de execuţie asemănător cu $T''_{4,H}$ se obţine dacă se foloseşte o familie de arbori de acoperire binomiali rotativi (cum este cea din figura 3.16), pe fiecare arbore P_0 transmiţând câte m/d valori din fiecare mesaj; algoritmul este prezentat în figura

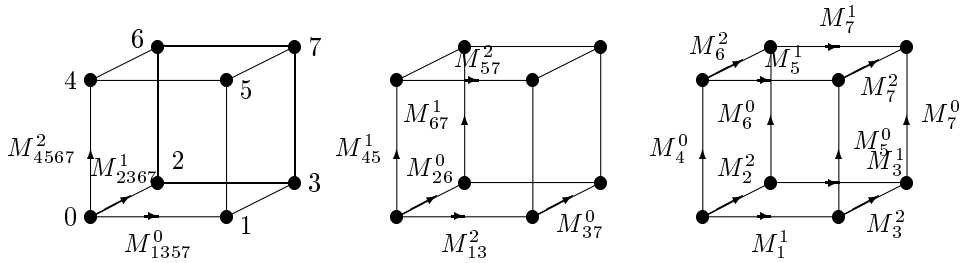


Figura 3.26: Difuzare personalizată pe o familie de arbori de acoperire binomiali rotativi.

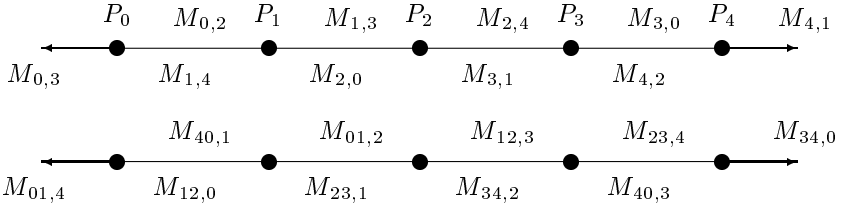


Figura 3.27: Schimb complet pe un inel cu 5 procesoare, în două etape; mesajele de deasupra liniilor orizontale circulă spre dreapta, iar cele de dedesubt, spre stânga.

3.26 şi este o generalizare acum banală a celui din figura 3.23.

3.3.5 Schimb complet

Deoarece în schimbul complet fiecare procesor efectuează o distribuţie, putem obţine algoritmi simpli prin simetrizarea celor de distribuţie. Ca şi la difuzarea generală, această strategie poate asigura ocuparea în paralel a tuturor canalelor, fără a fi necesară transmisia pipeline. Vom ilustra ideea doar în cazul inelului, cititorul fiind invitat să scrie algoritmii pentru tor şi hipercub.

Schimb complet pe inel. Dacă fiecare procesor trimite mesaje pe ambele canale, începând cu cele adresate celor mai depărtate procesoare, iar în etapele următoare transmite atât mesajele sale, cât şi pe cele primite de la vecini, evoluţia comunicaţiei este de genul celei prezentate în figura 3.27; în etapa k, cu $0 \leq k < p/2$, fiecare procesor transmite k mesaje concatenate (cu un mesaj mai mult decât în etapa precedentă) pe fiecare canal. Descriem mai jos algoritmul pentru p impar; în cazul p par se întâmplă acelaşi fenomen ca la distribuţie: în ultima etapă se comunică într-un singur sens; nu mai detaliem, pentru simplitate.

ALGORITM 3.7 (schimb complet pe inel, pentru p impar)

1. $T_s \leftarrow \emptyset$, $T_d \leftarrow \emptyset$
2. **pentru** $k = 0 : \lfloor p/2 \rfloor - 1$
   1. $jd \leftarrow (\mathrm{id} + \lfloor p/2 \rfloor - k) \bmod p$

2. $js \leftarrow (\text{id} - \lfloor p/2 \rfloor + k) \bmod p$
3. $L_d \leftarrow T_s$ concatenat cu $M_{\text{id},jd}$
4. $L_s \leftarrow M_{\text{id},js}$ concatenat cu T_d
5. **în paralel**
   1. **send**(L_d, dreapta), **send**(L_s, stânga), **recv**(T_d, dreapta), **recv**(T_s, stânga)

Variabilele folosite au următoarea semnificaţie: jd şi js sunt adresele celor două procesoare cărora le sunt destinate mesajele curente, aflate la dreapta procesorului curent, respectiv la stânga (pe calea cea mai scurtă); T_d şi T_s sunt zonele de memorie în care se depun mesajele primite, în etapa curentă, din dreapta, respectiv stânga; în fine, în L_d şi L_s se formează mesajele destinate transmisiei, în etapa curentă, spre dreapta, respectiv stânga. Deci, în fiecare etapă, un procesor concatenează câte un mesaj propriu mesajelor primite în etapa anterioară şi apoi le trimite mai departe; mesajele destinate lui sunt recepţionate în ultima etapă, deci se vor afla în variabilele T_d şi T_s. Datorită modului de scriere a instrucţiunilor 2.3 şi 2.4, T_d va conţine mesajele de la procesoarele, în ordine, $(\text{id}+1) \bmod p$, ..., $(\text{id}+\lfloor p/2 \rfloor) \bmod p$, iar T_s de la, tot în ordine, $(\text{id}-\lfloor p/2 \rfloor) \bmod p$, ..., $(\text{id}-1) \bmod p$; deci, mesajele sunt în ordinea crescătoare a adreselor expeditorilor, mai puţin o rotaţie.

Ca şi la distribuţie, au loc $\lfloor p/2 \rfloor$ etape de comunicaţie, deci timpul de execuţie al algoritmului, în regim full duplex, este

$$T_{5,R,F} = \sum_{k=1}^{p/2} (\sigma + km\beta) = \frac{p}{2}\sigma + \frac{p}{4}(\frac{p}{2}+1)m\beta.$$

Este interesant că se poate aplica o strategie într-un fel inversă; fiecare procesor începe prin a trimite jumătate din mesaje spre dreapta (cele pentru procesoarele mai apropiate prin dreapta) şi jumătate spre stânga. În a doua etapă, fiecare procesor reţine mesajul destinat lui, şi retransmite restul mai departe; deci, în etapa k ($0 \leq k < \lfloor p/2 \rfloor$), un procesor trimite $\lfloor p/2 \rfloor - k$ mesaje concatenate (cu unul mai puţin decât în etapa precedentă). Timpul de comunicaţie va fi exact acelaşi.

Schimb complet pe tor. Pe tor generalizăm algoritmii de la inel. Se face întâi câte un schimb complet pe fiecare inel orizontal, cu mesajele destinate tuturor procesoarelor de pe aceeaşi verticală; un procesor trimite altuia câte $\sqrt{p}$ mesaje. Urmează câte un schimb complet pe fiecare verticală, fiecare procesor având de transmis câte $\sqrt{p}$ mesaje fiecărui alt procesor de pe acelaşi inel vertical (câte unul de la fiecare procesor de pe inelul orizontal, inclusiv el însuşi). Timpii pentru cele două faze sunt deci egali (iată totuşi o diferenţă faţă de difuzarea generală şi distribuţie).

Din nou, dacă se separă fiecare mesaj în două pachete şi se foloseşte algoritmul precedent pentru primul pachet, şi acelaşi algoritm, dar întâi pe verticală, pentru al doilea, timpul de propagare se înjumătăţeşte.

Transmisie permutată. Vom aminti acum, pe scurt, o utilizare a schimbului complet într-o problemă de comunicaţie aparent fără mare legătură cu acesta. Să presupunem că fiecare procesor doreşte să transmită un singur mesaj, unui alt procesor,

astfel încât fiecare procesor are de transmis şi de recepţionat câte un mesaj. Problema se numeşte a transmisiei permutate (permuted send). Un exemplu de astfel de comunicaţie este *rotaţia*, în care fiecare procesor transmite vecinului său din dreapta, pe un inel; alt exemplu este *inversarea*, în care P_i schimbă câte un mesaj cu P_{p-1-i}. În general, pot apărea destule situaţii în care comunicaţia nu este deloc regulată.

O metodă de a regulariza transmisia permutată este următoarea: fiecare procesor îşi împarte mesajul în p pachete, pe care le transmite, câte unul fiecăruia dintre celelalte procesoare, unul păstrându-l; acesta este un schimb complet. Acum, fiecare procesor are tot p pachete, dar destinate câte unul fiecărui procesor; fiecare procesor păstrează pachetul său, trimiţându-le pe celelalte $p-1$ celorlalte procesoare; acesta este, din nou, un schimb complet. Dacă privim din punctul de vedere al transmiterii mesajului între procesoarele P_s şi P_a, atunci P_s distribuie cele p pachete, după care P_a le colectează. În acest fel, transmisia permutată se transformă în două schimburi complete. Aceasta poate fi o soluţie excelentă, mai puţin în cazurile în care există o variantă simplă, ca, de exemplu, pentru rotaţie.

3.3.6 Câteva concluzii

În tot restul lucrării vom utiliza primitive cu numele operaţiilor de comunicaţie globală prezentate până acum, folosind o descriere nu foarte formală, dar suficient de explicită, de exemplu

difuzare: P_x trimite mesajul A celorlalte procesoare.

Aici poate apărea o confuzie destul de gravă. Această primitivă nu este apelată doar de procesorul P_x, ci de *toate* procesoarele, în aceeaşi formă. În funcţie de adresa sa, fiecare procesor îşi îndeplineşte rolul în difuzare. Evident, adresa x trebuie să aibă aceeaşi valoare în toate apelurile, deci, în programul unui anume procesor, să nu depindă în nici un fel de adresa procesorului.

În ce priveşte estimarea timpilor pentru executarea acestor primitive, vom considera, în general, timpii cei mai buni dintre cei prezentaţi. Există totuşi un amendament, pentru cazurile în care se poate folosi pipeline. Dacă mesajele au dimensiuni mari (din păcate, noţiunea de "mare" nu poate fi explicitată prea mult, ea depinzând de modelul de comunicaţie şi de valorile parametrilor comunicaţiei), vom lucra cu timpii pentru pipeline, deci într-adevăr cei mai buni. Dacă însă mesajele sunt scurte, atunci vom presupune că nu se utilizează pipeline (chiar dacă s-ar utiliza, câştigul ar fi nesemnificativ). Oricum, în multe cazuri ne vom mulţumi cu o apreciere mai vagă a timpului, precizând, de exemplu, că o difuzare pe hipercub durează $O(m)$ (sau $O(m/\log p)$, cu pipeline).

Probleme

P 3.3.1 Deduceţi o limită inferioară pentru timpul de comunicaţie între două noduri aflate la distanţă j, într-un graf regulat oarecare de grad Δ. La fel, dacă graful este neregulat.

P 3.3.2 Considerăm un graf complet, în care oricare două noduri sunt conectate. Câte căi arc-disjuncte există între două noduri oarecare ?

P 3.3.3 Când sunt optime cele patru căi din figura 3.8a ?

P 3.3.4 Găsiţi patru căi arc-disjuncte între două noduri vecine pe o grilă. Indicaţie: lungimile lor sunt 1, 3, 5, respectiv 7.

P 3.3.5 Cum se construiesc căile de comunicare între nodurile P_s şi P_a dintr-un hipercub, cunoscând numai căile dintre nodurile 0 şi $2^j - 1$, unde $j = H(s, a)$?

P 3.3.6 a) Să se arate că cele d căi construite prin rotaţii ale căii principale între nodurile 0 şi $2^d - 1$, în hipercubul $\mathcal{H}_d$, nu au nici un nod comun (în afara capetelor). b) La fel, pentru cele d căi între nodurile 0 şi $2^j - 1$, construite în secţiunea 3.3.1.

P 3.3.7 Câte arce sunt între C_k şi C_{k+1}, în hipercubul $\mathcal{H}_d$?

P 3.3.8 În tabelul 3.1 sunt prezentate margini inferioare ale timpului necesar unor operaţii de comunicaţie globală, în modelele full şi half duplex, timp liniar, pentru un graf oarecare cu diametrul D şi gradul Δ. Demonstraţi valabilitatea acestor margini.

P 3.3.9 Care este numărul maxim de arbori de acoperire arc-disjuncţi, cu aceeaşi rădăcină, care se pot construi într-un graf oarecare, regulat şi de grad Δ ?

P 3.3.10 Scrieţi algoritmul de difuzare pe inel, după arborele de acoperire din figura 3.11.

P 3.3.11 Modificaţi algoritmul din problema anterioară astfel încât să fie corect indiferent de procesorul P_s care face difuzarea. La fel, pentru algoritmul 3.3.

P 3.3.12 Scrieţi algoritmul de difuzare pe tor, după arborele de acoperire din figura 3.13.

P 3.3.13 Dacă un arc este în dimensiunea k în $\mathcal{A}_0(0)$, în ce dimensiune se va afla după rotaţia prin care se obţine $\mathcal{A}_i(0)$?

P 3.3.14 În ce etapă de comunicaţie va primi mesajul M procesorul P_j, în algoritmul 3.4 ? Dar mesajul M^i, în algoritmul de difuzare rotativ ?

P 3.3.15 Scrieţi algoritmul de difuzare rotativ pentru hipercub.

P 3.3.16 Dintr-un algoritm de difuzare se poate obţine întotdeauna un algoritm de comunicare de la procesorul care difuzează (sursa) la un oricare altul (destinaţia), eliminând toate transmisiile inutile ale mesajului; gândind în termeni de arbori de acoperire, se elimină toate arcele care nu sunt pe căile dintre sursă şi destinaţie. Ce se obţine din particularizarea algoritmului de difuzare după familia de arbori arc-disjuncţi $\mathcal{A}_i^D(0)$?

P 3.3.17 Care este timpul de difuzare într-un graf oarecare de diametru D, considerându-se modelul multiport, cu timp constant ?

P 3.3.18 Care este timpul de difuzare într-un graf complet, în modelul 1-port, cu timp constant ? Dar într-un hipercub ? Care este limita inferioară pentru un graf oarecare ?

P 3.3.19 Cât durează difuzarea generală pe graful complet, în modelul timp constant, multiport, full duplex ? Dar în half duplex ? Dar în modelul 1-port ?

P 3.3.20 Scrieţi algoritmul de difuzare în regim half duplex pe inel.

Operaţia	Full duplex	Half duplex
Difuzare	$D\sigma + (D - 1 + \frac{m}{\Delta})\beta$	$\frac{2(p-1)}{p\Delta} m\beta$
Difuzare generală	$(p-1)\frac{m}{\Delta}\beta$	$2(p-1)\frac{m}{\Delta}\beta$
Distribuţie	$(p-1)\frac{m}{\Delta}\beta$	$(p-1)\frac{m}{\Delta}\beta$
Schimb complet, inel	$\frac{p}{4}(\frac{p}{2}+1)m\beta$	$\frac{p}{2}(\frac{p}{2}+1)m\beta$
Schimb complet, hipercub	$\frac{pm}{2}\beta$	$pm\beta$

Tabelul 3.1: Margini inferioare pentru timpii necesari operaţiilor de comunicaţie globală.

P 3.3.21 Care este timpul de distribuţie pe un graf oarecare în modelul cu timp constant ? Sugeraţi algoritmi de distribuţie în modelul 1-port (şi timp constant), pentru inel şi hipercub. Indicaţie: timpi optimi: $\lfloor (p+1)/2 \rfloor$ pentru inel şi $\log p$ pentru hipercub.

P 3.3.22 Sugeraţi un algoritm simplu de distribuţie pe tor, care să aibă un timp de execuţie asemănător cu $T'_{4,T}$, deci să aibă coeficientul lui β aproape optim, dar în care mesajele să nu fie împărţite în pachete.

P 3.3.23 Demonstraţi că, în algoritmul de schimb complet pe tor, fiecare mesaj parcurge calea cea mai scurtă între sursă şi destinaţie.

P 3.3.24 Demonstraţi că $\sum_{i=0}^{d} i\binom{d}{i} = d2^{d-1}$.

P 3.3.25 Se consideră o matrice pătrată A, partiţionată în blocuri egale, şi ele pătrate, notate A_{ij}. a) Presupunem că blocurile se distribuie celor p procesoare astfel încât P_k are blocurile P_{kj}, pentru $0 \leq j < p$ (are o linie de blocuri). Cum se efectuează transpunerea matricei, pe un inel ? b) Dar pe un hipercub ? c) Pe un tor, presupunem că procesorul P_{ij} deţine blocul A_{ij}. Care este acum algoritmul de transpunere ?

P 3.3.26 Cât durează schimbul complet pe un graf complet, în modelul timp constant şi multiport, respectiv 1-port ?

3.4 Comunicaţii în modelul comutare de circuit

Dedicăm o secţiune separată acestui model, dar fără amploarea celei pentru store and forward. Ne vom mulţumi cu câteva exemple care să sublinieze filosofia de abordare a problemelor. Notaţiile sunt cele deja utilizate. Trebuie să facem o mică modificare, totuşi; la primitiva **send**, în modelul store and forward era suficientă indicarea canalului pe care era transmis un mesaj; acum suntem obligaţi să precizăm şi procesorul destinaţie a mesajului, astfel:

send(*date, canal,* P_a)

Deci, mesajul *date* va fi trimis pe canalul *canal*, către procesorul P_a. Vă întrebaţi poate de ce mai trebuie precizat canalul; răspunsul e simplu: pentru că, în general,

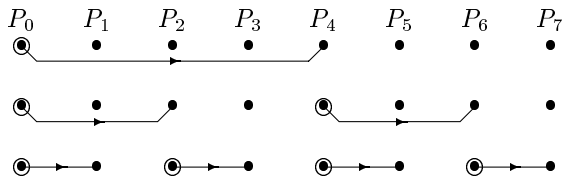


Figura 3.28: Difuzare într-un inel cu 8 procesoare, model comutare de circuit, în 3 etape.

nu există o cale unică până la P_a; alte argumente vor fi oferite peste câteva pagini, când vom vorbi despre dirijare (acolo se va lămuri şi cum se alege calea dintre sursă şi P_a, care nu e unică nici după ce sursa a trimis mesajul pe un canal precizat). Vom introduce şi pentru **recv** acest al treilea argument, cu semnificaţia de adresă a expeditorului unui mesaj; el este necesar pentru a putea selecta un mesaj din mai multe sosite pe acelaşi canal; în plus, clarifică scrierea algoritmilor.

Comunicaţie punct-la-punct. În comunicaţia între două procesoare, timpul este cu atât mai mic cu cât există mai multe căi arc-disjuncte între sursă şi destinaţie. Dacă există c căi, de lungime cel mult h, atunci mesajul este împărţit în c pachete, fiecare trimis pe o cale. Timpul total este dictat de calea cea mai lungă:

$$T_1 = \sigma + h\tau + (m/c)\beta.$$

Pentru comparaţie, revedeţi formula (3.5), de la store and forward în regim pipeline.

Difuzare pe inel, model 1-port. Dacă în comunicarea între două procesoare modelul comutare de circuit este evident mai performant, în difuzare lucrurile nu sunt chiar simple. Să studiem puţin difuzarea pe inel, în modelul 1-port. În primul pas de comunicaţie, procesorul sursă P_0 informează un alt procesor; în al doilea, cele două procesoare mai pot informa (simultan) alte două, ş.a.m.d., numărul procesoarelor informate dublându-se la fiecare pas. Deci, numărul minim de etape de comunicaţie este $\lceil \log p \rceil$. Desigur, pentru a se atinge acest minim, e nevoie ca, în fiecare etapă, căile de comunicaţie să fie arc-disjuncte. Din fericire, pentru inel, aceasta se obţine cu uşurinţă, după cum se vede în figura 3.28. Să presupunem că sunt $p = 2^r$ procesoare. În prima etapă P_0 comunică cu $P_{2^{r-1}}$; în a doua, cele două comunică cu procesoarele aflate la distanţă 2^{r-2} spre dreapta; în fiecare etapă distanţa de comunicare se înjumătăţeşte faţă de precedenta; în etapa k, cu $0 \leq k \leq r-1$, se comunică la distanţă 2^{r-k-1}. Algoritmul poate fi descris astfel:

ALGORITM 3.8 (P_0 difuzează mesajul M pe inel, model comutare de circuit, 1-port, $p = 2^r$, în r etape)

1. $j \leftarrow$ numărul de biţi finali de 0 din id
2. **dacă** id $\neq 0$ **atunci** **recv**(M, stânga, id $- 2^j$)
3. **pentru** $l = j - 1 : -1 : 0$ $\quad \{l = r - k - 1\}$
   1. **send**(M, dreapta, id $+ 2^l$)

Fiecare procesor primeşte o singură dată mesajul M şi îl transmite de j ori; valoarea j se calculează în instrucţiunea 1; echivalent, se poate scrie id $= z2^j$, cu z un număr impar (convenim că $j = r$, pentru id $= 0$). Timpul total pentru acest algoritm este

$$T_{2,R} = \sum_{k=0}^{r-1} (\sigma + 2^{r-k-1}\tau + m\beta) = (\log p)\sigma + (p-1)\tau + (\log p)m\beta.$$

Comparând acum cu timpii pentru difuzare pe inel în modelul store and forward, observăm că, pentru mesaje scurte, timpii erau $O(pm/2)$, dar pentru mesaje lungi, $O(m/2)$ în regim pipeline. Timpul în model comutare de circuit se află între aceştia; avantajul modului pipeline provine din faptul că toate procesoarele de pe o cale memorau conţinutul mesajelor, în timp ce acum ele nu o fac (am încercat să sugerăm aceasta în figura 3.28, liniile pe care "circulă" mesajele, netrecând prin procesoarele intermediare, deşi în realitate o fac, desigur).

Difuzare în model multiport şi generalizare. În modelul multiport, P_0 poate informa simultan două procesoare, de exemplu pe cele aflate la distanţă $p/3$; apoi aceste trei procesoare informează alte 6, la distanţă $p/3^2$; în general, la pasul k (primul pas este notat 0), se comunică la distanţă $p/3^{k+1}$, în total existând 3^{k+1} procesoare informate. Numărul de paşi este deci $\lceil \log_3 p \rceil$.

Aceasta strategie poate fi încercată pe orice graf regulat cu grad Δ. După prima etapă, sunt $\Delta + 1$ procesoare informate, după a doua $(\Delta + 1)^2$ etc. Se poate spera la un număr de $\lceil \log_{\Delta+1} p \rceil$ etape. Problema este de a găsi căi arc-disjuncte şi apoi, eventual, de a găsi distanţa optimă între procesoarele care comunică (de a minimiza coeficientul lui τ). De exemplu, pe tor, un algoritm destul de regulat poate decurge în modul schiţat în figura 3.29. Se împarte torul în 9 "pătrate" egale, P_{00} fiind în centrul pătratului central; în prima etapă P_{00} informează simultan procesoarele aflate în centrele celor patru pătrate din colţuri; în a doua, procesoarele din centrele celorlalte patru pătrate. Apoi, procesoarele din centrele tuturor celor 9 pătrate continuă în acelaşi mod, fiecare pătrat fiind împărţit în 9 pătrate mai mici; evident că în pătratele din colţuri difuzarea poate începe mai devreme, după cum se vede în figura 3.29. La fiecare două etape, numărul procesoarelor informate creşte de 9 ori; un mic calcul arată că numărul total de etape este $2\lceil \log_9 p \rceil$.

Alte operaţii de comunicaţie globală. În difuzarea generală nu se poate spera la îmbunătăţiri datorate comutării de circuit, deoarece în cei mai eficienţi algoritmi în modelul store and forward nu se utilizează pipeline, iar arcele sunt tot timpul ocupate. E interesant că nici la distribuţie nu se poate face mai bine; într-un algoritm în care procesorul sursă emite permanent pe toate canalele, este inutil ca el să-şi transmită mesajele mai departe de vecini, ceea ce ar costa mai mulţi τ, atâta vreme cât vecinii nu au altceva de făcut decât să retransmită mesajele primite. Totuşi, dacă distribuţia are loc în paralel cu alte operaţii de comunicaţie, este de aşteptat ca modelul comutare de circuit să fie mai eficient, utilizând algoritmul banal în care procesorul sursă trimite pe rând mesajele direct la destinaţie. Nici la schimbul complet lucrurile nu sunt diferite. Prezentăm totuşi un algoritm pe hipercub care are avantajul simplităţii.

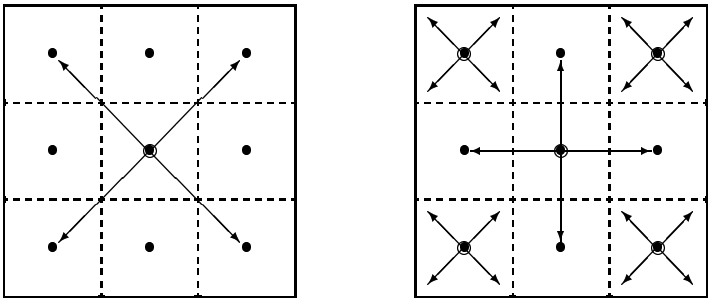


Figura 3.29: Ideea difuzării într-un tor.

Schimb complet pe hipercub. Folosind comutarea de circuit, cea mai banală idee de a realiza schimbul complet este următoarea: vor fi $p-1$ etape în care un anume procesor comunică, pe rând, cu celelalte $p-1$ procesoare, trimiţând fiecăruia mesajul său, şi primind în acelaşi timp mesajul de la procesorul respectiv. Întrebarea este dacă se poate ca toate procesoarele să facă acest lucru în paralel. Altfel zis, dacă în fiecare etapă se pot găsi $p/2$ perechi de procesoare (în fiecare etapă altele) astfel încât cele p căi ce leagă fiecare procesor cu perechea lui să fie arc-disjuncte.

Ideea fericită de formare a perechilor e ca în etapa k, cu $1 \leq k \leq p-1$, procesorul s să trimită mesajul pentru $s \oplus k$; se vede imediat că, în aceeaşi etapă, $s \oplus k$ trimite mesajul pentru s, pentru că $(s \oplus k) \oplus k = s$. De asemenea, în etape diferite, s comunică cu procesoare diferite (funcţia $f(k) = s \oplus k$ este bijectivă !). Mai rămâne problema nesuprapunerii căilor. Să presupunem că fiecare mesaj circulă pe calea principală dintre două procesoare; în etapa k, calea principală între procesoarele s şi $s \oplus k$ depinde numai de $s \oplus s \oplus k = k$; deci biţii de 1 din k decid dimensiunile pe care circulă mesajele, indiferent de sursă. Să demonstrăm că aceste căi sunt arc-disjuncte pentru cazul în care $k = p-1$, adică are toţi biţii de 1; primul arc de pe fiecare cale se află în dimensiunea 0 şi, evident, nu există nici o suprapunere; mulţimea nodurilor aflate la distanţă 1 pe fiecare cale este compusă din toate nodurile hipercubului (fiecare cale a ajuns într-un alt nod); al doilea arc este pe dimensiunea 1; cum se pleacă din noduri diferite, arcele sunt diferite şi ajung în noduri diferite; de asemenea sunt diferite de oricare prim arc de pe o cale, pentru că acesta era în dimensiunea 0; ş.a.m.d., fiecare cale având câte un arc în fiecare dimensiune. Pentru alţi k demonstraţia e similară, cu atât mai mult cu cât căile au mai puţine arce. În figura 3.30 sunt prezentate toate cele 7 etape de comunicaţie, cu căile respective, pentru un hipercub de dimensiune 3. Algoritmul este deci următorul:

ALGORITM 3.9 (Schimb complet în hipercub, model comutare de circuit)

1. **pentru** $k = 1 : p-1$
   1. $ds \leftarrow$ poziţia celui mai puţin semnificativ bit de 1 din k

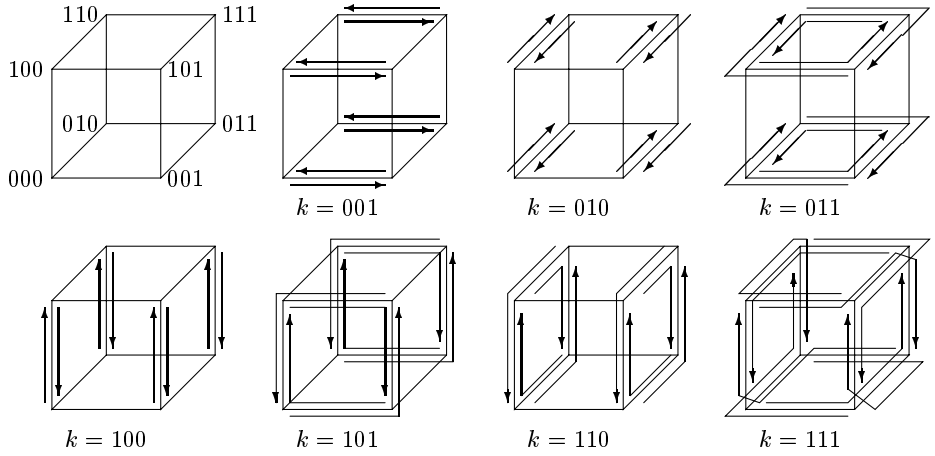


Figura 3.30: Schimb complet în hipercub, model comutare de circuit.

2. $dr \leftarrow$ poziţia celui mai semnificativ bit de 1 din k
3. **send**($M_{\mathrm{id},\mathrm{id}\oplus k}$, ds, id $\oplus\, k$)
4. **recv**($M_{\mathrm{id}\oplus k,\mathrm{id}}$, dr, id $\oplus\, k$)

Trebuie reamintit că algoritmul nu este performant decât dacă mesajele parcurg calea principală (sau alte căi, pentru care arcele de la acelaşi nivel sunt în aceeaşi dimensiune); nu se precizează nicăieri în algoritm acest lucru; calea este decisă de modul de dirijare; cum se face aceasta, vom vorbi ceva mai târziu. Variabilele ds şi dr conţin dimensiunile pe care pleacă, respectiv soseşte, un mesaj în etapa k. Algoritmului i se poate adăuga o operaţie de sincronizare efectuată în finalul fiecărei iteraţii; aceasta previne rămânerea în urmă a unor procesoare, care se poate agrava prin blocarea unor căi; totuşi, pentru a se ajunge aici, e necesar ca unele procesoare să ajungă cu două iteraţii înaintea altora, ceea ce este destul de improbabil.

Timpul necesar execuţiei algoritmului 3.9 este

$$T_{5,H} = \sum_{k=1}^{p-1} (\sigma + dist(0,k)\tau + m\beta) = (p-1)\sigma + \frac{dp}{2}\tau + (p-1)m\beta.$$

Coeficientul lui τ este suma $\sum_{i=1}^{d} i\binom{d}{i}$ pe care am calculat-o deja în problema 3.3.24 şi provine din faptul că un procesor comunică o dată cu fiecare alt procesor, şi sunt $\binom{d}{i}$ procesoare la distanţă i.

3.5 Dirijare (routing)

Dacă un procesor trimite un mesaj unui alt procesor, care nu este vecin cu el, mesajul trebuie să străbată o anume cale pentru a ajunge la destinaţie, trecând pe la procesoarele de pe această cale. Pentru procesoarele intermediare se pune problema alegerii unui canal de comunicaţie pe care să expedieze mesajul, cunoscând numai destinaţia şi, desigur, canalul pe care au primit mesajul; aceasta este problema dirijării (routing).

S-ar părea că ne-am ocupat deja de chestiune, în secţiunea 3.3.1. Acolo însă am presupus că *toate* procesoarele dintr-o arhitectură paralelă au cunoştinţă de operaţia de comunicaţie. Aici însă adoptăm ipoteze diferite: *numai sursa şi destinaţia* ştiu că va avea loc comunicaţia; celelalte procesoare, prin unităţile de comunicaţie de pe fiecare canal, aşteaptă eventuale mesaje din orice direcţie (şi aici e ceva nou, până acum n-am avut nevoie de o astfel de primitivă de interogare); dacă apare un mesaj, unitatea de comunicaţie îl recepţionează şi, pe baza destinaţiei lui, îl trimite mai departe sau îl reţine. Aceste ipoteze sunt mai realiste; în primul rând, pe o arhitectură paralelă se pot executa simultan mai multe programe, pe diverse seturi de procesoare, disjuncte sau nu; în acest caz mesajele pot trece pe la procesoare care nu lucrează la acelaşi program ca sursa şi destinaţia, deci nu au de unde şti de comunicaţia respectivă; în al doilea, în modurile comutare de circuit sau wormhole, unitatea de comunicaţie a unui procesor intermediar trebuie să rezolve rapid problema găsirii canalului pe care să transmită mai departe un mesaj şi, dacă se poate, asincron cu unitatea centrală a nodului; de aceea, practic, unităţile de comunicaţie au asociate funcţii de dirijare, care rezolvă problema.

Definiţii. Asociem unei arhitecturi MIMD cu memorie distribuită un graf orientat, $X = (V, E)$, cu V mulţimea nodurilor (reprezentând procesoarele) şi E mulţimea arcelor (reprezentând canalele de comunicaţie). Funcţia de dirijare $f : E \times V \to E$ asociază, pentru un nod oarecare $v \in V$, unui canal de intrare $c = (u, v) \in E$, un canal de ieşire $e = (v, w) \in E$. Există mai multe atribute posibile ale funcţiilor de dirijare, dintre care enumerăm:

- dirijare *distribuită*: decizia de alegere a căii de ieşire este luată de procesorul intermediar prin care trece mesajul; în acest mod nu se poate cunoaşte drumul deja parcurs de mesaj. Modul de lucru opus este dirijarea *centralizată*, în care un singur nod gestionează toate traseele.
- dirijare *statică*: există o funcţie de dirijare care asociază fiecărui cuplu (sursă, destinaţie) o cale unică. Pot apărea probleme de congestie, dacă traficul nu e omogen.
- dirijare *adaptivă*: funcţia de dirijare permite alegerea între mai multe căi (funcţia se poate modifica în raport cu contextul), atunci când unele sunt deja ocupate, pentru a echilibra traficul. Funcţia de dirijare se poate defini în acest caz astfel: $f : E \times V \times B \to E$, unde B reprezintă stările canalelor—ocupat sau nu.
- dirijare *pe calea cea mai scurtă*: traseul mesajului de la sursă la destinaţie are un număr minim de canale (atenţie, aceasta nu implică unicitate). Funcţia de dirijare poate fi văzută ca o scufundare de dilatare minimă a grafului complet în graful X.

Noţiunea de dirijare apare nu numai în calculatoarele paralele, ci şi în orice reţea eterogenă de calculatoare. De regulă, în reţele, funcţia de dirijare este realizată prin tabele, care asociază canale unor destinaţii; dirijarea este realizată prin program. În calculatoarele paralele sunt interesante funcţiile simple de dirijare, care pot fi implementate hardware, şi pot deci găsi în timp foarte scurt canalul de ieşire.

Se pune acum problema dacă tot ceea ce am făcut în secţiunea 3.3.1 este util. Vom vedea imediat că aproape toate operaţiile descrise acolo global, suportă foarte bine o descriere la nivel *local*. Ne vom ocupa de dirijarea distribuită şi statică. Pe scurt, întrebarea la care vrem să răspundem este: putem construi o funcţie de dirijare astfel încât, dacă procesorul sursă trimite un pachet pe fiecare canal, aceste pachete urmează căi arc-disjuncte până la destinaţie ?

Dirijarea pe inel este foarte simplă: un procesor retransmite spre dreapta mesajele primite dinspre stânga şi invers. Se vede imediat că orice algoritm de comunicare între două procesoare funcţionează corect, indiferent dacă se utilizează o singură cale (ca în algoritmul 3.1), sau amândouă, indiferent dacă se utilizează pipeline sau nu.

Dirijare pe tor. Pentru a transmite pe cele patru căi din figura 3.8a, e suficient ca procesorul sursă să trimită cele patru pachete pe canalele sale de ieşire, iar celelalte procesoare să aplice următorul algoritm de dirijare (notăm cu id_x şi id_y poziţia procesorului curent pe orizontală, respectiv verticală):

ALGORITM 3.10 (dirijare pe tor; procesorul id determină canalul de ieşire *co*, pentru un mesaj venit pe canalul *ci*)

1. recepţionează un mesaj pentru P_{xy}, pe canalul *ci*
2. **dacă** $\mathrm{id} = (x, y)$ **atunci** păstrează mesajul
3. **altfel dacă** $\mathrm{id}_x \neq x$ şi $\mathrm{id}_y \neq y$ **atunci**
    1. **dacă** $ci = \langle \text{stânga}, \text{dreapta}, \text{sus}, \text{jos} \rangle$ **atunci**
        1. $co \leftarrow \langle \text{dreapta}, \text{stânga}, \text{jos}, \text{sus} \rangle$
4. **altfel dacă** $\mathrm{id}_x = x$ **atunci**
    1. **dacă** $ci = \langle \text{stânga}, \text{dreapta}, \text{sus}, \text{jos} \rangle$ **atunci**
        1. $co \leftarrow \langle \text{sus}, \text{jos}, \text{jos}, \text{sus} \rangle$
5. **altfel dacă** $\mathrm{id}_y = y$ **atunci**
    1. **dacă** $ci = \langle \text{stânga}, \text{dreapta}, \text{sus}, \text{jos} \rangle$ **atunci**
        1. $co \leftarrow \langle \text{dreapta}, \text{stânga}, \text{stânga}, \text{dreapta} \rangle$

Unele instrucţiuni **dacă** au în condiţie toate valorile posibile pentru *ci*; variabilei *co* i se atribuie valoarea din poziţia din lista din instrucţiunea sa identică cu poziţia valorii actuale a lui *ci*; de exemplu, dacă în 3.1 avem $ci = \text{sus}$, atunci în 3.1.1 se execută atribuirea $co \leftarrow \text{jos}$.

Desigur că algoritmul este executat în buclă infinită, mai sus fiind descrisă o singură iteraţie.

Din păcate, în cazul în care sursa şi destinaţia sunt pe aceeaşi linie sau coloană, algoritmul 3.10 trebuie îmbunătăţit; în forma aceasta el funcţionează corect, însă

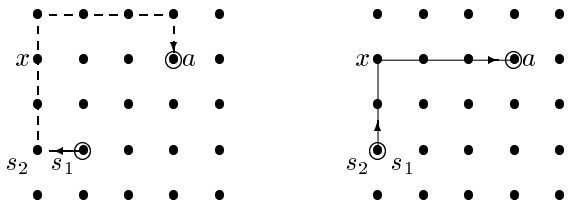


Figura 3.31: Două căi pentru care nu se poate construi o funcţie de dirijare în nodul x, pe tor.

ineficient (două mesaje din patru ar face întâi un tur complet de inel înainte de a schimba direcţia, pe căi ce s-ar suprapune cu primele două). Pentru că algoritmul 3.10 este practic foarte bun, e mai bine ca procesorul sursă să rezolve el problema, netrimiţând decât două mesaje.

Şi tot din păcate, pentru variantele de căi arc-disjuncte din figura 3.8bc, nu se pot construi funcţii de dirijare. Un contraexemplu imediat pentru cazul 3.8b este prezentat în figura 3.31; în primul desen, nodul s_1 trimite un mesaj către a, începând spre stânga; în al doilea, s_2 trimite un mesaj, tot spre a, începând în sus; căile pe care circulă aceste mesaje trec prin nodul x, folosind acelaşi canal de intrare, dar canale de ieşire diferite. Numai că, printr-o funcţie de dirijare, se asociază unui canal de intrare şi unei destinaţii, un singur canal de ieşire. Evident, singura soluţie aici ar fi ca funcţia de dirijare să ţină seama şi de adresa sursei mesajului; în acest fel ne îndepărtăm de modul clasic de a privi dirijarea.

Dirijare pe hipercub. Pe hipercub, ca întotdeauna, soluţiile sunt mai elegante. O primă idee e de a dirija orice pachet pe calea principală. Un procesor oarecare trimite un mesaj pe dimensiunea cu cel mai mic număr de ordine în care adresa sa diferă de cea a destinatarului; deci, co este poziţia bitului de 1 cel mai nesemnificativ din $\mathrm{id} \oplus a$ (P_a fiind destinaţia). De exemplu, dacă nodul 10001 recepţionează un mesaj pentru 01011, indiferent pe ce canal, îl trimite mai departe pe canalul 1. Faptul că nu se ţine seama de canalul de intrare sugerează că nu întotdeauna se pot obţine d căi disjuncte între sursă şi destinaţie. De exemplu, dacă 000 trimite către 110, începând cu canalul 0, deci lui 001, acesta va retrimite înapoi mesajul lui 000, apoi acesta circulând pe ruta 000, 010, 110, deci pe calea începând din 000 pe dimensiunea 1; vor fi numai două căi arc-disjuncte între 000 şi 110.

Defectul acesta este remediat în dirijarea *rotativă*. Dacă un procesor primeşte un mesaj pe ci, atunci îl va expedia pe primul canal aflat într-o dimensiune mai mare, în care adresa sa diferă de cea a destinatarului, sau pe cel în cea mai mică astfel de dimensiune, dacă nu există una mai mare; altfel zis, co este poziţia bitului de 1 din $\mathrm{id} \oplus a$, cea mai aproape prin stânga (ciclic) de ci. De exemplu, dacă 10001 primeşte un mesaj pentru 01011 pe canalul 2, îl va retrimite pe canalul 3; dacă $ci = 4$, atunci $co \leftarrow 0$. Căile sunt întotdeauna arc-disjuncte, dacă sursa trimite mesaje spre aceeaşi

destinaţie pe toate canalele. Dacă 000 trimite spre 110, căile vor fi: prima 000, 001, 011, 111, 110, a doua 000, 010, 110, a treia 000, 100, 110.

Pentru comunicarea dintre procesoarele 0 şi $2^j - 1$ se regăsesc exact căile din algoritmul 3.2; primele j căi sunt calea principală şi rotaţiile ei; pentru celelalte, dacă se porneşte pe dimensiunea $l \geq j$, următorul arc, pornind din nodul 2^l, se află în dimensiunea 0 (nu există biţi diferiţi în poziţii mai mari ca l, în adresele 2^l şi $2^j - 1$), al treilea în dimensiunea 1, etc., deci se urmează calea principală dintre 2^l şi $2^j - 1$. În cazul general, căile dintre sursă şi destinaţie nu sunt însă identice cu cele din observaţia din rezolvarea problemei 3.3.5.

Probleme

P 3.5.1 Scrieţi un algoritm de dirijare pe grilă.

P 3.5.2 Considerând dirijarea rotativă la dreapta, în care co este poziţia bitului de 1 cel mai apropiat spre dreapta (ciclic) de poziţia ci, în $\mathrm{id} \oplus a$, asigură aceasta d căi arc-disjuncte între sursă şi destinaţie ? Care sunt căile între 000 şi 110 ?

P 3.5.3 Propuneţi un algoritm de dirijare pe graful complet şi analizaţi-l.

Capitolul 4

Algoritmi paraleli: concepţie şi apreciere

Acest capitol este dedicat modului în care se studiază un algoritm paralel, de la formularea problemei până la aprecierea performanţelor obţinute. Nu trebuie însă să vă aşteptaţi la un panaceu universal, ci numai la unele instrumente, în general folositoare, dar cu aplicabilitate diferită de la caz la caz.

Formularea problemei de rezolvat se face la fel ca în cazul secvenţial, adăugând acum specificaţiile arhitecturii ţintă. Proiectarea unui algoritm de rezolvare porneşte de obicei de la algoritmii secvenţiali existenţi. Se studiază în ce măsură aceştia oferă un grad de paralelism satisfăcător şi, dacă e cazul, se scrie algoritmul paralel corespondent. Se estimează apoi performanţele algoritmului, după care acesta se implementează şi se testează experimental; desigur, e de dorit ca estimarea să fie cât mai adecvată, pentru a economisi din timpul de calculator necesar testării (încă relativ scump pentru calculatoarele paralele). Dacă algoritmii secvenţiali clasici nu dau satisfacţie, trebuie încercată găsirea unora noi, eventual nu atât de valoroşi secvenţial, dar cu potenţial mare de paralelizare.

De obicei, algoritmii secvenţiali au forme consacrate; apar însă şi o serie de variante ale lor, obţinute printr-o altă *ordonare* a aceloraşi operaţii; ceea ce e important însă, şi comun tuturor variantelor, este ideea unică ascunsă în spatele lor. Ceea ce numim algoritm paralel este tot o formă a aceleiaşi idei, dar în care sunt evidenţiate operaţiile ce pot decurge în paralel; de asemenea, există o multitudine de variante ale unui algoritm paralel, chiar pe o aceeaşi arhitectură; toate aceste forme permit descrieri secvenţiale, de multe ori echivalente. Pentru a face distincţia terminologică necesară, vom numi algoritm secvenţial, aşa cum se utilizează curent, combinaţia idee + descriere secvenţială; la fel, algoritm paralel = idee + descriere paralelă pe o arhitectură precizată (eventual idealizată). Am vrut astfel să subliniem că ideea este esenţialmente comună, deosebirile între secvenţial şi paralel apărând îndeosebi în descriere. În această lucrare vor fi prezentate atât idei noi, născute având ca scop

implementarea paralelă, cât şi descrieri noi ale unor idei vechi. În ambele cazuri se vorbeşte însă despre algoritmi paraleli noi; de multe ori există justificarea: o descriere nouă poate încorpora multă originalitate şi poate da un aspect nebănuit ideii originale. Vom conserva această mică ambiguitate de limbaj, intrată de-acum în uz.

4.1 Performanţele unui algoritm paralel

Vom începe cu etapa finală, aprecierea unui algoritm, pentru că astfel putem fixa mai bine obiectivele de avut în vedere la proiectare. În tot ce urmează, ne vom referi la o unică problemă, rezolvată prin unul sau mai mulţi algoritmi, paraleli şi secvenţiali. Trebuie deci să fie clar că nu se apreciază un algoritm în sine, izolat de context, ci eficacitatea cu care el rezolvă problema în cauză.

4.1.1 Criterii de bază

Timpul de execuţie. Principalul criteriu de apreciere a performanţelor unui algoritm paralel este timpul de execuţie. Vom nota acest timp cu $T(n,p)$, arătând prin aceasta dependenţa sa de cel puţin doi factori; n este dimensiunea problemei, reprezentând într-un anume fel numărul datelor de intrare, de exemplu (cazul cel mai curent), dimensiunea unor matrice sau vectori; p este, ca şi până acum, numărul de procesoare. Vom adăuga acestei notaţii diverşi indici, arătând de obicei algoritmul pentru care este valabil timpul respectiv. Toate celelalte criterii sunt derivate ale timpului de execuţie.

Aprecierea timpului de execuţie se poate face teoretic sau experimental; în ambele cazuri se consideră ca timp de execuţie intervalul dintre momentul în care *primul* procesor începe lucrul şi cel în care *ultimul* procesor termină lucrul.

Pentru un procesor, timpul de lucru este consumat nu numai pentru calculele pe care le efectuează, ci şi pentru comunicaţie (cu alte procesoare sau pentru transferuri între memoria locală şi cea comună); de asemenea, pot exista şi eventuali timpi morţi, necesari sincronizării cu alte procesoare (aşteptarea unor mesaje, a accesului la secvenţe critice). Pentru calcule vom considera, în general, ca unitate de măsură timpul necesar execuţiei unui flop, notat cu α. Pentru timpii de comunicaţie sunt prezentate mai multe modele în capitolul anterior.

În ce priveşte măsurarea experimentală a timpilor de execuţie există probleme de implementare care ţin de arhitectura calculatorului. În general, în arhitecturile MIMD cu memorie distribuită, se atribuie unui procesor sarcina de a trimite mesaje de pornire tuturor celorlalte şi apoi, după terminarea execuţiei părţii din algoritm atribuite lui, de a recepţiona mesaje de terminare de la celelalte procesoare; el va măsura timpul dintre emiterea primului mesaj şi recepţionarea ultimului, o aproximare superioară timpului de execuţie. Sau, soluţie convenabilă şi pentru arhitecturile cu memorie comună, algoritmul începe şi se termină cu câte o sincronizare, timpii iniţial şi final

fiind măsuraţi imediat după acestea. Dacă există un ceas global, fiecare procesor memorează momentul de început şi pe cel de sfârşit.

Menţionăm că, de cele mai multe ori, timpul necesar traficului datelor între o memorie externă şi memoriile locale ale procesoarelor (în cazul memoriei distribuite) sau memoria comună este neglijat. Toate aprecierile timpului de execuţie se fac începând dintr-un moment în care datele se află într-un loc accesibil direct procesoarelor şi până când rezultatele sunt depuse tot acolo. Aceasta este de obicei corect atunci când se compară doi algoritmi paraleli, dar poate fi incorect la comparaţia între un algoritm paralel şi unul secvenţial.

Accelerarea. Modul uzual în care este gândită valoarea unui algoritm paralel poate fi formulat astfel: pe un calculator cu p procesoare va fi acesta mai rapid de p ori, sau de aproape p ori, decât algoritmul secvenţial (executat pe un singur procesor) ? Cu alte cuvinte: se obţine un câştig semnificativ de viteză în rezolvarea problemei dacă se utilizează algoritmul paralel ? Este acest câştig pe măsura puterii de calcul folosite ? Dacă răspunsul este negativ—ceea ce înseamnă fie că algoritmul este profund secvenţial, fie că nu a fost găsită o variantă paralelă bună—atunci cercetarea trebuie continuată. Pe de altă parte, găsirea unui algoritm paralel foarte avantajos, comparat cu varianta sa secvenţială (cum am spus, orice algoritm paralel se poate descrie secvenţial), nu înseamnă şi sfârşitul căutării; deoarece se urmăreşte rezolvarea unei probleme, compararea algoritmului paralel se face cu *cel mai rapid* algoritm secvenţial din clasa celor care rezolvă problema.

Parametrul care cuantifică acest mod de a privi performanţa este *accelerarea* (speed-up, creştere de viteză) algoritmului paralel X, definită prin relaţia

$$S_X(n,p) = T_S(n)/T_X(n,p),$$

în care $T_S(n)$ este timpul de execuţie al celui mai rapid algoritm secvenţial care rezolvă problema respectivă, iar $T_X(n,p)$ timpul de execuţie al algoritmului paralel. Accelerarea depinde, deseori semnificativ, de n şi p; în general, pentru p constant, aceasta creşte o dată cu n (vom vedea ceva mai târziu cum); de asemenea, pentru n constant, creşte o dată cu p, dar fără a depăşi o valoare maximă. În general, accelerarea este mărginită astfel:

$$1 \leq S(n,p) \leq p,$$

valoarea din dreapta fiind considerată ideală. Sunt posibile şi valori mai mici ca 1, de exemplu în cazul în care algoritmul paralel se desfăşoară de fapt secvenţial pe ansamblul celor p procesoare, pierzându-se şi timp suplimentar pentru comunicaţie; cazul $S(n,p) > p$, aparent imposibil, poate apărea în practică; vom exemplifica puţin mai târziu.

Eficienţa. Parametrul cel mai folosit în aprecierea unui algoritm paralel X este *eficienţa*, definită astfel:

$$\varepsilon_X(n,p) = \frac{S_X(n,p)}{p} = \frac{T_S(n)}{pT_X(n,p)}. \tag{4.1}$$

Eficienţa are (aproape) întotdeauna o valoare subunitară; un algoritm este cu atât mai valoros cu cât eficienţa sa este mai aproape de 1. În general, eficienţa creşte o dată cu n şi scade o dată cu p. Vom spune că un algoritm paralel are eficienţa *asimptotic egală cu 1*, dacă, pentru orice p constant, avem

$$\lim_{n\to\infty} \varepsilon(n,p) = 1.$$

Desigur că, pentru un anume algoritm, limita de mai sus poate fi dovedită numai teoretic; rezultatele experimentale o pot doar sugera.

Vom numi *cost* (sau volum de lucru) al unui algoritm paralel mărimea

$$W(n,p) = pT(n,p),$$

reflectând timpul de execuţie cumulat pe cele p procesoare. El poate fi comparat cu costul algoritmului secvenţial; diferenţa $W(n,p) - W(n,1)$ (unde $W(n,1) \equiv T_S(n)$) arată timpul total suplimentar consumat în execuţia paralelă.

Pentru varianta paralelă XP a unui algoritm secvenţial XS, o formulă de acelaşi tip cu (4.1) defineşte *eficienţa relativă* a paralelizării:

$$\varepsilon'_X(n,p) = \frac{T_{XS}(n,p)}{pT_{XP}(n,p)}.$$

Aceasta este utila pentru a aprecia cât de bună este paralelizarea algoritmului, fără a ne interesa cât de bine este rezolvată problema.

În general, vom nota $S(p)$, $\varepsilon(p)$ etc., ignorând dimensiunea problemei, ori de câte ori va fi posibil.

4.1.2 Despre overhead

Definiţie şi clasificare. Am menţionat deja că pentru orice algoritm paralel eficienţa este subunitară; este interesantă studierea cauzelor care contribuie la ineficienţă ($1 - \varepsilon$). Denumirea generică a factorilor care micşorează eficienţa este de *overhead*. Prezentăm în continuare câteva tipuri generale de overhead, valabile pentru toţi algoritmii (se pot defini şi tipuri specifice, în funcţie de natura problemei):

- 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*; aceasta reprezintă o activitate cu totul nouă faţă de cazul secvenţial, deci 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 al arhitecturii paralele. Exemplul cel mai simplu este cel al deciziilor. Există, în plus, anumite operaţii suplimentare datorate implementării software a algoritmului—decizii în funcţie de adresa procesorului, aritmetică întreagă repetată pe mai multe procesoare etc.

Estimarea overhead-ului. Dacă estimarea teoretică a fiecărui tip de overhead poate fi făcută relativ uşor, în schimb determinarea experimentală nu este deloc simplă; aceasta e necesară deseori, pentru a confirma estimările teoretice, care nu pot ţine cont de toţi factorii posibili. Prezentăm în continuare un mod de a face determinarea overhead-urilor. Pentru a simplifica notaţia vom omite în continuare dependenţa de n şi de p, presupuse fixe, a diverselor mărimi definite mai sus.

Notăm cu T_{nc} timpul de execuţie al unui algoritm în care nu se efectuează comunicaţiile şi folosim un model simplificat pentru expresia lui T_p—timpul de execuţie pentru un algoritm paralel:

$$T_p = T_a + T_c + T_l,$$

unde T_a este timpul pentru operaţii aritmetice (în cazul unei încărcări echilibrate a procesoarelor, atunci $T_a \approx T_s/p$ unde T_s este timpul secvenţial—de execuţie pe un singur procesor), $T_c = T_p - T_{nc}$ este timpul consumat pentru comunicaţie, iar $T_l = T_{nc} - T_a$ timpul pierdut în execuţia paralelă din cauza încărcării inegale a procesoarelor, specificului implementării software, etc. Astfel se pot calcula două tipuri de overhead (pentru detalii vezi [12]):

1. overhead datorat comunicaţiei: $O_c = 1 - T_{nc}/T_p$.
   De fapt, deoarece uneori protocolul de comunicaţie implică sincronizare, o parte nemăsurabilă (pe care o neglijăm) a acestui overhead se datorează timpului mort de aşteptare a sincronizării. De asemenea, alţi timpi morţi pot apărea atunci când un procesor aşteaptă de la celelalte date de prelucrat şi nu mai are nici o operaţie de efectuat; în acest caz, cauza overhead-ului este în primul rând algoritmică.

2. overhead datorat încărcării inegale, software, algoritmic: $O_l = (T_{nc} - \frac{T_s}{p})/T_p$.

4.1.3 Legea lui Amdahl

Legea lui Amdahl. Să analizăm puţin comportarea generală a unui algoritm în funcţie de numărul de procesoare p, făcând o ipoteză simplificatoare; presupunem că algoritmul poate fi împărţit în două părţi: una pur secvenţială, al cărei cost este W_1, şi o alta perfect paralelă, indiferent de numărul de procesoare, al cărei cost este W_P. Atunci timpul de execuţie este

$$T(p) \equiv T_1 + T_P = W_1 + W_P/p,$$

unde $T_1 = W_1$ şi $T_P = W_P/p$, doar operaţiile din partea perfect paralelă fiind repartizate între procesoare. Timpul de execuţie are aspectul din figura 4.1a. Accelerarea

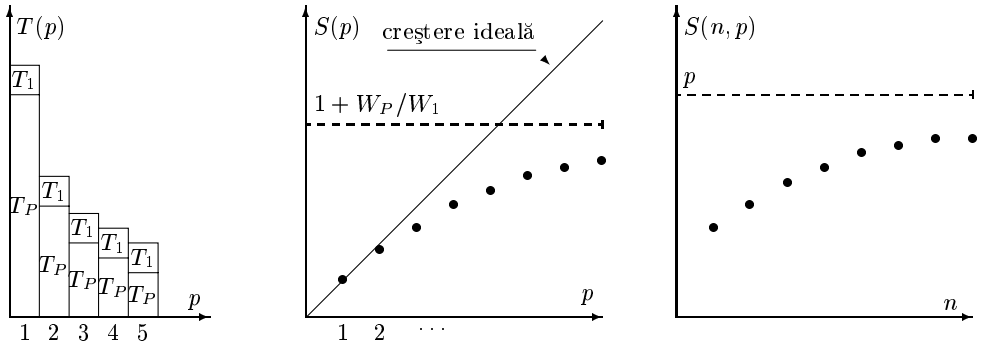


Figura 4.1: Evoluţii specifice pentru: (a) Timpul de execuţie al unui algoritm paralel funcţie de numărul de procesoare p; (b) Accelerarea funcţie de p; (c) Accelerarea funcţie de dimensiunea problemei.

are expresia

$$S(p) = \frac{W_1 + W_P}{W_1 + W_P/p} < 1 + \frac{W_P}{W_1}, \tag{4.2}$$

cunoscută sub numele de *legea lui Amdahl*, şi o evoluţie de genul celei din figura 4.1b. Aşadar, accelerarea are o limită inerentă; din fericire, pentru mulţi algoritmi W_1 este modest.

Discuţie. În cele de mai sus am presupus implicit că raportul W_1/W_P nu depinde de dimensiunea problemei, ceea ce nu se întâmplă de obicei în practică. Să presupunem acum că W_1 depinde de n altfel decât W_P, anume raportul W_p/W_1 crescând o dată cu n; de exemplu, $W_1 = O(n)$, dar $W_P = O(n^2)$; deci, cu cât creşte n, cu atât valoarea maximă a creşterii de viteză din (4.2) este mai mare. Aşadar, pentru p fixat, o alură tipică a creşterii de viteză în funcţie de dimensiunea problemei este cea din figura 4.1c; o evoluţie asemănătoare are şi eficienţa, valoarea maximă p fiind înlocuită cu 1. Dacă eficienţa algoritmului este asimptotic egală cu 1, atunci accelerarea tinde către orizontala de ordonată p.

Să definim acum o altă mărime destul de utilizată, *performanţa temporală*, $R(n,p) = 1/T(n,p)$; ea se foloseşte de multe ori pentru că este cu atât mai mare cu cât algoritmul se execută mai rapid, ceea ce e mai aproape de obiceiurile noastre de a aprecia performanţa. De asemenea, pentru n fixat, îi este asociată o altă formă a legii lui Amdahl, anume

$$R(p) = \frac{R_\infty}{1 + p_{1/2}/p}, \tag{4.3}$$

în care $R_\infty = 1/W_1$ este performanţa asimptotică, iar $p_{1/2} = W_P/W_1$ numărul de procesoare pentru care se atinge jumătate din performanţa asimptotică (înlocuiţi

aceste valori în (4.3), pentru verificare). Şi pentru accelerare se poate scrie o relaţie de tipul (4.3) (calculaţi valorile parametrilor !).

Superliniaritate. Cum se poate ajunge la o accelerare mai mare decât p ? Algoritmii cu această proprietate, deci cu $T(n,p) < T(n,1)/p$, sunt impropriu numiţi *superliniari*; existenţa lor contrazice intuiţia şi, desigur, legea lui Amdahl (în (4.2), ar trebui ca $W_1 < 0$, ceea ce e imposibil, şi $W_1 + W_P/p > 0$). Până acum ne-am referit numai la timpi de execuţie, atunci când a fost vorba de evaluarea unui algoritm. Să ne referim acum la numărul de operaţii; în principiu, n-ar trebui să existe nici o diferenţă; mai sus am definit timpul de execuţie α al unui flop ca pe o mărime fixă (şi aşa o vom face şi după acest paragraf).

Superliniaritatea poate proveni din două motive: fie numărul total de operaţii (costul), în paralel, este mai mic decât cel secvenţial; fie o operaţie se execută mai rapid în contextul paralel, faţă de cel secvenţial. Ambele par obscure şi greu de imaginat. Pentru primul, nu vom da detalii, lăsând tema deschisă; de altfel, există destule controverse; ca exemplu (destul de dubios), imaginaţi-vă simularea unui mediu de programare paralel: secvenţial, paralelismul trebuie simulat, în paralel el este parţial real (pe un procesor se pot totuşi simula mai multe), deci e simulat mai rapid. Pentru al doilea însă, lucrurile sunt mai clare; timpul de execuţie al unui algoritm nu depinde numai de numărul de flop, ci şi, de exemplu, de numărul de referinţe la memorie (pentru citirea datelor şi a instrucţiunilor); presupunând că un nod are o arhitectură cu memorie ierarhică, precum cea din figura 1.5, timpul necesar unui acces la memorie este mai mic dacă data se află în memoria cache; ori, atunci când un algoritm se execută secvenţial, există o singură memorie cache; la execuţia paralelă, pentru aceeaşi cantitate totală de date, memoria cache este de p ori mai mare; deci, mult mai multe date vor avea loc în memoria cache, şi astfel un acces la memorie va dura în medie mai puţin. Aceleaşi argumente sunt valabile pentru nivelul ierarhic următor de memorie, adică atunci când se pune problema transferurilor (repetate) între memoria principală şi memoria externă (în contextul utilizării memoriei virtuale).

4.1.4 Alte criterii

Complexitate paralelă. Încercăm să ne apropiem acum de caracterizări mai teoretice ale unui algoritm. În general, vom înţelege prin *complexitate* a unui algoritm numărul de operaţii necesare execuţiei lui; pentru un algoritm paralel, este vorba de numărul de operaţii care pot fi executate de un singur procesor, între începutul şi sfârşitul execuţiei algoritmului; deci, mai multe operaţii executate simultan de mai multe procesoare sunt contorizate (după cum e normal), ca una singură a algoritmului paralel; de asemenea, sunt echivalaţi în număr de operaţii eventualii timpi morţi ai unui procesor. Vom face distincţie între complexitatea aritmetică şi cea a comunicaţiei, cea de-a doua referindu-se la numărul de elemente comunicate (în aceleaşi condiţii ca pentru numărul de operaţii, după cum am făcut-o deja în capitolul precedent). Deoarece între complexitate şi timpul de execuţie al algoritmului este, în principiu, o relaţie de proporţionalitate, cele două noţiuni vor fi câteodată (dar cât

mai rar) folosite una în locul celeilalte.

Fiecare problemă are un număr minim de operaţii necesar rezolvării ei în paralel, cu un număr stabilit de procesoare, fie că există, fie că nu există un algoritm care să atingă această limită. Vom numi complexitate paralelă a *problemei*, numărul de operaţii necesar rezolvării ei, cu un număr oricât de mare de procesoare şi neglijând comunicaţia (folosind deci toate mijloacele posibile); notaţia consacrată este $\Omega(f(n))$, unde $f(n)$ este o funcţie simplă; $\Omega(\log n)$, de exemplu, poate însemna la fel de bine $\log n$ sau $3\log n$; important este ordinul de mărime, şi nu eventualii coeficienţi. Complexitatea unui *algoritm* a fost definită mai sus şi se notează cu $O(f(n))$; evident, ea este mai mare sau egală cu complexitatea problemei.

Vom folosi într-un mod mai neriguros, dar sugestiv notaţia $O(\cdot)$. De exemplu, deşi $O(n)$ şi $O(n/p)$ pot avea teoretic aceeaşi expresie, vom sublinia astfel influenţa lui p asupra complexităţii; sau, deşi $O(n)$ şi $O(n+\log n)$ sunt identice, a doua notaţie evidenţiază mărimea termenului al doilea dintr-o sumă, pentru a compara sume în care primul termen e identic.

Algoritmi optimali, algoritmi eficienţi. Fie $polylog(n) = \bigcup_{k>0} O(\log^k n)$. Fie S o problemă pentru care cel mai rapid algoritm secvenţial se execută în timp $T(n)$.

DEFINIŢIA 4.1 Un algoritm paralel care rezolvă S şi care se execută în timpul $t(n)$ pe $p(n)$ procesoare se numeşte *optimal* dacă:
a) $t(n) = polylog(n)$, şi
b) volumul de lucru $W(n) = p(n) \cdot t(n)$ este $O(T(n))$.

Cu alte cuvinte, algoritmul este optimal dacă timpul său de execuţie este logaritmic şi dacă volumul său de lucru este de acelaşi ordin de mărime cu cel al algoritmului secvenţial (nu se execută semnificativ mai multe operaţii în algoritmul paralel, pe ansamblul procesoarelor, faţă de cel secvenţial). După cum se vede, notând $p(n)$ am lăsat să se înţeleagă că pentru atingerea optimalităţii, numărul de procesoare poate depinde de dimensiunea problemei (aceasta e specific unei abordări teoretice; practic, desigur că p este cel oferit de arhitectura paralelă).

DEFINIŢIA 4.2 Un algoritm paralel este *eficient* dacă condiţia b) din definiţia 4.1 se înlocuieşte cu:
b') volumul de lucru este $W(n) = T(n) \cdot polylog(n)$.

Un algoritm eficient are un înalt grad de paralelism, dar accelerarea nu este atinsă în mod optimal, ci cu preţul unor operaţii suplimentare, ceea ce se reflectă într-un cost de un factor logaritmic de ori mai mare decât cel secvenţial.

Alte criterii în afara complexităţii. În toată această secţiune nu ne-am referit decât la un singur criteriu de apreciere (şi comparaţie) a algoritmilor paraleli: timpul de execuţie (sau, echivalent, numărul de operaţii, complexitatea). Nu trebuie uitate alte criterii, utilizate la fel şi în algoritmica secvenţială; poate fi vorba despre precizia cu care se obţin soluţiile, stabilitatea numerică a algoritmului, memoria necesară etc.

În continuare, vor fi menţionate şi acestea; decizia alegerii unui algoritm trebuie să depindă şi de ele.

4.2 Tehnici generale de paralelizare

În drumul de la un algoritm descris în formă secvenţială, până la scrierea algoritmului pentru un anume calculator paralel, trebuie parcurse întotdeauna următoarele etape:

- gruparea în taskuri a operaţiilor ce trebuie executate.
- identificarea ordinii în care se pot executa taskurile, precum şi a taskurilor ce se pot executa în paralel; de un mare ajutor în această etapă este construirea grafului de precedenţă a taskurilor.
- planificarea (scheduling) execuţiei taskurilor, adică stabilirea unui mod de a descrie procesorul pe care se execută fiecare task.

Uneori, în prezentarea algoritmilor, etapele nu sunt separate în mod foarte explicit, în special a doua de cea de-a treia; de asemenea, de multe ori, nu se prezintă constrângerile de precedenţă a taskurilor. Vom expune, totuşi, pe larg, noţiunile generale legate de etapele de mai sus, pentru a crea un cadru sistematic de abordare a problemei paralelizării unui algoritm. (Trebuie precizat că există în literatură numeroase articole ce conţin abordări brutale ale problemelor, justificate finalmente prin buna eficienţă experimentală a algoritmului studiat. Vom folosi şi noi această tactică, pentru a nu lungi, de multe ori inutil, expunerea.)

În plus faţă de aceste etape, mai apar probleme legate de comunicaţie: trecerea de la comunicarea între taskuri, operaţie logică, la comunicarea între procesoare, operaţie fizică.

4.2.1 Partiţionarea în taskuri

Task. Un task este definit ca o unitate indivizibilă de operaţii, specificată numai în termeni ai comportării sale exterioare (intrări, ieşiri, timp de execuţie etc.). Noţiunea de operaţie este elementară şi nu va fi definită riguros; operaţiile luate în considerare ca durată—mai ales în algoritmii de calcul numeric—sunt cele aritmetice (adunare, înmulţire etc.).

Operaţiile din interiorul unui task se execută secvenţial.

Formarea taskurilor. Gruparea operaţiilor în taskuri este o problemă delicată; nu există reţete universale pentru a face aceasta. Sunt o serie de criterii care pot ajuta în formarea taskurilor, dintre care unele euristice; de multe ori validarea se face experimental, alegându-se din mai multe variante.

Un prim criteriu se referă la ordinea în care se pot executa taskurile: părţile eminamente secvenţiale vor fi grupate în acelaşi task, în timp ce operaţii ce se pot

executa în paralel se vor introduce, în general, în taskuri diferite. Mai multe despre aceasta în secţiunea următoare.

Arhitectura unui procesor din componenţa calculatorului paralel poate influenţa şi ea; dacă procesorul este vectorial, taskurile vor fi alese astfel încât să cuprindă cât mai multe operaţii cu vectori; dacă procesorul are memorie ierarhică, se vor crea taskuri pentru care datele să aibă dimensiuni apropiate de cele ale memoriei rapide şi să fie folosite cât mai intens.

În fine, de mare însemnătate este ponderea comunicaţiilor în raport cu calculele în algoritmul ce se implementează; în general, trebuie minimizate cantitatea de informaţie comunicată şi, eventual, numărul de mesaje. Trebuie luată în considerare eficienţa comunicaţiilor în raport cu calculele pe arhitectura ţintă. De asemenea, se vor utiliza la maximum primitivele de comunicaţie oferite de către calculator şi sistemul de operare.

BLAS. Pentru algoritmii de calcul numeric, filosofia BLAS (Basic Linear Algebra Subprograms) este foarte utilă şi flexibilă în alegerea taskurilor; ideea fundamentală este de a izola câteva rutine de bază—care implementează foarte eficient operaţiile cele mai curente, de exemplu înmulţirea matrice-vector sau matrice-matrice, sau rezolvarea de sisteme liniare triunghiulare—folosite apoi în descrierea algoritmilor; taskurilor vor fi compuse atunci din apeluri la astfel de rutine, şi nu din operaţii elementare.

În 1990, Dongarra et al [10] au propus rutinele de bază pentru nivelul 3 BLAS. Acestea conţin operaţii matrice-matrice, de genul $C = \alpha AB + \beta C$, care implică $O(n^3)$ flop, şi pentru care datele au dimensiune $O(n^2)$; deci, rutinele de nivel 3 sunt dense în calcule, în raport cu comunicaţia (sau cu transferurile între memoria comună şi cele locale). Folosirea, în interiorul taskurilor, a operaţiilor la nivel de bloc de matrice, pe lângă faptul că permite o mai mare claritate a algoritmului, oferă posibilitatea alegerii dimensiunilor blocurilor în concordanţă cu posibilităţile oferite de arhitectura ţintă. În capitolul 6, vor fi prezentaţi şi algoritmi la nivel de bloc.

Granularitate. Noţiunea fundamentală în ce priveşte dimensiunea taskurilor este cea a *granularităţii* algoritmilor. Un acelaşi algoritm poate fi scris folosind operaţii la nivel de element, caz în care este denumit cu granularitate *fină* (fine grain) sau la nivel de blocuri, fiind denumit cu granularitate *mare* (coarse grain). În primul caz taskurile au dimensiuni mici, fiind formate din câteva operaţii, în al doilea dimensiunile sunt mari. Subliniem încă o dată importanţa caracteristicilor calculatorului ţintă; de exemplu, pentru calculatoarele MIMD cu memorie distribuită uzuale, algoritmii cei mai eficienţi au, în general, granularitate mare; motivul principal este structura ierarhică a memoriei, care favorizează operaţiile cu blocuri de matrice; de asemenea, un alt motiv este faptul că o aceeaşi cantitate de date se comunică mai repede în puţine mesaje de lungime mare, decât în mai multe de lungime mică.

4.2.2 Grafuri de precedenţă

Noţiunea de graf de precedenţă a taskurilor—introdusă în [8]—modelează dependenţele dintre taskuri, la nivel de date de intrare-ieşire.

Definiţii. Datele de intrare ale unui task T_j pot fi chiar date de intrare ale întregului algoritm sau pot proveni din datele de ieşire ale altor taskuri. În primul caz, T_j poate fi executat oricând; în al doilea, T_j trebuie să aştepte execuţia taskurilor care îi furnizează datele de intrare. Deci, ordinea în care se execută taskurile trebuie să respecte anumite constrângeri de precedenţă, impuse de circulaţia datelor. Relaţia de precedenţă este notată cu $\prec$; dacă, pentru taskurile T_i, T_j, $T_i \prec T_j$, atunci operaţiile din T_j folosesc date prelucrate de T_i (şi apoi, eventual, şi de alte taskuri); deci, execuţia taskului T_j poate începe numai după terminarea execuţiei taskului T_i. În graful de precedenţă a taskurilor, fiecărui nod îi este asociat un task şi, implicit, un număr reprezentând durata de execuţie a acestuia; arcele (orientate) reprezintă constrângerile de precedenţă; dacă $T_i \prec T_j$, atunci există o cale de la T_i la T_j. Acest graf este orientat şi aciclic (DAG - Directed Acyclic Graph); existenţa unui ciclu ar însemna o relaţie de genul $T_i \prec T_i$, evident absurdă.

Un DAG seamănă cu un arbore; există totuşi diferenţe importante. Un DAG are mai multe "rădăcini", adică taskuri fără predecesor; acestea folosesc exclusiv date de intrare ale algoritmului. De asemenea, un nod poate avea mai mulţi taţi. Ieşirile taskurilor frunze sunt ieşirile întregului algoritm.

Vom prezenta în continuare câteva definiţii privind DAG şi, în particular, grafurile de precedenţă; pentru simplitate, vom presupune implicit că duratele de execuţie ale tuturor taskurilor sunt egale (pentru mulţi algoritmi aceasta este sau adevărat sau o bună aproximaţie). Multe dintre definiţii au un echivalent asemănător pentru arbori.

- *lungimea* unei căi este suma timpilor de execuţie asociaţi nodurilor de pe calea respectivă.
- *nivelul* se defineşte astfel: nodurile terminale (care nu au nici un succesor) au nivel 0; nivelul $l(v)$ al unui nod v este $l(v) = \max_i l(v_i) + 1$, unde v_i sunt succesorii imediaţi (fiii) ai lui v. Uneori se asociază nivelul 0 nodurilor iniţiale (care nu au nici un predecesor) şi, în relaţia de mai sus, v_i sunt predecesorii imediaţi ("taţii") ai nodului v.
- *înălţimea* (adâncimea) unui DAG este lungimea celei mai lungi căi.
- *lăţimea* unui DAG este $\max |L_k|$, unde L_k este mulţimea nodurilor având nivelul k, iar $|L_k|$ este cardinalul acestei mulţimi.

Reducerea grafului de precedenţă. Dacă un graf de precedenţă ar cuprinde arce pentru toate perechile de noduri care sunt în relaţie de precedenţă, atunci acest graf ar fi foarte dificil de folosit. De aceea, în măsura în care este posibil, din graf trebuie eliminate arcele care nu conţin informaţie necesară. Această operaţie se face de multe ori în mod natural, în analiza unui algoritm, când nu se studiază decât succesorii imediaţi ai unui task (cei care folosesc direct datele de ieşire ale acelui task) şi nu se reţin decât relaţiile de precedenţă cu aceştia. O regulă de eliminare a arcelor inutile se bazează pe tranzitivitatea relaţiei de precedenţă şi pe o idee cât se poate de evidentă; dacă există o cale de la T_i la T_j trecând prin cel puţin un alt nod T_k, atunci arcul de la T_i la T_j se poate elimina; atenţie, dacă există două căi între T_i şi T_j, amândouă conţinând încă cel puţin câte un alt nod, nu se poate elimina cea mai

scurtă dintre ele sau fie şi numai un arc de pe aceasta.

Complexitatea şi graful de precedenţă. Cu ajutorul grafului de precedenţă asociat unei partiţionări în taskuri a operaţiilor dintr-un algoritm se poate determina paralelismul intrinsec al partiţionării. Presupunând disponibil un număr suficient de mare de procesoare şi ignorând comunicaţiile între taskuri, conflictele de acces la memoria comună şi orice alte întârzieri, atunci timpul cel mai scurt în care se poate executa algoritmul este dat de înălţimea grafului. Într-adevăr, taskurile de pe orice cale trebuie executate secvenţial, iar înălţimea este dată de cea mai lungă cale. Pentru a obţine paralelismul intrinsec (teoretic) al algoritmului, se alege partiţionarea în care fiecare operaţie elementară constituie un task. Înălţimea grafului este, în acest caz, *complexitatea paralelă* a algoritmului.

Lăţimea grafului dă informaţii despre numărul necesar de procesoare pentru a atinge timpul cel mai scurt de execuţie. Între taskurile de pe acelaşi nivel nu există constrângeri de precedenţă (dacă $T_i \prec T_j$, atunci $l(T_i) > l(T_j)$); deci toate taskurile de pe acelaşi nivel se pot executa în paralel. Deci, numărul de procesoare necesar pentru obţinerea timpului minim de execuţie este mai mic sau egal cu lăţimea grafului. Dacă timpii de execuţie ai taskurilor nu sunt egali, atunci problema numărului necesar de procesoare se complică, fiind nevoie de altă definiţie pentru lăţimea grafului.

În final subliniem încă o dată că noţiunea de graf de precedenţă este asociată unei partiţionări în taskuri, şi nu unui algoritm. Deci, fără o alegere adecvată a taskurilor, graful de precedenţă în sine nu poate aduce singur bunele performanţe.

4.2.3 Planificarea taskurilor

Prin definiţie, fiind indivizibil, un task se execută de către un singur procesor. Se pune imediat întrebarea: pe ce procesor se va executa un anume task ?

DEFINIŢIA 4.3 Dată fiind o arhitectură cu p procesoare notate P_i, cu $i = 0 \ldots p-1$, *problema planificării taskurilor* este aceea de a găsi o funcţie s definită pe mulţimea taskurilor, cu valori în mulţimea procesoarelor, semnificaţia relaţiei $s(T_j) = P_i$ fiind că taskul T_j se va executa pe procesorul P_i.

În construirea funcţiei de planificare apar numeroase condiţii legate de minimizarea timpului de execuţie, a comunicaţiilor între procesoare şi, evident, de respectarea constrângerilor de precedenţă.

Tipuri de planificare. Planificarea poate fi de două feluri:

- *statică*, în care funcţia s este cunoscută înaintea începerii execuţiei algoritmului.
- *dinamică*, în care funcţia s se construieşte pe măsură ce algoritmul se execută, atribuindu-se taskuri spre execuţie unor procesoare libere. Nu se poate preciza, la începutul execuţiei, pe ce procesor se va executa un anume task.

Planificarea dinamică este mai flexibilă, ea fiind recomandată în condiţiile unui sistem de operare concurent; de asemenea, este unica viabilă în cazul defectării unui procesor (în acest din urmă caz execuţia este blocată în cazul unei planificări statice). Pe de altă parte, planificarea dinamică este consumatoare de timp suplimentar, datorită programului care determină funcţia s; cu cât granularitatea este mai fină, cu atât costul planificării este mai important. În cazul arhitecturilor cu memorie partajată se foloseşte de obicei planificarea dinamică, iar pentru cele cu memorie distribuită planificarea statică. Motivul este legat de comunicaţie: în primul caz procesoarele sunt egal distanţate între ele, în timp ce în al doilea distanţele diferă; de exemplu se poate găsi o planificare statică în care comunicaţiile să fie numai între vecini; cu o planificare dinamică, nu ar exista garanţia eficienţei comunicaţiei. Totuşi, o situaţie în care planificarea dinamică este evident mai utilă este cea în care durata taskurilor nu poate fi estimată dinainte; în acest caz, la o planificare statică pot apărea timpi morţi mari. În această lucrare vom folosi exclusiv planificarea statică, ea reliefând mai bine paralelismul unui algoritm; de altfel, e mai uşor de obţinut o planificare dinamică din una statică, decât invers; în plus, în majoritatea cazurilor, timpul de execuţie al taskurilor va putea fi aproximat destul de bine.

Planificarea după listă. Într-un mod devenit clasic, problema planificării se reduce la problema construirii unei liste cuprinzând toate taskurile; apoi se alocă taskurile, în ordinea din listă, procesoarelor libere, cu condiţia de a nu se planifica simultan decât taskuri independente (din punct de vedere al precedenţei).

Prin reformulare, problema nu se simplifică deloc; din păcate ea este NP-completă pentru mai mult de 2 procesoare (nerezolvabilă în timp polinomial). Există numeroşi algoritmi de planificare suboptimală—care determină rapid funcţia de planificare s, dar care nu asigură timp minim de execuţie al algoritmului planificat; se pot obţine rezultate foarte bune pentru anumiţi algoritmi, dar slabe pentru alţii.

Nici pentru timpi egali de execuţie a taskurilor problema planificării nu e simplă. Construirea listei, de la sfârşit spre început, în ordinea nivelelor, este o euristică foarte folosită, dar care nu duce obligatoriu la optim. Deci, ultimele în listă sunt nodurile de pe nivelul 0, imediat înaintea lor cele de pe nivelul 1 etc. Pentru a ordona nodurile de pe acelaşi nivel se foloseşte, de exemplu, criteriul numărului de succesori direcţi: primul în listă este pus taskul cu cei mai mulţi succesori direcţi. Aceste criterii de ordonare sunt foarte intuitive; ordinea inversă a nivelelor e sugerată de relaţiile de precedenţă, automat respectate în acest caz; e bine ca un task cu mai mulţi succesori să fie executat cât mai repede posibil, pentru a da şi acestora posibilitatea de execuţie. Nu vom da alte detalii despre algoritmii de planificare, ci vom prezenta un exemplu simplu care sugerează dificultatea problemei.

Exemplu. Să considerăm graful de precedenţă a taskurilor din figura 4.2, în care cele şapte taskuri au durate egale de execuţie (egale cu unitatea, pentru simplitate); taskul 7 are nivelul 0, taskurile 4, 5, 6 au nivelul 1, iar taskurile 1, 2, 3 au nivelul 2. Vom încerca planificarea taskurilor din acest graf pe o arhitectură cu două procesoare. Dacă utilizăm algoritmul sugerat mai sus, nu avem nici o prioritate între taskurile 1, 2, 3, sau între taskurile 4, 5, 6; le vom ordona deci la întâmplare. Lista 1234567

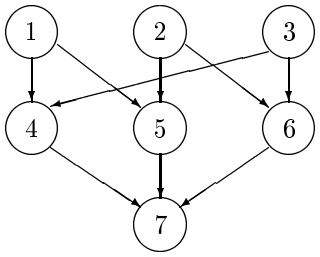


Figura 4.2: Un exemplu de graf de precedenţă.

a			**b**			**c**	
P_0	P_1		P_0	P_1		P_0	P_1
1	2		1	3		1	2
3			2	4		3	5
4	5		5	6		4	6
6			7			7	
7							

Tabelul 4.1: Planificări pe 2 procesoare pentru graful de precedenţă din figura 4.2.

conduce la planificarea din tabelul 4.1a, în care coloana pe care se află un task indică procesorul pe care se va executa acesta, iar linia momentul de timp la care va avea loc execuţia; planificarea se construieşte astfel: în prima etapă se execută taskurile 1 şi 2; în a doua, 3, dar 4 nu, pentru că e succesor al lui 3 în graful de precedenţă; deci 4 se execută în etapa a treia, împreună cu 5; în etapa a patra taskul 6, dar nu şi 7, care este succesor al lui 6; deci taskul 7 rămâne să se execute în etapa a cincea; timpul de execuţie al întregului algoritm este deci 5. Se vede imediat că ordinile 1324567 sau 1235467, care conduc la planificările din tabelele 4.1b şi 4.1c, au timpul de execuţie 4, care este, de altfel, cel optim. Dar dacă pe un astfel de graf sunt puţine variante posibile, pe unul cu mii (milioane) de noduri, planificarea nu mai e deloc banală.

Presupunând că execuţia algoritmului paralel are loc exact după planificările din tabelul 4.1, un mod curent de a prezenta situaţia ocupării procesoarelor este diagrama Gantt. În figura 4.3 sunt prezentate astfel de diagrame pentru primele două planificări din tabel; pe abscisă este reprezentat timpul (de multe ori omis, însă); zonele haşurate reprezintă perioadele în care procesoarele sunt ocupate, iar cele albe, timpii morţi.

Compromisul eficienţă/timp de planificare. În ce priveşte efortul de calcul efectuat în vederea obţinerii unei bune planificări trebuie făcută o precizare de multe ori neglijată. Pentru un algoritm care va fi executat de puţine ori, e suficientă şi o planificare mediocră, dar obţinută rapid. În schimb pentru un algoritm pentru care se estimează un număr mare de execuţii merită făcută investiţia iniţială a căutării unei

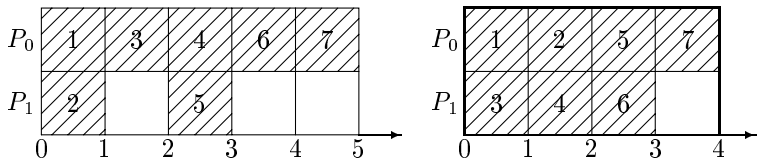


Figura 4.3: Diagrame Gantt pentru planificările din tabelul 4.1ab.

planificări apropiate de optim; aceasta cu atât mai mult cu cât execuţia algoritmului se face pe un calculator paralel, unde timpul de calcul este scump, în timp ce planificarea se poate face pe orice calculator.

Principiul planificării (Brent). Am vorbit în secţiunea precedentă despre paralelismul intrinsec al unui algoritm (sau al unei partiţionări în taskuri asociată algoritmului). Considerăm cazul în care toate taskurile au durată 1. Să presupunem că pentru a obţine timpul minim de execuţie t sunt necesare p_m procesoare. Care va fi timpul de execuţie atunci când sunt disponibile doar $p < p_m$ procesoare ? (De observat că acesta este cazul curent, numărul de procesoare al calculatorului ţintă neavând legătură cu problema de rezolvat.)

Răspunsul imediat—cunoscut şi sub numele de *principiul planificării al lui Brent*—este că timpul de execuţie va fi de cel mult $\lceil p_m/p \rceil t$. Într-adevăr la un moment de timp oarecare se execută cel mult p_m taskuri, care sunt independente; folosind doar p procesoare, cele p_m taskuri se pot planifica oricum pe aceste procesoare, execuţia lor durând maximum $\lceil p_m/p \rceil$; cum sunt t etape (corespunzând celor t niveluri din graful de precedentă) în execuţia algoritmului pe p_m procesoare, se obţine rezultatul de mai sus.

Rafinând puţin prezentarea, să presupunem că pentru execuţia etapei i sunt necesare x_i procesoare, deci $p_m = \max_i x_i$. Cu p procesoare timpul de execuţie al pasului i va fi $\lceil x_i/p \rceil \leq x_i/p + 1$ şi deci timpul total va fi mai mic decât $\lceil x/p \rceil + t$, unde $x = \sum_i x_i$.

Pe de altă parte, în ambele cazuri, o limită inferioară a timpului de execuţie pe p procesoare este $t - 1 + \lceil p_m/p \rceil$; aceasta se obţine când, în algoritmul cu paralelism maxim, într-o singură etapă se folosesc efectiv cele p_m procesoare (o astfel de etapă trebuie să existe, altfel nu se justifică numărul de procesoare), iar în restul mai puţin de p procesoare.

Vom ilustra principiul lui Brent în secţiunea următoare.

Efectul repartizării datelor. În arhitecturile cu memorie distribuită, la algoritmii de calcul numeric, de multe ori planificarea porneşte de la o repartizare iniţială a datelor în aşa fel încât fiecare procesor să lucreze cât mai mult cu datele pe care le are (deci să se minimizeze comunicaţia) şi încărcarea procesoarelor să fie echilibrată (overhead-ul de încărcare inegală să fie cât mai mic). Această abordare, care se bazează pe doar câteva variante—între care este uşor de ales—este facilitată de

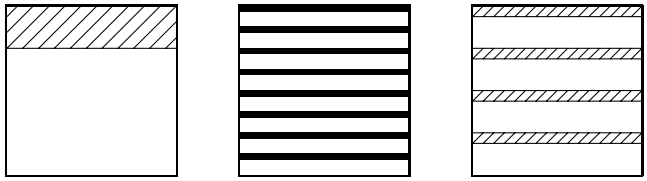


Figura 4.4: Elementele matricei A locale procesorului P_0 în repartizările: (a) bloc linii; (b) ciclic pe linii; (c) bloc ciclic pe linii.

aspectul algoritmilor, compuşi în principal din bucle cu număr cunoscut de paşi şi omogeni din punct de vedere al organizării calculelor.

De aceea considerăm necesară prezentarea aici a modurilor uzuale de repartizare a elementelor unei matrice între procesoarele unei arhitecturi paralele. Se doreşte în general ca datele să fie repartizate procesoarelor în mod *echitabil* şi *fără redundanţe* (aceeaşi dată în memoria mai multor procesoare), astfel încât acestea să ocupe cât mai puţin spaţiu în memoriile locale şi deci să se poată rezolva probleme de dimensiuni cât mai mari. Fie A o matrice cu n linii; fiecare dintre cele p procesoare va trebui deci să primească $m = n/p$ linii; pentru simplitate presupunem că p divide n. Repartizările următoare sunt naturale pe un inel (dar se pot avea în vedere şi alte arhitecturi) şi sunt ilustrate în figura 4.4:

- *bloc linii* în care primele m linii sunt locale procesorului P_0, următoarele m lui P_1, etc. În general, procesorul P_i posedă liniile cu numerele de la im la $(i+1)m-1$ (liniile sunt numerotate începând de la 0).
- *ciclic pe linii* în care procesorului P_i îi aparţin liniile cu numerele $jp+i$, cu $j = 0, 1, \ldots, m-1$; sau, altfel spus, liniile l pentru care $l \bmod p = i$ (o imagine mai plastică: liniile sunt împărţite procesoarelor precum cărţile de joc dintr-un pachet unor jucători aşezaţi în cerc).
- *bloc ciclic pe linii* în care fiecare procesor are câte d linii consecutive; matricea iniţială se împarte în submatrice de câte pd linii, fiecare submatrice repartizându-se bloc linii pe procesoare.

Desigur că cele trei moduri descrise mai sus se pot folosi pe coloane. Particularizările în cazul unui vector sunt şi ele evidente (matricea are o singură coloană).

Pe un tor sau o grilă, aceste moduri se pot generaliza, repartizarea făcându-se simultan pe linii şi pe coloane. Vom discuta pe larg despre acest subiect în capitolul 6.

Pe hipercub, se folosesc uneori repartizări după codul Gray; notăm cu $g(i)$ codul Gray al numărului i. Se trece imediat de la cele trei repartizări pe linii de mai sus (bloc, ciclic şi bloc ciclic), la alte trei, astfel: dacă o linie aparţinea acolo procesorului P_i, acum ea va fi repartizată procesorului $P_{g(i)}$. Un avantaj al acestei tehnici este că, întotdeauna, linii vecine în matrice sunt repartizate aceluiaşi procesor sau unor

procesoare vecine (pe un ciclu hamiltonian); deci, atunci când se va adapta de la inel la hipercub un algoritm în care au loc doar comunicaţii între vecini, pentru a păstra această proprietate trebuie aleasă o repartizare conform codului Gray (aceeaşi, dintre cele trei, ca şi pentru inel).

Indiferent de modul de repartizare, atunci când linia (coloana) i este repartizată în întregime unui singur procesor, atunci acest procesor va fi notat şi cu $P(i)$.

Din nou despre indici globali şi locali. Presupunem, de exemplu, o repartizare oarecare pe linii a unei matrice A; aceasta înseamnă că un procesor deţine un bloc $m \times n$. Într-un program, matricea A ca atare nu este referită printr-o singură variabilă (globală), ci prin p variabile locale fiecărui procesor, purtând acelaşi nume, să zicem B (evident, de obicei se foloseşte chiar numele A). Atunci când vom descrie un algoritm, ne vom referim de regulă la valorile indicilor ca şi cum A ar fi memorată global; deci, a_{ij} este un element pe linia i a matricei A, repartizată procesorului $P(i)$; în variabila locală B a acestui procesor, acest element se numeşte $b_{\ell j}$, unde $\ell \in 0 : m-1$ este indicele local al liniei respective; de pildă, pentru repartizare bloc linii, $\ell = i \bmod m$ (iar $P(i) = \lfloor i/m \rfloor$). Nu vom efectua decât rareori această transformare din indici globali în indici locali, pentru a nu îngreuna citirea algoritmilor. La implementare, însă, ea este obligatorie; altfel, memoria locală nu e utilizată eficient, ci aproximativ ca şi cum întreaga matrice A ar fi memorată de fiecare procesor, dar un procesor nu ar utiliza efectiv decât cele m linii "ale sale".

Planificare şi comunicaţie. Tot în arhitecturile cu memorie distribuită, ce efect are planificarea asupra comunicaţiei ? Dacă datele de ieşire ale unui task T_1 sunt date de intrare pentru un task T_2, spunem că există o operaţie de comunicaţie logică între aceste taskuri; se poate construi un graf orientat al comunicaţiilor logice, în noduri fiind taskurile, iar arcele legând taskuri care comunică. Dacă T_1 este planificat pe procesorul P_i, iar T_2 pe procesorul P_j, atunci, la execuţia algoritmului, va avea loc o comunicaţie între aceste două procesoare; comunicaţia la acest nivel va fi numită fizică.

Are loc, aşadar, o operaţie de scufundare a grafului legăturilor logice dintre taskuri, în graful legăturilor fizice dintre procesoare. O bună planificare trebuie să ţină seama şi de acest aspect. Noţiunile definite în primul capitol capătă acum semnificaţii noi. Dilatarea va reprezenta distanţa maximă între procesoare care comunică; ea este foarte importantă, deoarece multe comunicaţii la distanţe mari vor mări mult timpul de execuţie al algoritmului. Idealul este de a reuşi o planificare în care să se comunice doar între vecini; de multe ori, în concepţia algoritmilor, vom porni de la un astfel de criteriu. Congestia arată numărul legăturilor logice care se suprapun pe o aceeaşi legătură fizică; o congestie mare nu este neapărat nesatisfăcătoare, deoarece comunicaţiile pot avea loc la momente diferite, deci, în realitate, nu vor avea loc conflicte de utilizare a legăturii fizice; astfel, minimizarea congestiei este un obiectiv secundar, în general.

Concluzii. Am enumerat până acum câteva criterii de planificare: planificarea cât mai multor taskuri în paralel, pentru a obţine o încărcare egală, o distribuţie

echitabilă şi regulată a datelor, reducerea distanţelor între procesoarele care comunică. Este practic imposibil ca toate aceste criterii să fie utilizate într-o manieră explicită; adică, să se enumere toate (sau multe) variantele posibile de planificare, să se aprecieze performanţele lor şi apoi să se aleagă varianta optimă. De cele mai multe ori sunt alese doar 2-3 planificări, pe baza unor particularităţi ale problemei (şi a spiritului de observaţie al celui care o rezolvă), dintre care se alege varianta cea mai bună. După dobândirea unei oarecari experienţe, metoda este într-adevăr eficientă; sperăm că lectura acestei lucrări vă va ajuta în dezvoltarea unui bun spirit de observaţie specific.

4.3 Câţiva algoritmi paraleli fundamentali

Vom prezenta în această secţiune câteva exemple de studiu al paralelismului unor algoritmi rezolvând probleme simple, dar care apar foarte des; acestea ilustrează în acelaşi timp limitele ideilor de paralelizare descrise până acum, dar şi procedee intuitive de obţinere a paralelismului. Tratarea va fi aici mai completă decât în capitolele următoare, pentru că suntem la început; în plus, aceşti algoritmi vor fi folosiţi efectiv mai târziu.

4.3.1 Suma a n numere

Prima problemă de care ne ocupăm este cea a calculului sumei a n numere $x_0, x_1, \ldots, x_{n-1}$, elemente ale unui vector x. Ea ilustrează în primul rând importanţa modului în care se formează taskurile.

Notăm $y_i^j = \sum_{l=i}^{j} x_l$. Pentru a putea determina paralelismul intrinsec al algoritmilor, vom presupune taskurile formate dintr-o singură operaţie. Vom utiliza relaţia (de recurenţă):

$$y_i^k = y_i^j + y_{j+1}^k, \quad \text{pentru } i \leq j < k$$

şi cazul ei particular $y_i^{k+1} = y_i^k + x_{k+1}$ (care spune că suma a q numere se poate calcula adunând suma primelor $q-1$ cu cel de-al q-lea număr).

Algoritmul secvenţial. Într-o primă variantă notăm cu T_i, $0 < i \leq n-1$, operaţia de calcul al sumei parţiale $y_0^i = y_0^{i-1} + x_i$; calculul, în ordine crescătoare a indicilor, al sumelor parţiale, este algoritmul secvenţial uzual de calcul al sumei. Se observă însă cu uşurinţă că, în acest fel, calculele se desfăşoară pur secvenţial; într-adevăr, cum $T_{i-1} \prec T_i$, graful de precedenţă este o listă—deci fiecare nivel este format dintr-un singur task. Complexitatea paralelă este de $(n-1)$ operaţii, identică cu cea secvenţială.

Paralelismul maxim. O a doua variantă se poate defini recursiv: se efectuează numărul maxim de adunări care pot decurge în paralel, e.g. se adună $x_i + x_{i+1}$, pentru toţi i pari; se obţin astfel $n/2$ numere, pentru care se repetă algoritmul; la fiecare pas lungimea şirului de numere se înjumătăţeşte, deci complexitatea este $\log n$

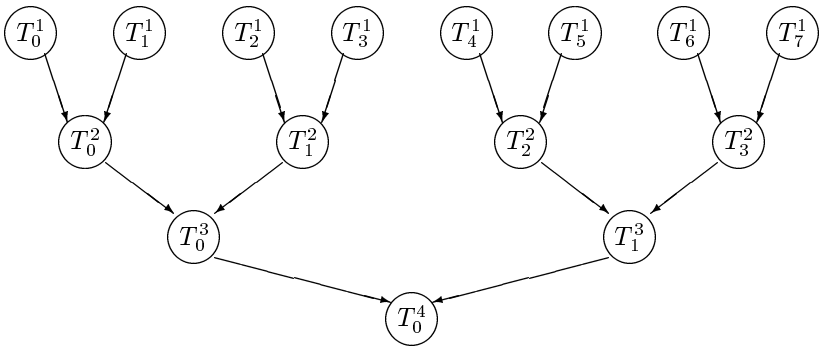


Figura 4.5: Graf de precedenţă pentru calculul sumei a 16 numere.

(presupunem, pentru simplitate, că n este o putere a lui 2). Altfel spus, pentru $k = 1, \ldots, \log n$, se calculează în taskul T_i^k suma parţială

$$y_{i2^k}^{(i+1)2^k-1} = y_{i2^k}^{i2^k+2^{k-1}-1} + y_{i2^k+2^{k-1}}^{(i+1)2^k-1}, \quad \text{pentru } 0 \leq i \leq \frac{n}{2^k} - 1.$$

Relaţiile de precedenţă sunt de tipul $T_{2i}^{k-1} \prec T_i^k$ şi $T_{2i+1}^{k-1} \prec T_i^k$, ceea ce conduce la un graf de precedenţă de forma unui arbore binar (cu frunzele în sus !) echilibrat, deci având o înălţime de $\log n$; un astfel de arbore este prezentat în figura 4.5. Lăţimea acestui graf este $n/2$. Aşadar, putem spune că algoritmul are complexitate $O(\log n)$, folosind $O(n)$ procesoare; de altfel, aceasta este şi complexitatea problemei calculului sumei. Costul algoritmului este $O(n \log n)$, dar efectiv nu se execută decât $n/2 + n/4 + \ldots + 1 = n - 1$ adunări, exact cât şi pentru algoritmul secvenţial precedent. Din punct de vedere teoretic, algoritmul nu este optimal, dar este eficient.

Se observă diferenţa enormă între cele două variante (care, de altfel, sunt extreme), provenind exclusiv din împărţirea în taskuri a operaţiilor, adică din ordonări diferite ale operaţiilor; secvenţial, algoritmii au acelaşi număr de operaţii.

Algoritmi pentru PRAM

Cazul $p = n/2$. Folosind $n/2$ procesoare, planificarea taskurilor din graful de precedenţă este simplă: în etapa k, procesorul P_i execută taskul T_i^k. Pe un EREW PRAM, algoritmul este următorul.

ALGORITM 4.1 (suma numerelor x_i, pe EREW PRAM, $p = n/2$, id $\equiv i$)

1. **pentru** $k = 1 : \log n$
   1. **dacă** $i < n/2^k$ **atunci**
      1. execută T_i^k ($x_{i2^k} \leftarrow x_{i2^k} + x_{i2^k+2^{k-1}}$ sau $x_i \leftarrow x_{2i} + x_{2i+1}$)

a				**b**			
P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
T_0^1	T_1^1	T_2^1	T_3^1	T_0^1	T_2^1	T_4^1	T_6^1
T_4^1	T_5^1	T_6^1	T_7^1	T_1^1	T_3^1	T_5^1	T_7^1
T_0^2	T_1^2	T_2^2	T_3^2	T_0^2	T_1^2	T_2^2	T_3^2
T_0^3	T_1^3			T_0^3	T_1^3		
T_0^4				T_0^4			

Tabelul 4.2: Două planificări, respectând principiul lui Brent, pentru graful de precedenţă din figura 4.5.

În instrucţiunea 1.1.1 sunt date două implementări posibile ale taskului T_i^k, în care elementele iniţiale sunt modificate, astfel încât suma este conţinută, în final, de x_0. Cele două variante diferă prin locaţia în care se depune rezultatul unei sume. În prima, suma se depune în prima dintre cele două variabile sumate; astfel, după prima iteraţie, sumele parţiale sunt depuse în locaţiile pare, după a doua în locaţiile cu număr de ordine divizibil cu 4 etc. În a doua variantă, sumele parţiale se depun în primele locaţii ale vectorului x: prima jumătate, după prima iteraţie, primul sfert, după a doua iteraţie etc. Prima variantă este recomandată, deoarece, într-o iteraţie, fiecare locaţie de memorie este accesată doar de un singur procesor; în a doua, de exemplu, în acelaşi pas, x_1 este citit de P_0 şi apoi scris de P_1. Aşa cum am anunţat în sectiunea 2.4, presupunem că procesoarele se sincronizează după fiecare iteraţie; mai mult, în a doua variantă presupunem că între citirea şi scrierea din acelaşi pas, procesoarele se sincronizează. Accelerarea şi eficienţa au valorile

$$S(n) = \frac{n-1}{\log n}; \qquad \varepsilon(n) = \frac{2(n-1)}{n \log n} \approx \frac{2}{\log n}.$$

Deci, deşi timpul de execuţie al algoritmului e foarte scurt, procesoarele nu sunt folosite eficient; în general, $\varepsilon(n) \ll 1$; aceasta se datorează timpilor morţi ai majorităţii procesoarelor; doar P_0 lucrează permanent, în timp ce jumătate dintre procesoare nu fac decât o singură operaţie.

Cazul $p \ll n$. De obicei, lungimea vectorului x este semnificativ mai mare decât numărul de procesoare, adică $p \ll n$; cum planificăm acum taskurile ? De exemplu, pentru $p = 4$, dacă aplicăm principiul lui Brent pentru graful de precedenţă din figura 4.5 (unde $n = 16$), atunci planificarea arată ca în tabelul 4.2a. Desigur, aceasta nu este unica variantă posibilă; acelaşi timp de execuţie îl are şi planificarea din tabelul 4.2b, care respectă şi ea principiul lui Brent; deşi aceasta din urmă nu pare a avea o calitate deosebită, totuşi e mai naturală: fiecare procesor calculează suma a 4 numere consecutive, după care se foloseşte ideea de planificare din cazul $p = n/2$. Deci, în formă generală, algoritmul este următorul.

ALGORITM 4.2 (suma numerelor x_i, pe EREW PRAM, $p \ll n$, id $\equiv i$)

1. calculează (secvenţial) $z_i \leftarrow \sum_{j=in/p}^{(i+1)n/p-1} x_j$
2. apelează algoritmul 4.1, pentru numerele z_l, $l = 0 : p-1$, cu $p/2$ procesoare

Am presupus că p divide n şi că fiecare procesor calculează suma a n/p numere consecutive (şi deci, o repartizare bloc linii a elementelor din vectorul x, pe procesoare); rezultatul final se va afla în z_0. După calculul secvenţial al sumelor, se obţin p sume parţiale; pentru aplicarea algoritmului 4.1 sunt suficiente $p/2$ procesoare, cele cu adrese $i = 0 : p/2 - 1$. Complexitatea algoritmului 4.2 este $T(n,p) = n/p - 1 + \log p$, iar eficienţa

$$\varepsilon(n,p) = \frac{n-1}{p(n/p - 1 + \log p)} \approx \frac{n}{n + p\log p}$$

este asimptotic egală cu 1 şi, în general, apropiată de 1. În acest caz, putem deci considera algoritmul valoros.

Algoritmi pentru MIMD cu memorie distribuită

Trecem acum la arhitecturile MIMD cu memorie distribuită, încercând să vedem care sunt implicaţiile ideilor algoritmilor de mai sus asupra comunicaţiilor. Presupunem că elementele din x sunt iniţial distribuite egal celor p procesoare. Vom aborda întâi cazul în care $n = p = 2^d$, deci procesorul P_i are în memoria locală un singur element, să zicem x_i (acesta este cel mai mic număr de elemente de care merită să ne ocupăm; dacă rămâneam la o analogie perfectă cu cazul memorie comună, am fi ales $n = 2p$; cazul $n = p$ e însă cel mai defavorabil, dar care implică toate procesoarele). Păstrăm planificarea în care P_i execută taskul T_i^k; dacă încercăm să utilizăm una dintre cele două forme ale instrucţiunii 1.1.1 din algoritmul 4.1, vom constata că P_i nu are în memoria proprie nici unul dintre elementele de sumat (prima formă e totuşi adaptabilă, vezi problema 4.3.5). Vom încerca o a treia formă, în care P_i să calculeze de fiecare dată suma dintre elementul pe care îl posedă şi un altul (pentru simplificarea scrierii am inversat şi ordinea de ciclare):

1. **pentru** $k = \log n - 1 : -1 : 0$
   1. **dacă** $i < n/2^{d-k}$ **atunci** $x_i \leftarrow x_i + x_{i+2^k}$

În descriere recursivă, această formă seamănă cu cea din algoritmul 4.1, doar că se formează altfel perechile de elemente a căror sumă se calculează; se fac întâi sumele $x_0 + x_{n/2}$, $x_1 + x_{n/2+1}$, etc.; se obţin $n/2$ numere, pentru care se repetă procedeul. Atunci se observă imediat că procesorul P_i efectuează suma dintre numărul său (în care se acumulează o sumă parţială) şi numărul (suma parţială) primit de la procesorul P_{i+2^k}; condiţia din instrucţiunea 1.1 spune că bitul k al numărului binar i este 0, adică P_i şi P_{i+2^k} sunt vecini dintr-un hipercub. În scrierea algoritmului în stil SPMD, vom ţine seama de faptul că elementul x_i ocupă o singură locaţie în memoria locală a procesorului P_i, pe care o vom numi generic x (folosim aşadar indici locali, în timp ce mai sus erau folosiţi indici globali). Deci, algoritmul are forma:

ALGORITM 4.3 (suma a n numere, pe hipercub, $p = n$; numărul din memoria locală este x; suma se calculează pe loc în x şi se va afla în memoria procesorului P_0)

1. **pentru** $k = d-1 : -1 : 0$
   1. **dacă** id $< 2^k$ **atunci**
      1. **recv**(z, k), $x \leftarrow x + z$
   2. **altfel dacă** id $< 2^{k+1}$ **atunci** **send**(x, k)

Se observă că după iteraţia k rămân active doar procesoarele dintr-un hipercub de dimensiune k, cel notat $\mathcal{H}_d^{d-k}(0)$; fiecare astfel de procesor P_i calculează în această etapă suma elementelor iniţial locale procesoarelor din hipercubul de dimensiune $d-k$ notat $\bar{\mathcal{H}}_d^k(i)$. În final, suma este calculată de P_0.

Timpul de execuţie al algoritmului 4.3 este

$$T(p) = (\log p)\alpha + (\log p)(\sigma + \beta),$$

primul termen fiind pentru adunări, celălalt pentru comunicaţii. Desigur că eficienţa algoritmului este modestă.

Dacă $n \gg p$, atunci fiecare procesor are în memoria locală n/p elemente, le calculează suma, după care se utilizează algoritmul 4.3. Timpul total este acum

$$T(n,p) = (n/p - 1)\alpha + (\log p)(\alpha + \sigma + \beta),$$

iar eficienţa $\varepsilon(n,p) \approx \frac{n\alpha}{n\alpha + p(\log p)(\alpha+\sigma+\beta)}$, asimptotic egală cu 1.

Generalizare: colectare cu sumare. Să încercăm să facem acum legătura cu un algoritm de comunicaţie. Cu puţină atenţie, se vede că, strict din punct de vedere al ordinii şi dimensiunilor pe care se comunică, algoritmul 4.3 este inversul algoritmului sugerat în figura 3.23, de distribuţie după un arbore de acoperire binomial, la fiecare pas după o direcţie. Deci, seamănă cu un algoritm de colectare.

Într-adevăr, o primă idee naivă de a calcula suma pe o arhitectură cu memorie distribuită ar putea fi simpla colectare a elementelor x_i într-un singur procesor, urmată de calculul secvenţial al sumei, în $p-1$ operaţii. Aceasta este probabil o variantă mai puţin eficientă; ea poate fi luată în considerare dacă există o funcţie hardware de colectare, mult mai rapidă decât una realizată prin program. Dar algoritmul 4.3 îmbină ideea colectării cu cea a calculului sumelor parţiale pe parcurs, astfel încât se transmite un singur element la un moment dat; în acest fel se reduce şi numărul de operaţii aritmetice, şi timpul de comunicaţie. Ideea se poate aplica pentru orice topologie şi algoritm de colectare. Acesta este motivul pentru care algoritmii de sumare pe arhitecturi cu memorie distribuită sunt denumiţi, de multe ori, algoritmi de *colectare cu sumare*.

O ultimă remarcă: toţi algoritmii din această secţiune sunt valabili nu numai pentru sumă, ci şi pentru orice altă operaţie, cu singura condiţie ca aceasta să fie asociativă şi, doar pentru algoritmul 4.3, comutativă (vezi problema 4.3.5 pentru a elimina această condiţie). Calculul $x_0 \bullet x_1 \bullet \ldots \bullet x_{n-1}$, unde $\bullet$ este o operaţie asociativă mai este numit şi *reducere*.

4.3.2 Suma globală

Problema sumei globale e o mică variaţie pe tema sumei, pe arhitecturi cu memorie distribuită. Fiind date n numere x_0, x_1, ... x_{n-1}, elemente ale unui vector x, să se calculeze suma lor astfel încât *toate* procesoarele să o cunoască (să o posede în memoria locală). Desigur, calculul sumei ca mai sus, urmat de o difuzare efectuată de P_0 ar rezolva problema. Să căutăm însă o metodă mai rapidă. Ne vom mărgini discuţia la hipercub. Presupunem întâi $p = n$, deci procesorul P_i deţine în memoria locală numărul x_i.

Dacă algoritmul de calcul al sumei implica o comunicaţie identică celei de la colectare, formularea problemei sumei globale ar sugera să ne orientăm spre difuzarea generală. Să mai privim o dată figura 3.21, în care este prezentat un algoritm de difuzare generală, la fiecare pas pe o dimensiune. Modificăm algoritmul de difuzare generală astfel: după ce două procesoare vecine într-o anume dimensiune îşi transmit valorile lor locale, ele efectuează şi suma acestor valori, pe loc. Aşadar, numerele asociate nodurilor în figura 3.21 reprezintă indicii iniţiali şi finali ai sumelor parţiale deţinute de nodurile respective. Pentru exemplificare, să ne referim la procesorul P_4; în prima etapă, schimbă elementul său (x_4) cu cel al lui P_5, deci pe dimensiunea 0, şi calculează suma lor y_4^5; apoi, schimbă această sumă cu cea a lui P_6, pe dimensiunea 1, şi calculează noua sumă, adică y_4^7; în fine, schimbă cu P_0, pe dimensiunea 2, şi adună, obţinând suma totală y_0^7. Dacă $p < n$ generalizarea este imediată iar algoritmul este următorul.

ALGORITM 4.4 (suma globală a n numere, pe hipercub)

1. $x \leftarrow$ suma elementelor locale din vector
2. **pentru** $k = 0 : d - 1$
   1. **în paralel** **recv**(z, k), **send**(x, k)
   2. $x \leftarrow x + z$

Operaţiile efectuate sunt aceleaşi pentru toate procesoarele. Timpul său de execuţie este identic cu cel pentru sumă, ceea ce justifică efortul de elaborare a algoritmului de sumă globală. Este interesant că am obţinut un timp bun, chiar dacă unele operaţii sunt dublate; de exemplu, în primul pas, P_4 şi P_5 calculează amândouă y_4^5; mai mult, în ultimul pas, toate procesoarele efectuează aceeaşi adunare. Avantajul este că prin aceste operaţii redundante se elimină necesitatea comunicaţiei (a difuzării finale a sumei); în plus, ele sunt făcute de procesoare care altfel ar fi stat; timpii morţi din algoritmul 4.3 sunt eliminaţi, fără ca P_0, singurul care lucra permanent acolo, să aibă acum ceva în plus de făcut.

Calculul mai multor sume globale. Dacă trebuie calculate mai multe sume, aceasta se poate face comunicându-se în paralel. După calculul sumelor locale (dacă $n > p$), pentru fiecare sumă globală se începe comunicaţia pe altă dimensiune; de exemplu, pentru prima pe dimensiunea 0, pentru a doua pe dimensiunea 1, etc.; la pasul al doilea, pentru suma j se comunică pe dimensiunea $(j + 1) \bmod d$; ş.a.m.d., la pasul k, cu $0 \leq k \leq d - 1$, pentru suma j se comunică pe dimensiunea $(j +$

k) mod d, comunicaţia desfăşurându-se în paralel, dacă sunt cel mult d sume globale de calculat. Deci, pentru d sume, timpul de comunicaţie este acelaşi ca pentru una singură; desigur, timpii necesari adunărilor se cumulează. Sursa de inspiraţie a acestei idei este algoritmul rotativ de difuzare generală !

4.3.3 Suma prefixelor

Aceasta este o generalizare a problemei sumei, cerându-se acum calculul tuturor sumelor parţiale $s_i \equiv y_0^i = \sum_{l=0}^{i} x_l$, cu $i = 0, \ldots, n-1$. Numele de suma prefixelor (prefix sum) provine din faptul că aceste sume parţiale sunt formate întotdeauna cu primele elemente din vectorul x. Secvenţial, calculul este banal, recurenţa $s_{i+1} = s_i + x_{i+1}$ implicând $n-1$ operaţii; însă, după cum am văzut la problema sumei, acest mod de calcul este absolut secvenţial.

Algoritmi pentru PRAM

Sumare în cascadă. Să încercăm întâi generalizarea algoritmului 4.1; acolo, după prima iteraţie, $n/2$ procesoare calculau suma a câte două elemente consecutive din x; după a doua, $n/4$ procesoare calculau suma a câte patru elemente; şi tot aşa, din ce în ce mai puţine procesoare calculând suma tot mai multor elemente. Apare imediat ideea de a încerca să lăsăm *toate* procesoarele să lucreze, după acelaşi principiu: la pasul k, fiecare să calculeze suma a 2^k elemente consecutive. Pentru aceasta, trebuie modificată puţin prima formă a instrucţiunii 1.1.1 din algoritmul 4.1; se observă că se efectua suma între elemente aflate la distanţă 2^{k-1} în vectorul x; vom schimba doar locul în care se va depune rezultatul, anume în locaţia elementului cu indice mai mare. În plus, vom face aceasta pentru toate elementele x_i, de fiecare element ocupându-se un procesor; mai trebuie să avem grijă ca sumele să cuprindă doar elemente valide ale vectorului x, adică să nu efectuăm nici o operaţie dacă poziţia $i - 2^{k-1}$ este în afara vectorului (mai mică decât 0). Mersul calculelor este exemplificat în figura 4.6, fiecare procesor efectuând o sumă parţială, şi se desfăşoară conform algoritmului următor, numit şi de sumare în cascadă, în care sumele sunt calculate pe loc în elementele x_i.

ALGORITM 4.5 *(sumare în cascadă)* (suma prefixelor, pe CREW PRAM, $p = n$, id $\equiv i$)

1. **pentru** $k = 0 : \log n - 1$
    1. **dacă** $i \geq 2^k$ **atunci**
        1. $x_i \leftarrow x_i + x_{i-2^k}$

În etapa k, locaţia i e citită de două procesoare, P_i şi P_{i+2^k}; pentru a adapta algoritmul pe EREW PRAM, citirile ar trebui să aibă loc în două etape (folosind n locaţii de memorie în plus, pentru copii ale sumelor parţiale). De asemenea, chiar şi pe CREW PRAM, funcţionarea corectă a instrucţiunii 1.1.1 presupune că procesoarele se sincronizează între citiri şi scrieri. Complexitatea paralelă a algoritmului este $\log n$, dar costul său de $n \log n$ îl face neoptimal (dar eficient, totuşi).

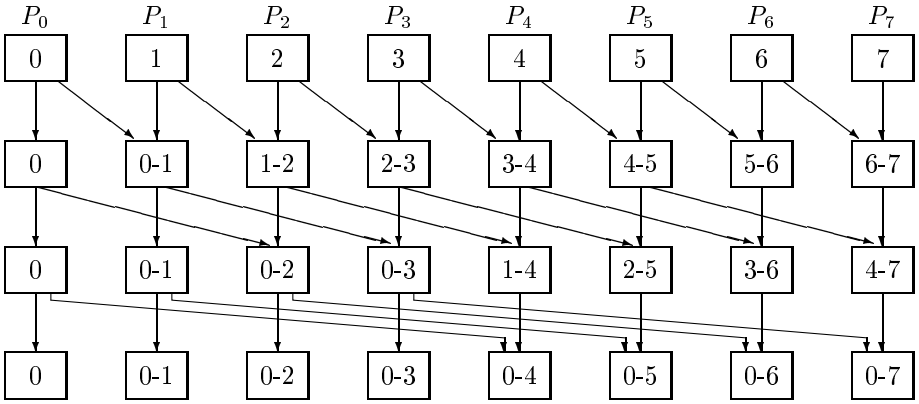


Figura 4.6: Calculul sumei prefixelor conform algoritmului 4.5 (sumare în cascadă); fiecare dreptunghi corespunde unei locaţii de memorie, conţinutul reprezentând indicii limită ai sumelor parţiale (y_j^l este figurat ca *j-l*); etapele de calcul au loc de sus în jos; săgeţile indică dependenţele de date (se poate privi acest desen şi ca o reprezentare a grafului de precedenţă).

În cazul $n \gg p$, aplicarea principiului lui Brent algoritmului 4.5 nu duce la un rezultat prea fericit; costul algoritmului nu se modifică, deci timpul este $(n \log n)/p$. Ori, ne aşteptăm să obţinem ceva de genul n/p, pentru a avea eficienţă aproape de 1. Pentru ameliorarea eficienţei, putem proceda astfel: fiecare procesor P_i calculează suma elementelor "sale", cu indici de la in/p la $(i+1)n/p-1$, într-o variabilă z_i; pentru aceste p variabile se aplică algoritmul 4.5; apoi, fiecare procesor calculează secvenţial cele n/p sume ale sale, adunând pe rând elementele sale la suma $z_{i-1} = s_{in/p-1}$ (suma tuturor elementelor cu indice mai mic decât cel al elementelor sale).

ALGORITM 4.6 (suma prefixelor, pe CREW PRAM, $p \ll n$, id $\equiv i$)

1. $z_i \leftarrow \sum_{j=in/p}^{(i+1)n/p-1} x_j$ {calculul sumelor locale}
2. calculează suma prefixelor cu algoritmul 4.5, pentru elementele z_i
3. **dacă** $i \neq 0$ **atunci** $x_{in/p} \leftarrow x_{in/p} + z_{i-1}$
4. **pentru** $j = in/p + 1 : (i+1)n/p - 1$
   1. $x_j \leftarrow x_j + x_{j-1}$

Numărul de adunări este de n/p în instrucţiunea 1, $\log p$ pentru suma prefixelor, şi n/p pentru instrucţiunea 4; deci, în total, sunt $2n/p + \log p$ adunări. Iată deci un prim caz în care eficienţa asimptotică nu mai este 1, ci 1/2; aceasta se datorează efectuării de două ori a adunărilor cu x_i, prima dată doar pentru a putea reduce dimensiunea problemei de la n la p; este un procedeu destul de des întâlnit, acela de a face calcule suplimentare pentru ajungerea la o situaţie cu bun paralelism. În orice caz, rezultatul este mai bun decât cel obţinut cu principiul lui Brent.

Algoritmul Ladner-Fischer. Vom prezenta acum un alt algoritm pentru care dependenţele de date sunt mai simple; algoritmul se va implementa lesne pe un EREW PRAM. Ideea este de natură divide et impera; se reduce dimensiunea problemei sumei prefixelor la jumătate, se apelează recursiv algoritmul, apoi, din cele $n/2$ sume astfel calculate se deduc cele n sume cerute iniţial. O descriere generală are forma următoare:

ALGORITM 4.7 *(Ladner-Fischer)* (suma prefixelor în paralel, n putere a lui 2)

1. **pentru** $i = 0 : n/2 - 1$, **în paralel** $\xi_i \leftarrow x_{2i} + x_{2i+1}$
2. apelează recursiv acest algoritm, pt. suma prefixelor valorilor ξ_i ($z_i \leftarrow \sum_{\ell=0}^{i} \xi_\ell$)
3. **pentru** $i = 0 : n - 1$, **în paralel**
   1. **dacă** i e impar **atunci** $s_i \leftarrow z_{(i-1)/2}$
   2. **dacă** i e par **atunci** $s_i \leftarrow z_{i/2-1} + x_i$

În instrucţiunea 1 se efectuează sumele a $n/2$ perechi de valori; după calculul recursiv al sumei prefixelor pentru aceste sume ξ_i, jumătate din sumele dorite sunt deja calculate (cele pentru i impar); pentru celelalte sume trebuie efectuată încă o adunare, cea din recurenţa $s_k = s_{k-1} + x_k$ (cu k par); convenim că $s_{-1} = z_{-1} = 0$. Se poate remarca o oarecare asemănare cu algoritmul 4.6, diferenţa esenţială fiind că acela nu era recursiv. Complexitatea paralelă a algoritmului 4.7 este de $T(n) = 2 \log n$, câte $\log n$ înaintea, respectiv după apelurile recursive (se respectă relaţia $T(n) = T(n/2) + 2$). Pentru a prezenta o planificare a operaţiilor, vom trece la o formă iterativă a algoritmului. Vom nota cu w_i^k variabila ce conţine suma parţială i, la nivelul de recursie k ($w_i^0 \equiv x_i$).

ALGORITM 4.8 (suma prefixelor în paralel, pe EREW PRAM, n putere a lui 2, $p = n/2$, id $\equiv i$)

1. **pentru** $k = 1 : \log n$
   1. **dacă** $i < n/2^k$ **atunci** $w_i^k \leftarrow w_{2i}^{k-1} + w_{2i+1}^{k-1}$
2. **pentru** $k = \log n : -1 : 1$
   1. **dacă** $i < n/2^k$ **atunci**
      1. $w_{2i+1}^{k-1} \leftarrow w_i^k, \quad w_{2i}^{k-1} \leftarrow w_{i-1}^k + w_{2i}^{k-1}$

După prima buclă **pentru**, $w_0^{\log n}$ conţine suma totală; în a doua, în iteraţia k, procesorul P_i execută echivalentul instrucţiunilor 3.1 şi 3.2 din algoritmul 4.7; deoarece acum privim din punctul de vedere al adresei procesorului, iar acesta se ocupă de elementele cu indicii $2i$ şi $2i + 1$, se înlocuieşte în 3.1 i cu $2i + 1$, şi în 3.2 i cu $2i$. În plus, convenim că $w_{-1}^k = 0$. Într-o iteraţie, o locaţie de memorie nu este citită decât de un singur procesor; sunt necesare $n/2 + n/4 + \ldots + 1 = n - 1$ locaţii suplimentare de memorie. Costul algoritmului este $O(n \log n)$.

În cazul $n \gg p$, se aplică aceeaşi idee ca la algoritmul 4.6, după sumarea secvenţială apelându-se însă algoritmul 4.8; complexitatea paralelă este $2n/p + 2 \log p$. Eficienţa asimptotică este deci 1/2, pentru orice $p \leq O(n/\log n)$ (vezi problema 4.3.2, demonstraţia fiind asemănătoare celei a algoritmului 4.2), iar costul algoritmului $O(n)$.

Algoritmi pentru MIMD cu memorie distribuită

Ne ocupăm doar de cazul $n = p$, pe hipercub; dacă $n \gg p$, se aplică ideea din algoritmul 4.6.

Sumare în cascadă. O primă încercare este de a vedea ce comunicaţii implică algoritmul 4.5; dacă elementul x_i este repartizat procesorului P_i, atunci, la pasul k, procesorul i comunică cu $i - 2^k$; din păcate, aceste două procesoare sunt vecine doar atunci când bitul k din i este 1; dacă însă $i_k = 0$, atunci toţi biţii din poziţiile mai mari decât k (inclusiv) sunt diferiţi, deci distanţa este $d - k$; concluzia este că algoritmul e inaplicabil în această formă.

Dacă se utilizează o repartizare după codul Gray, deci dacă x_i este în memoria locală a procesorului $g(i)$, atunci lucrurile se schimbă. Se poate demonstra (vezi problema 4.3.10) că distanţa pe hipercub între procesoarele $g(i)$ şi $g(i - 2^k)$ este cel mult 2; aceasta conduce la comunicaţii destul de performante. Nu detaliem algoritmul; se pot alege două implementări. Una este cea obişnuită, în care elementele rămân pe locurile lor, şi se utilizează comunicaţia (cu eventualele funcţii de dirijare oferite de sistemul de operare). A doua se bazează pe interschimbări ale elementelor, care se fac astfel încât să se comunice doar între vecini; un exemplu este prezentat în figura 4.7; în hipercubul $\mathcal{H}_3$, au loc trei etape de comunicaţie, fiecare în doi paşi (mai puţin ultima, după care nu mai e nevoie de interschimbări); fiecare desen reprezintă un pas de comunicaţie; limitele sumelor parţiale rezultate sunt cele din desenul următor; în primul pas se comunică la distanţă 1, pe ciclul hamiltonian; urmează două interschimbări, reprezentând al doilea pas, care fac ca la al treilea pas să se comunice tot între vecini, etc. Nu este întâmplător faptul că, în final, repartizarea este cea în ordinea uzuală (şi nu după codul Gray).

În ambele implementări, volumul comunicaţiei poate fi apreciat la $(2 \log p)(\sigma + \beta)$, ceea ce este destul de convenabil; pentru adunări se consumă timpul $(\log p)\alpha$. Prima implementare este comodă, dacă există funcţii accesibile de dirijare; a doua implică descoperirea şi detalierea mecanismului general de interschimbare (la care vă urăm succes !).

Suma prefixelor şi suma globală. Întrucât varianta pe hipercub a algoritmului 4.7 (Ladner-Fischer) nu este foarte simplă, vom prezenta o altă posibilitate, aceea de a adapta algoritmul de sumă globală astfel încât să fie calculate şi sumele parţiale de care avem nevoie. Împărţim hipercubul în două jumătăţi: inferioară (adresele mai mici decât $p/2$) şi superioară; calculăm *atât* suma globală z, *cât şi* suma prefixelor s, în fiecare dintre aceste hipercuburi, printr-un apel recursiv (z şi s sunt variabile locale); apoi fiecare nod din hipercubul inferior schimbă suma sa globală cu cea a nodului echivalent din hipercubul superior; fiecare nod actualizează suma sa globală ca în algoritmul 4.4, adică adună suma z locală cu cea primită; pentru nodurile din hipercubul inferior, suma prefixelor este deja corectă; cele din hipercubul superior adună la suma prefixelor s locală, suma globală z primită. Justificarea este simplă: pentru $i \geq p/2$, avem $y_0^i = y_0^{p/2-1} + y_{p/2}^i$; dar $y_0^{p/2-1}$ este suma globală în hipercubul inferior, iar $y_{p/2}^i$ este suma prefixelor în hipercubul superior. În algoritmul următor, forma

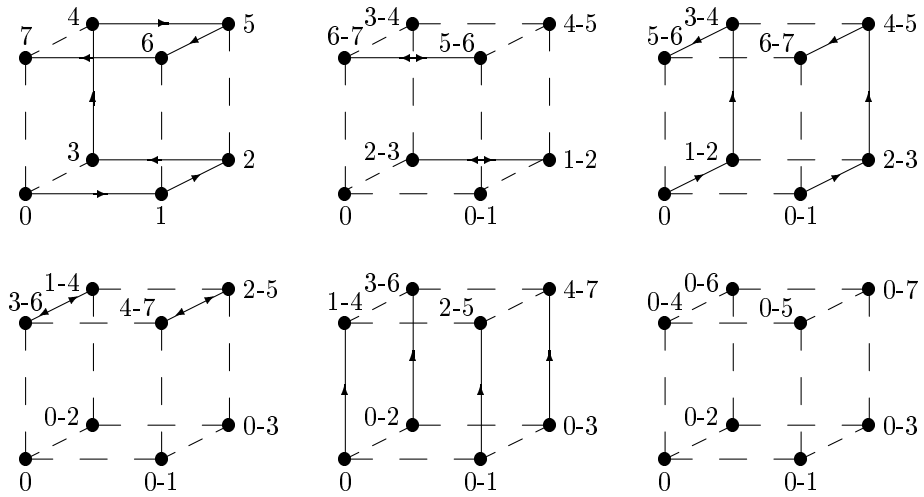


Figura 4.7: O implementare a algoritmului 4.5, pe hipercub, cu repartizare iniţială după codul Gray şi interschimbări la fiecare etapă.

recursivă capătă aspect iterativ datorită proprietăţilor topologice ale hipercubului.

ALGORITM 4.9 (suma prefixelor pe hipercub, adaptare a sumei globale, $p = n$, numărul din memoria locală este x)

1. $z \leftarrow x$, $s \leftarrow x$ $\quad \{z -$ suma globala, $s -$ suma prefixelor$\}$
2. **pentru** $k = 0 : d - 1$
   1. **în paralel** **recv**(v, k), **send**(z, k)
   2. **dacă** $\mathrm{id}_k = 1$ **atunci** $s \leftarrow s + v$
   3. $z \leftarrow z + v$

În 2.2 se actualizează suma prefixelor, în hipercubul superior, iar în 2.3, suma globală, întotdeauna. Timpul necesar calculelor este $(2 \log p)\alpha$ (P_{p-1} face la fiecare iteraţie câte două adunări), deci de două ori mai mare decât pentru adaptarea algoritmului 4.5 prezentată puţin mai sus. În schimb, timpul de comunicaţie este de două ori mai mic, adică $(\log p)(\sigma + \beta)$. Cum, în general, calculele sunt mai rapide decât comunicaţia, algoritmul 4.9 este de preferat. Totuşi, dacă în locul adunării este o altă operaţie, trebuie avut în vedere că algoritmul 4.9 ca atare necesită şi comutativitatea operaţiei, spre deosebire de ceilalţi algoritmi, pentru care asociativitatea este suficientă; o mică modificare face ca şi acest algoritm să nu depindă de comutativitate (vezi problema 4.3.11).

4.3.4 Recurenţe de ordinul I

Un şir $\{x_i\}$, recurent (liniar) de ordinul I, se defineşte prin relaţia $x_{i+1} = a_i x_i + b_i$, cu $0 \leq i \leq n-1$, unde a_i şi b_i sunt coeficienţi reali. Fiind daţi aceşti coeficienţi şi valoarea x_0, se pune problema determinării valorii termenului x_n. Calculul sumei a n numere este un caz particular al acestei probleme; dacă $s_i = \sum_{l=0}^{i} x_i$, atunci în recurenţa $s_{i+1} = a_i s_i + b_i$, coeficienţii sunt $a_i = 1$ şi $b_i = x_i$. Aplicarea formulei de recurenţă ca atare conduce la un algoritm perfect secvenţial, cu complexitate de $2n$ operaţii.

Reducere ciclică. Ideea de paralelizare este clasică, şi va mai fi întâlnită în această lucrare. În relaţia de calcul al termenului x_{i+2}, se înlocuieşte x_{i+1} prin relaţia ce-l caracterizează, i.e.

$$x_{i+2} = a_{i+1}x_{i+1} + b_{i+1} = a_{i+1}a_i x_i + a_{i+1}b_i + b_{i+1} \equiv a'_i x_i + b'_i,$$

unde

$$a'_i = a_{i+1}a_i; \qquad b'_i = a_{i+1}b_i + b_{i+1}. \tag{4.4}$$

Se obţine astfel o nouă relaţie de recurenţă, de data aceasta între x_{i+2} şi x_i, tot liniară şi tot de ordinul I. Dacă se calculează noii coeficienţi ai relaţiei, ca în (4.4), pentru toţi i pari, cu $0 \leq i < n$, se obţine un nou şir recurent, conţinând doar $n/2$ termeni. Se procedează analog în continuare, în fiecare etapă înjumătăţindu-se lungimea şirului; în etapa k, lungimea şirului este $n/2^k$; dacă nu renumerotăm elementele şirului, după etapa k rămân în cursă elementele din poziţiile de forma $i = j2^k$, cu $0 \leq j < n/2^k$, ale şirului iniţial; relaţia de recurenţă este acum de forma $x_{i+2^{k-1}} = a_i^k x_i + b_i^k$, iar noii coeficienţi sunt

$$a_i^k = a_{i+2^{k-1}}^{k-1} a_i^{k-1}; \qquad b_i^k = a_{i+2^{k-1}}^{k-1} b_i^{k-1} + b_{i+2^{k-1}}^{k-1}, \tag{4.5}$$

unde am notat în mod natural $a_i^0 = a_i$, $b_i^0 = b_i$. Presupunând că $n = 2^r$, după r etape se obţine o unică relaţie de recurenţă: $x_n = a_0^r x_0 + b_0^r$, din care se calculează x_n.

Acest algoritm poartă numele de *reducere ciclică*, datorită reducerii permanente a numărului de elemente din şir, şi, ca mecanism de desfăşurare, este asemănător cu 4.1, cu modificarea din problema 4.3.5; totuşi, trebuie făcută o observaţie importantă; deşi graful de precedenţă a taskurilor este, ca şi pentru sumă, un arbore binar echilibrat cu sensurile tuturor arcelor schimbate, semnificaţia este cu totul alta; la sumă, o operaţie consta în efectuarea unei sume parţiale; la reducerea ciclică, însă, nu se calculează decât coeficienţii unei noi relaţii, şi nu ceva în legătură cu termenii x_i. Dacă sunt $n/2$ procesoare, calculele de tipul (4.5) se pot efectua în paralel, în fiecare etapă k; pentru a planifica operaţiile, ţinem seama de faptul că un procesor P_j se ocupă de două perechi de coeficienţi; deci, în etapa k, P_j va calcula a_i^k, cu $i = j2^k$ $(k \geq 1)$. Forma algoritmului este următoarea:

ALGORITM 4.10 (reducere ciclică, pe EREW PRAM, $p = n/2$, $n = 2^r$)

1. **pentru** $k = 1 : \log n$
   1. **dacă** $\mathrm{id} < n/2^k$ **atunci**
      1. $i \leftarrow \mathrm{id} \cdot 2^k$
      2. $a_i^k \leftarrow a_{i+2^{k-1}}^{k-1} a_i^{k-1}$, $b_i^k \leftarrow a_{i+2^{k-1}}^{k-1} b_i^{k-1} + b_{i+2^{k-1}}^{k-1}$
2. **dacă** $\mathrm{id} = 0$ **atunci**
   1. $x_n \leftarrow a_0^r x_0 + b_0^r$

Se observă că indicele superior al variabilelor a_i^k şi b_i^k poate fi omis, calculele efectuându-se pe loc. Deoarece formulele (4.5) implică 3 operaţii, înseamnă că numărul total de operaţii va fi de $2 + 3\log n$. Costul algoritmului este de $O(n \log n)$. Dar, ca şi la calculul sumei, majoritatea procesoarelor nu lucrează decât o mică parte din timp; varianta secvenţială a algoritmului 4.10 are numai $3(n/2 + n/4 + \ldots + 1) + 2 = 3n - 1$ operaţii.

Cazul $n \gg p$**.** Aplicându-se principiul lui Brent, algoritmul obţinut va avea o destul de bună eficienţă. Dacă procesorul P_i se ocupă de calculul coeficienţilor a_l şi b_l, cu $in/p \le l \le (i+1)n/p - 1$, atunci, prin $3n/p - 3$ operaţii, el reduce numărul de elemente din acest subşir la unul singur (aceasta se poate face şi în altă ordine decât în algoritmul 4.10; de exemplu, P_0 poate calcula astfel: $a_0 \leftarrow a_1 a_0$, $b_0 \leftarrow a_1 b_0 + b_1$, apoi $a_0 \leftarrow a_2 a_0$, $b_0 \leftarrow a_2 b_0 + b_2$, ş.a.m.d., deci se elimină elementele din şir în ordine crescătoare); în acest fel, rămân în total p elemente din şir, pentru reducerea cărora se aplică algoritmul 4.10 cu $p/2$ procesoare, ceea ce va consuma $2 + 3\log p$ operaţii; pentru o descriere mai riguroasă a algoritmului, vezi problema 4.3.13. Deci, în total, aproximativ $3(n/p + \log p)$ operaţii, ceea ce conduce la o eficienţă

$$\varepsilon(n,p) \approx \frac{2n}{3(n + p\log p)} \overset{n\to\infty}{\longrightarrow} \frac{2}{3}.$$

Prefixele unei recurenţe. O altă problemă legată de şirurile recurente de ordinul I este calculul *tuturor* valorilor x_i, cu $0 \le i \le n$. Dacă analogul calculului elementului final x_n era calculul sumei, pentru această nouă problemă ne gândim, în mod natural, la o analogie cu suma prefixelor. Într-adevăr, toţi algoritmii prezentaţi acolo pot fi ajustaţi cu uşurinţă; ne vom ocupa aici numai de adaptarea algoritmului 4.5, pentru ceilalţi propunându-vă câteva probleme.

Ideea de bază este de a aplica algoritmul 4.10 din punctul de vedere al fiecărui procesor, după o schemă de tipul celei din figura 4.6. Se folosesc tot formulele (4.5), dar memorând acum noile valori ale coeficienţilor a şi b în poziţiile cu indice superior şi efectuând calculele pentru toţi indicii i pentru care au sens; cu alte cuvinte, se folosesc formulele

$$a_i^k = a_i^{k-1} a_{i-2^{k-1}}^{k-1}; \qquad b_i^k = a_i^{k-1} b_{i-2^{k-1}}^{k-1} + b_i^{k-1}, \tag{4.6}$$

pentru toţi $i > 2^{k-1}$. Se obţine aşadar algoritmul:

ALGORITM 4.11 (reducere ciclică complet paralelă, pe CREW PRAM, $p = n$, $\mathrm{id} \equiv i$)

1. **pentru** $k = 0 : \log n - 1$
   1. **dacă** $i \geq 2^k$ **atunci**
      1. $b_i \leftarrow a_i b_{i-2^k} + b_i,\ \ a_i \leftarrow a_i a_{i-2^k}$
2. $x_{i+1} \leftarrow a_i x_0 + b_i$

După cum este normal, procesorul P_i calculează valoarea x_{i+1} (x_0 fiind cunoscut). Atribuirile din instrucţiunea 1.1.1 sunt scrise în această ordine pentru a se putea respecta relaţiile (4.6). Trecerea la un algoritm EREW PRAM se face aşa cum am discutat la suma prefixelor; totuşi, aici apare o problemă în plus: accesul simultan al mai multor procesoare la x_0; chestiunea poate fi rezolvată foarte simplu, printr-un artificiu de notaţie; se renumerotează x_i prin x_{i+1}, şi se alege $x_0 = 0$; astfel, dificultatea dispare pur şi simplu.

4.3.5 Produsul matrice-vector

Deşi locul problemei calculului vectorului $z = Ax + y$, unde A este o matrice $n \times n$, y şi z vectori de dimensiune n, ar fi mai degrabă în capitolul de algoritmi numerici, o vom prezenta totuşi aici, în chip de avanpremieră; vom avea astfel o imagine a caracteristicilor algoritmilor implicând matrice şi vectori. Operaţia $z = Ax + y$ este cunoscută şi sub numele de gaxpy (General Ax plus y, matricea A fiind generală, i.e. neavând o structură specială); desigur, problema mai dificilă este de a calcula produsul matrice-vector, suma a doi vectori fiind o operaţie perfect paralelă.

Algoritmi pentru PRAM

Complexitatea paralelă. Notăm prin a_{ij} elementul din linia i, coloana j, al matricei A; desigur, x_i este elementul din poziţia (linia) i a vectorului x; vom numerota liniile şi coloanele începând de la zero. Calculul vectorului z se face conform relaţiei

$$z_i = y_i + \sum_{j=0}^{n-1} a_{ij} x_j, \text{ cu } 0 \leq i \leq n-1, \tag{4.7}$$

care implică $2n^2$ operaţii; pentru un i fixat, această relaţie este un produs scalar între linia i din A şi vectorul x, ceea ce este sugerat în figura 4.8, în care termenul y_i nu a mai fost reprezentat. Se observă că, pe un CREW PRAM, fiecare z_i poate fi calculat independent; dacă alocăm câte $O(n)$ procesoare pentru calculul fiecărui z_i, acesta poate fi efectuat în $O(\log n)$ operaţii, fiind vorba despre o sumă; deci, cu $p = O(n^2)$ procesoare, complexitatea paralelă este $O(\log n)$. Desigur, deoarece costul este $O(n^2 \log n)$ şi numărul de procesoare foarte mare, această abordare este utilă doar pentru a stabili care este complexitatea problemei.

Cazul $p = n$. Până la a ajunge la cazul practic $p \ll n$, să ne oprim la situaţia $p = n$, care permite o bună analiză a dificultăţilor ce apar. O primă ideea e de a utiliza planificarea în care procesorul P_i calculează z_i; din (4.7), se observă că P_i

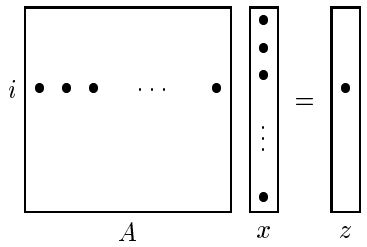


Figura 4.8: Produsul matrice-vector calculat prin produse scalare.

foloseşte linia i din A şi vectorii x şi y, în întregime. Pe un CREW PRAM, algoritmul decurge în felul următor.

ALGORITM 4.12 (gaxpy pe CREW PRAM, $p = n$, id $\equiv i$)

1. $z_i \leftarrow y_i$
2. **pentru** $j = 0 : n-1$
    1. $z_i \leftarrow z_i + a_{ij}x_j$

Pentru a obţine o versiune EREW PRAM, singurul impediment este accesul procesoarelor la vectorul x; în ce priveşte matricea A, nu există nici un conflict la citire. Deoarece în instrucţiunea 2.1 ordinea de sumare este indiferentă, putem planifica operaţiile astfel încât, la un moment dat, fiecare element din x să fie citit de un alt procesor; cea mai la îndemână modalitate este ca procesorul P_i să înceapă sumarea din 2.1 de la $j = i$ şi să continue crescând j; după $j = n-1$, se continuă de la $j = 0$, deci ciclic.

ALGORITM 4.13 (gaxpy pe EREW PRAM, $p = n$, id $\equiv i$)

1. $z_i \leftarrow y_i$
2. **pentru** $j = i : n-1, \quad z_i \leftarrow z_i + a_{ij}x_j$
3. **pentru** $j = 0 : i-1, \quad z_i \leftarrow z_i + a_{ij}x_j$

Desigur că eficienţa acestui algoritm depinde în mod critic de sincronismul execuţării instrucţiunilor.

Cazul $p \ll n$. Numărul de operaţii efectuate fără necesitatea sincronizării este acum mai mare. Dacă notăm cu $\mathcal{L}_i$ mulţimea liniilor de care se ocupă procesorul P_i (de exemplu, de la in/p la $(i+1)n/p - 1$), atunci calculele se desfăşoară în felul următor:

ALGORITM 4.14 (gaxpy pe EREW PRAM, $p \ll n$, id $\equiv i$)

1. **pentru** $i \in \mathcal{L}_i$, $z_i \leftarrow y_i$
2. **pentru** $j = i : n-1$
    1. **pentru** $i \in \mathcal{L}_i, \quad z_i \leftarrow z_i + a_{ij}x_j$
3. **pentru** $j = 0 : i-1$
    1. **pentru** $i \in \mathcal{L}_i, \quad z_i \leftarrow z_i + a_{ij}x_j$

Este de remarcat ordinea buclelor, care face ca un procesor să execute succesiv toate citirile unui element din x; pentru a micşora şi mai mult riscul conflictelor, se poate pune imediat înainte de instrucţiunile 2.1 şi 3.1 un transfer de tipul $w_i \leftarrow x_j$, care mută elementul curent din x într-o locaţie de memorie "proprie" procesorului P_i.

Toţi algoritmii de mai sus decurg în perfect paralelism; deci, dacă $p = n$, atunci timpul de execuţie este $T(n) = 2n$; dacă $p \ll n$, atunci $T(n) = 2n^2/p$.

Algoritmi pentru MIMD cu memorie distribuită

Repartizare pe linii a matricei A. Ne ocupăm în special de cazul $p = n$, generalizarea fiind simplă. O primă idee e de a utiliza planificarea în care procesorul P_i calculează z_i; la fel am procedat şi în cazul memoriei comune. Implicaţia cea mai importantă este asupra repartizării datelor. Aşa cum rezultată din relaţia (4.7) şi din figura 4.8, procesorul P_i ar trebui să aibă în memoria locală linia i din matricea A, elementul y_i şi vectorul x în întregime. Pe de altă parte, este naturală nu numai o repartiţie echilibrată a datelor, cum e cea de mai sus, dar şi una fără alocarea unei aceleiaşi date mai multor procesoare. Pentru a satisface şi acest din urmă considerent, repartizarea iniţială a datelor este una pe linii, pentru A, x şi y; de exemplu, P_i deţine linia i din A, x_i şi y_i. Rezultatul va fi şi el distribuit în aceeaşi manieră, deci P_i va deţine în final z_i.

Un algoritm imediat este următorul: fiecare procesor participă la o operaţie de comunicaţie globală în urma căreia va avea întregul x în memoria locală. Apoi, având toate datele necesare, aplică (4.7) pentru a calcula z_i. Deoarece fiecare P_i posedă x_i, comunicaţia globală constă în trimiterea acestui element tuturor celorlalte procesoare, deci este vorba de o difuzare generală. Pe scurt, algoritmul este următorul (peste tot în această secţiune folosim indici globali):

ALGORITM 4.15 (gaxpy pe MIMD cu memorie distribuită, $p = n$, id $\equiv i$, repartizare pe linii pentru A, x, y)

1. **difuzare generală**: P_i trimite x_i tuturor celorlalte procesoare şi recepţionează restul elementelor din x
2. $z_i \leftarrow y_i + \sum_{j=0}^{n-1} a_{ij}x_j$ (sau $z(i) \leftarrow y(i) + A(i,:) \cdot x$)

În instrucţiunea 2 am prezentat şi o alternativă de scriere a relaţiei (4.7), în stilul limbajului Matlab; $x(i)$ este acelaşi cu x_i, iar $A(i,:)$ reprezintă linia i din matricea A (indicele ':' are semnificaţia de "tot", în cazul nostru toate coloanele $j = 1 : n$).

Operaţiile din algoritmul 4.15 decurg în perfect paralelism; apare însă un overhead datorat comunicaţiei, a cărui mărime depinde de topologia arhitecturii distribuite (cu atât mai mic cu cât conectivitatea este mai bună). Timpul de execuţie este de $2n\alpha$ pentru operaţii aritmetice, şi, de exemplu pentru inel, de aproximativ $n/2(\sigma + \beta)$ pentru comunicaţie (vezi timpii de difuzare generală din secţiunea 3.3.3).

Acest algoritm are defectul că un procesor trebuie să memoreze întregul vector x; în unele cazuri (pentru dimensiuni mari ale datelor), este de dorit evitarea acestei situaţii. Ne propunem deci să găsim un algoritm în care fiecare procesor să nu aibă la

un moment dat decât un singur element din vectorul x. Ideea este destul de simplă: pe parcursul operaţiei de difuzare generală a vectorului x, fiecare procesor efectuează operaţiile care implică elementele din x primite la un moment dat, după care trimite mai departe acele elemente către procesoare care nu le-au primit încă, însă *fără a le memora*; acest proces se repetă până la recepţionarea tuturor elementelor din x.

Vom exemplifica aceasta pe inel, în cazul celui mai simplu algoritm cu putinţă, cel adecvat modelului half duplex. Deci, difuzarea generală decurge astfel: P_i posedă un element din x, pe care-l trimite vecinului din stânga; aceasta operaţie se repetă de $p-1$ ori (până când fiecare element ajunge la vecinul din dreapta al posesorului său iniţial, după ce a parcurs întreg inelul). Procesorul P_i, primind la un moment dat x_j, execută operaţia $z_i \leftarrow z_i + a_{ij}x_j$ (partea care implică x_j din 4.7) şi trimite x_j mai departe, adică celuilalt vecin. În formă compactă, aceasta se poate scrie astfel:

ALGORITM 4.16 (gaxpy pe inel, $p = n$, id $\equiv i$, repartizare pe linii pt. A, x, y)

1. $z_i \leftarrow y_i$, $j \leftarrow i$
2. **pentru** $k = 0 : p-1$
   1. $z_i \leftarrow z_i + a_{ij}x_j$
   2. **dacă** $k < p-1$ **atunci**, **în paralel**
      1. **send**(x_j, stânga)
      2. **recv**($x_{(j+1) \bmod p}$, dreapta)
   3. $j \leftarrow (j+1) \bmod p$

Un procesor nu trebuie să folosească decât două variabile locale pentru valorile x_j recepţionate şi retransmise (deoarece transmisia şi recepţia se fac simultan); memoria locală folosită pentru vectorul x este mult mai mică decât în algoritmul 4.16. Pentru o versiune a algoritmului 4.16 folosind numai indici locali, vezi problema 4.3.19.

Ultimii doi algoritmi se pot uşor generaliza pentru cazul în care $n \gg p$. Fiecare procesor deţine n/p elemente din x şi toate operaţiile făcute mai sus cu un singur element x_i se vor face cu toate aceste n/p elemente. Vezi problema 4.3.20. De asemenea, generalizarea este banală şi în cazul matricelor dreptunghiulare.

Repartizare pe coloane a matricei A. Ce se întâmplă dacă presupunem o repartizare pe coloane a matricei A ? (Evident, x şi y, fiind vectori, îşi păstrează repartizarea.) Întrebarea nu este fără sens; produsul matrice-vector apare de obicei într-un context mai general, ca etapă în rezolvarea unei probleme şi nu ca scop în sine; de aceea, repartizarea matricei A poate fi impusă de împrejurări.

Rămânând la cazul $n = p$, procesorul P_k va avea coloana k din A. Pentru o linie i oarecare, $0 \leq i \leq n-1$, procesorul P_k poate contribui la calculul $z_i = y_i + \sum_{j=0}^{n-1} a_{ij}x_j$ din relaţia (4.7) doar cu efectuarea operaţiei $s_{ik} = a_{ik}x_k$, restul elementelor din A nefiindu-i locale. Dacă fiecare procesor procedează la fel, calculul elementului z_i se reduce la efectuarea unei sume pe ansamblul procesoarelor. Modul de calcul al produsului matrice-vector este sugerat în figura 4.9 şi este denumit prin *sumă vectorială*, deoarece, după efectuarea produselor locale, calculul lui z se reduce la o sumă de vectori; în figură este reprezentată doar partea ce revine procesorului P_k.

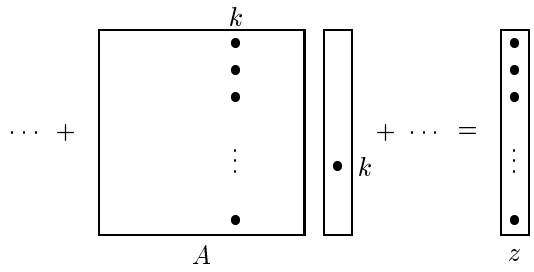


Figura 4.9: Produsul matrice-vector calculat prin sumă vectorială.

O metodă simplă de a realiza suma $z_i = y_i + \sum_{j=0}^{n-1} s_{ij}$ este colectarea tuturor valorilor s_{ij} în P_i (care trebuie să posede z_i, în final) şi apoi efectuarea locală a sumei. Cum au loc p colectări simultane (pentru calculul tuturor elementelor din z), operaţia de comunicaţie globală este un schimb complet: procesorul P_i primeşte valoarea s_{ij} de la P_j, pentru orice $i, j \in 0 : p-1$. Algoritmul este următorul.

ALGORITM 4.17 (gaxpy pe MIMD cu memorie distribuită, $p = n$, id $\equiv k$, repartizare pe coloane pentru A)

1. **pentru** $i = 0 : n-1$
   1. $s_{ik} \leftarrow a_{ik} x_k$
2. **schimb complet**: P_k trimite s_{ik} lui P_i, cu $i = 0 : p-1$
3. $z_k \leftarrow y_k + \sum_{j=0}^{n-1} s_{kj}$

Fiecare procesor efectuează n produse şi n sume, ca în algoritmul 4.15. Volumul comunicaţiei este însă mai mare, în general, deoarece aici este vorba de un schimb complet, fiecare procesor având n elemente, pe când acolo se făcea o difuzare generală, fiecare procesor având un singur element.

În plus, acest algoritm are acelaşi dezavantaj ca şi 4.15: memoria locală ocupată este relativ mare; este necesară, pe lângă datele iniţiale, şi memorarea tuturor valorilor s, adică încă n pentru fiecare procesor.

Dar schimbul complet se poate face cu acumulare (procesorul P_i nu are nevoie de toate elementele s_{ij}, ci de suma lor); două valori, să zicem s_{iq} şi s_{ir}, ambele cu destinaţia P_i (transmise de P_q şi, respectiv, P_r), pot fi adunate pe traseu, într-un procesor în care se află amândouă la un moment dat, mai departe fiind transmisă doar suma lor. În fond, este acelaşi lucru cu a efectua p sume în paralel, pentru fiecare sumă destinaţia fiind un alt procesor. Aceasta reduce şi memoria locală ocupată, şi volumul comunicaţiei, după cum se va vedea mai jos.

De exemplu, pe inel, o sumă se poate calcula foarte simplu transmiţând datele într-un singur sens; fiecare procesor recepţionează din dreapta o valoare (sumă parţială), o adună la cea proprie şi trimite rezultatul mai departe spre stânga; altfel zis, suma circulă pe inel, îmbogăţindu-se la fiecare pas cu o nouă valoare. În acest fel se pot

calcula p sume în paralel, fără nici un conflict de comunicaţie; la fiecare pas, pentru sume diferite sunt utilizate legături între perechi diferite de procesoare.

Algoritmul este echivalent cu a "plimba" vectorul z pe inel, fiecare procesor deţinând la un moment dat un element. La fiecare pas de comunicaţie, un element z_i se actualizează pe baza datelor oferite de procesorul la care se află; pentru a micşora memoria locală ocupată, procesorul P_k nu mai calculează de la început toate valorile s_{ik} (cu $0 \leq i < n$), ca în algoritmul 4.17, ci pe parcurs, pe măsură ce sunt necesare în actualizarea unui element z_i. Mai detaliat, forma algoritmului este:

ALGORITM 4.18 (gaxpy pe inel, $p = n$, id $\equiv k$, repartizare pe coloane pentru A)

1. $i \leftarrow (k+1) \bmod p$, $z_i \leftarrow 0$
2. **pentru** $l = 0 : p-1$
   1. $z_i \leftarrow z_i + a_{ik} x_k$
   2. **dacă** $l < p-1$ **atunci**, **în paralel**
      1. **send**(z_i, stânga)
      2. **recv**($z_{(i+1) \bmod p}$, dreapta)
   3. $i \leftarrow (i+1) \bmod p$
3. $z_k \leftarrow z_k + y_k$

Deoarece z_k parcurge inelul spre stânga, procesorul de la care porneşte (pentru a ajunge în final la P_k) este vecinul din stânga al lui P_k, care are adresa $(k-1) \bmod p$. Privind din punctul de vedere al procesoarelor, P_k deţine iniţial elementul din z care va ajunge în cele din urmă la vecinul său din dreapta, i.e. la procesorul $z_{(k+1) \bmod p}$, ceea ce explică instrucţiunea 1. Datorită acestui mod de comunicare, operaţia de adunare cu y din instrucţiunea 3 trebuie amânată la sfârşit, după ajungerea elementului z_k în P_k.

Algoritmii 4.18 şi 4.16 au acelaşi timp de execuţie atât pentru pentru operaţii aritmetice, cât şi pentru comunicaţie.

Pe alte topologii, ideile sunt asemănătoare, doar implementarea este mai complicată. Este de subliniat că, atunci când există rutine eficiente de comunicaţie globală şi când memoria locală nu este o problemă, atunci algoritmii generali 4.15 (în special) şi 4.17 sunt recomandaţi, datorită simplităţii lor.

Repartizare bidimensională a matricei A. Vom încheia acest paragraf, prezentând o variantă de înmulţire matrice-vector pe tor. Pentru simplitatea discuţiei, considerăm cazul în care fiecare procesor deţine un element al matricei A, i.e. avem $p = n^2$ şi procesorul P_{ij} deţine elementul a_{ij}. Vectorii x, y şi z sunt repartizaţi pe o singură dimensiune, de exemplu pe prima linie sau prima coloană de procesoare din tor. O repartizare echivalentă, la care se ajunge imediat din cele anterioare, dar care prezintă avantaje în cazul înmulţirii matrice-vector, este pe diagonala torului, i.e. procesorul P_{ii} deţine elementele x_i, y_i şi, în final, z_i.

Pentru a vedea structura comunicaţiei necesară efectuării produsului Ax, să observăm că, în mod natural, procesoarele de pe linia i a torului cooperează pentru calculul elementului z_i. Fiecare procesor efectuează produsul $s_{ij} = a_{ij} x_j$. Pentru

aceasta, procesorul P_{ij} are nevoie de elementul x_j, aflat la P_{jj}; în concluzie, este necesară o difuzare a lui x_j pe coloana j a torului, efectuată de procesorul diagonal de pe acea coloană. După calculul produsului s_{ij}, procesoarele de pe linia i a torului cooperează în calculul sumei $x_i = \sum_{j=0}^{n-1} s_{ij}$, cu destinaţia în procesorul P_{ii}. Aşadar, algoritmul are structura următoare.

ALGORITM 4.19 (gaxpy pe tor, $p = n^2$, pentru procesorul P_{ij}, care deţine a_{ij}; procesorul diagonal P_{ii} deţine x_i, y_i şi, în final, z_i)

1. **difuzare** pe coloane: P_{ij} recepţionează x_j de la P_{jj}
2. $s_{ij} \leftarrow a_{ij} x_j$
3. **sumă** pe linii: procesorul P_{ii} colectează $x_i = \sum_{j=0}^{n-1} s_{ij}$
4. **dacă** $i = j$ **atunci** $x_i \leftarrow x_i + y_i$

Se observă că operaţiile sunt echilibrate între procesoare. Comunicaţia se desfăşoară pe grupuri de procesoare, aflate pe aceeaşi linie sau coloană a torului.

Cazul $p \ll n$. Generalizarea algoritmilor anteriori pentru cazul $p \ll n$ este extrem de simplă. Vom exemplifica doar pentru repartizarea bidimensională. În loc să repartizăm individual fiecare element al matricei A, considerăm blocuri A_{ij} de dimensiune $r \times r$; fiecare procesor primeşte unul sau mai multe blocuri. De asemenea, vectorii x, y şi z sunt împărţiţi in blocuri de lungime r, notate X_i, Y_i, respectiv Z_i. Se poate scrie o formulă perfect analoagă cu (4.7), dar conţinând nu operaţii elementare, ci cu blocuri (matrice sau vectori) de dimensiune r, i.e.

$$Z_i = Y_i + \sum_{j=0}^{m} A_{ij} X_j, \text{ cu } 0 \leq i \leq m,$$

unde $m = n/r$ este numărul de blocuri pe orizontală sau verticală. Orice algoritm scris la nivel de element se poate generaliza imediat la nivel de bloc, atât în privinţa calculelor cât şi a comunicaţiei. Capitolul 6 va prezenta numeroase exemple de algoritmi la nivel de bloc, care sunt preferaţi în practică datorită eficienţei lor.

Probleme

P 4.3.1 Cum se poate calcula produsul scalar a doi vectori, x şi y ?

P 4.3.2 Care este numărul de procesoare pentru care algoritmul 4.2 este optimal ? (Adică, se execută în timp $O(\log n)$ şi are costul de $O(n)$.)

P 4.3.3 După iteraţia k a algoritmului 4.3, suma căror elemente este locală procesorului P_i ?

P 4.3.4 Scrieţi un program MPI care să implementeze algoritmul 4.3.

P 4.3.5 Adaptaţi algoritmul 4.1 astfel încât, folosind prima variantă a instrucţiunii 1.1.1, comunicaţia implicată de implementarea pe un hipercub să aibă loc numai între vecini.

P 4.3.6 Aplicaţi ideea colectării după o familie de arbori rotativi, la fiecare pas după o direcţie, pentru calculul sumei a $n \gg p$ numere.

P 4.3.7 Sugeraţi un algoritm de calcul al sumei pe inel.

P 4.3.8 Cum se poate calcula suma globală pe un inel ? Dar mai multe sume ?

P 4.3.9 Se poate elimina o operaţie din algoritmul 4.6 ?

P 4.3.10 Dacă $g(i)$ este codul Gray al numărului i, demonstraţi că, pe un hipercub, distanţa dintre nodurile cu adresele $g(i)$ şi $g(i+2^l)$ este cel mult 2.

P 4.3.11 Modificaţi algoritmul 4.9 a.î. să fie corect şi pentru operaţii necomutative.

P 4.3.12 Se mai poate reduce timpul de execuţie al algoritmului 4.10 ?

P 4.3.13 Scrieţi versiunea algoritmului reducerii ciclice 4.10 pentru cazul $n \gg p$.

P 4.3.14 Fie şirul recurent definit prin relaţia $x_{i+1} = (a_i x_i + b_i)/(c_i x_i + d_i)$, în care a_i, b_i, c_i şi d_i sunt coeficienţi cunoscuţi; de asemenea, se cunoaşte valoarea x_0. Indicaţi un algoritm paralel pentru calculul termenului x_n.

P 4.3.15 Propuneţi un algoritm de rezolvare a unui sistem de ecuaţii liniare $Ax = b$, atunci când A este inferior bidiagonală ($a_{ij} = 0$ dacă $i < j$ sau $i > j+1$), de dimensiune $n \times n$.

P 4.3.16 Adaptaţi algoritmul 4.8, pentru calculul tuturor termenilor unui şir recurent de ordinul I.

P 4.3.17 Scrieţi un algoritm pentru calculul termenilor unui şir recurent pe un hipercub, inspirându-vă din algoritmul 4.9.

P 4.3.18 Să se adapteze algoritmul de difuzare generală 3.5 pentru realizarea comunicaţiei necesare în calcularea produsului matrice-vector pe un inel.

P 4.3.19 Să se scrie o versiune a algoritmului 4.16 folosind doar indici locali.

P 4.3.20 Să se generalizeze algoritmul 4.16, de calcul al produsului matrice-vector pe un inel, la cazul $n \gg p$; consideraţi şi cazul în care n nu se divide cu p.

P 4.3.21 Care dintre algoritmii 4.16 şi 4.18 este mai eficient pentru înmulţirea matrice-vector pe un inel, atunci când matricea A este dreptunghiulară, de dimensiune $m \times n$; se presupun $m \neq n$ şi $m, n \gg p$.

Capitolul 5

Sortare şi probleme conexe

Fie $A = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ o secvenţă de n elemente ale unei mulţimi care admite o relaţie de ordine; a_i pot fi numere de diverse tipuri, şiruri de caractere etc.; a_i vor fi uneori numite şi *chei*, deoarece ele pot face parte din înregistrări cuprinzând mai multă informaţie—numai cheile fiind folosite pentru ordonare. Pentru simplitate, vom presupune că cele n elemente sunt distincte[1]; mai mult, de multe ori vom presupune că A este o permutare a secvenţei $\langle 0, 1, \ldots, n-1 \rangle$ (atenţie însă, această ipoteză este făcută doar pentru a uşura prezentarea; dacă se cunosc informaţii suplimentare asupra elementelor secvenţei A, sortarea se poate simplifica mult; noi nu vom profita de aceasta). Dacă nu este specificat altfel, vom considera că n este o putere a lui 2. Algoritmii prezentaţi sunt valabili, uneori cu modificări minore (de "bucătărie", lăsate cititorului, de obicei), şi în absenţa tuturor acestor simplificări.

Problemele de care ne vom ocupa în acest capitol sunt următoarele:

- problema sortării—aceea de a ordona crescător (sau descrescător) secvenţa A, deci de a găsi $A' = \langle a_{i_0}, a_{i_1}, \ldots, a_{i_{n-1}} \rangle$ astfel încât $a_{i_j} < a_{i_k}$ (sau $a_{i_j} > a_{i_k}$) dacă $j < k$;

- problema găsirii minimului, deci a găsirii indicelui i_m pentru care $a_i > a_{i_m}$, oricare $i \neq i_m$;

- problema selecţiei—a găsirii elementului din poziţia j în secvenţa sortată; altfel zis, a elementului pentru care există fix $j-1$ alte elemente mai mici decât el (am precizat aceasta pentru a nu rămâne cumva impresia că secvenţa trebuie sortată pentru aflarea celui de-al j-lea element);

[1] Matematic vorbind, e uşor să diferenţiem elementele egale dintr-o secvenţă; în loc de elementul a_i vom considera perechea (a_i, i); în orice secvenţă vor fi atunci perechi distincte. Algoritmic vorbind, procedeul este neplăcut, implicând calcule şi memorie suplimentare; e de preferat să ne descurcăm fără acest subterfugiu.

- problema găsirii rangului unui element, inversa precedentei; dându-se elementul $v \in A$, să se găsească poziţia sa în secvenţa sortată (sau, dacă vreţi, câte elemente sunt mai mici decât v);

- problema interclasării (fuziunii, engl. *merge*): dându-se două secvenţe crescătoare A şi B, de lungimi n, respectiv m, să se găsească secvenţa sortată $A \perp B$ care cuprinde cele $n + m$ elemente din A şi B.

Evident, vom insista asupra tratării paralele a problemelor. O imensă sursă de algoritmi secvenţiali este Knuth [17], apărută în 1976 şi în traducere românească. Surse de algoritmi paraleli vor fi citate pe parcurs; o bună lucrare pentru algoritmi pe PRAM este [16].

Pentru aprecierea complexităţii algoritmilor vom contoriza numai comparaţiile între chei (deci nu vom număra operaţiile de interschimbare a două elemente, aritmetica indicilor etc.); în acest capitol α va însemna timpul necesar pentru o comparaţie (el nu va fi folosit în formule decât atunci când este imperios necesar).

Mulţi dintre algoritmii prezentaţi în acest capitol au complexităţi care depind, uneori semnificativ, de ordinea elementelor din A; de aceea expresiile pe care le vom prezenta pentru acestea au o nuanţă probabilistă; va fi vorba deci despre timpi de execuţie în medie, în cel mai favorabil caz sau în cel mai defavorabil caz, cu precizarea cât de aproape de valorile respective se poate spera să fie timpul de execuţie real. Trebuie făcută totuşi o ipoteză (probabilistă) asupra ordinii din A: vom presupune că indicii elementelor din A sunt repartizaţi uniform, adică un element se poate afla, cu aceeaşi probabilitate, pe orice poziţie din A; aceasta este, de altfel, ipoteza uzuală.

Notaţii. În scrierea algoritmilor vom folosi notaţia $exch(a_i, a_j)$ sau—cu aceeaşi semnificaţie—$exch(i, j)$ (când va fi evident care este secvenţa implicată) pentru operaţia de comparaţie a elementelor a_i şi a_j, urmată de interschimbare dacă $a_i > a_j$; deci, în prima poziţie va ajunge elementul mai mic. Pe un calculator cu memorie comună, această operaţie va fi efectuată de un procesor (deci secvenţial). În cazul unui calculator cu memorie distribuită sunt două posibilităţi; dacă ambele elemente aparţin aceluiaşi procesor, operaţia este cea banală; dacă elementele aparţin unor procesoare diferite, atunci fiecare procesor comunică elementul său celuilalt, apoi fiecare face comparaţia şi reţine o valoare: primul minimul, al doilea maximul; se efectuează un pas de comunicaţie şi o comparaţie. Să observăm că în cazul memorie distribuită interschimbarea se face prin comunicaţie (şi se comunică indiferent de valorile elementelor), în timp ce pentru memoria comună prin trei atribuiri succesive (făcute însă numai dacă e cazul).

Dacă A_i şi A_j sunt două secvenţe, vom folosi următoarele notaţii: $\langle A_i, A_j \rangle$ este secvenţa obţinută prin concatenarea, în ordinea precizată, a secvenţelor A_i şi A_j; $A_i < A_j$ semnifică faptul că orice element din A_i este mai mic decât orice element din A_j; ca un caz particular, $v < A_i$, unde v este o cheie, înseamnă că v este mai mic decât orice element din A_i.

5.1 Găsirea minimului

Problema aflării elementului minim dintr-o secvenţă constituie un caz particular de reducere şi poate fi rezolvată prin algoritmii din secţiunea 4.3.1. Vom relua aici unii algoritmi şi vom prezenta şi alţii noi, care nu se pot generaliza pentru reducere.

Algoritmul secvenţial uzual este următorul: se compară primele două elemente; minimul se compară cu al treilea; noul minim cu al patrulea etc. Sunt necesare $n-1$ comparaţii, această valoare fiind optimă (fiecare comparaţie elimină un candidat la minim, şi trebuie eliminaţi $n-1$). În această formă, nu este nici urmă de paralelism.

Algoritm pe EREW PRAM. Găsirea minimului se poate face utilizând un arbore binar echilibrat de decizie, ca în figura 4.5, care are în cele n frunze elementele din A şi în care fiecare nod tată are asociată valoarea minimă a celor doi fii; altfel zis, algoritmul urmează principiul de organizare a unui turneu de tenis. Aşadar, se formează $n/2$ perechi de elemente şi se extrage din fiecare minimul; apoi, din aceste minime se formează $n/4$ perechi, ş.a.m.d. La pasul i, găsirea minimului din fiecare dintre cele $n/2^i$ perechi se poate face într-o singură operaţie de către $n/2^i$ procesoare. Algoritmul ce urmează este asemănător cu algoritmul 4.1 de calcul al sumei.

ALGORITM 5.1 (găsire minim pe EREW PRAM, $p = n/2$, id $\equiv k$)

1. **pentru** $i = 1 : \log n$
    1. **dacă** $k < n/2^i$ **atunci**
        1. $exch(k2^i, k2^i + 2^{i-1})$

În final minimul se găseşte în a_0. Deci, $T(n) = T(n/2) + 1$, adică $T(n) = \log n$, şi se folosesc $O(n)$ procesoare. Acest timp este optim; de altfel şi în modelul CREW PRAM, găsirea minimului necesită $\Omega(\log n)$ comparaţii.

Algoritm pe COMMON CRCW PRAM. În acest model, lucrurile nu mai sunt atât de evidente, chiar dacă simple. Se poate imagina un algoritm care să găsească minimul cu o singură comparaţie !

Să observăm întâi că putem compara simultan toate perechile posibile (a_i, a_j), cu $i \neq j$; pentru aceasta este nevoie de $\binom{n}{2} = n(n-1)/2$ procesoare. Ce facem însă cu rezultatele comparaţiilor ?

Fie un vector c de dimensiune n, iniţializat cu valoarea 1 peste tot. Procesorul care compară a_i cu a_j scrie 0 în poziţia din c corespunzătoare elementului mai mare dintre cele două. Nu există nici un conflict de scriere: dacă mai multe procesoare scriu în aceeaşi locaţie, ele scriu toate valoarea 0. În urma acestei operaţii, un singur element din c va păstra valoarea iniţială 1, anume cel din poziţia minimului; toate celelalte sunt superioare cel puţin unui alt element, deci cel puţin un procesor va scrie 0 în poziţia corespunzătoare din c.

Presupunând deci că sunt $p = n(n-1)/2$ procesoare, să scriem algoritmul pentru procesorul notat P_{ij} $(i > j)$:

ALGORITM 5.2 (găsire minim pe COMMON CRCW PRAM)

1. $c_i \leftarrow 1$; $c_j \leftarrow 1$
2. **dacă** $a_i > a_j$ **atunci** $c_i \leftarrow 0$
3. **altfel** $c_j \leftarrow 0$

Deci, o singură comparaţie e suficientă; e drept că numărul de procesoare e foarte mare.

Algoritmul de mai sus poate fi generalizat. La fiecare pas căutarea minimului va fi limitată la o mulţime S, care iniţial coincide cu A, iar în final conţine un singur element—minimul.

Fie x cel mai mic întreg astfel încât atunci când S este partiţionată în x blocuri de dimensiune $\lfloor m/x \rfloor$ sau $\lceil m/x \rceil$ (unde $m = |S|$), sunt suficiente p comparaţii pentru a compara elementele fiecărei perechi dintr-un acelaşi bloc. Astfel, în x paşi (folosind algoritmul 5.2 cu p procesoare, în fiecare bloc), este determinat minimul din fiecare bloc şi S se reduce la x elemente.

Pentru a compara într-un pas toate elementele dintr-un bloc sunt necesare $\binom{m/x}{2}$ procesoare, deci $p = x\frac{m/x(m/x-1)}{2}$ şi, în fine, $x = \frac{m^2}{m+2p}$.

Obţinem deci relaţia de recurenţă $T(m) = T(m^2/(m+2p)+x$. Se poate arăta că, dacă $n/2 \leq p \leq \binom{n}{2}$, atunci numărul de operaţii pentru a reduce cardinalitatea lui S de la n la 1 este $T(n,p) = \log\log n - \log\log(2p/n) + O(1)$. Mai mult decât atât, acesta este şi optimul pentru găsirea minimului.

Algoritm pentru MIMD cu memorie distribuită. Găsirea minimului este asemănătoare colectării. Întâi, fiecare nod calculează minimul între cele $m = n/p$ elemente locale (în $m-1$ operaţii). Apoi, pe arborele de acoperire pe care se realizează colectarea, fiecare procesor aşteaptă minimele care vin dinspre frunze, le compară cu minimul local şi trimite mai departe elementul minim dintre acestea.

De exemplu, într-un hipercub, algoritmul de găsire a minimului de către P_0 folosind un arbore de acoperire binomial, este următorul (asemănător cu algoritmul 4.3):

ALGORITM 5.3 (găsire minim în hipercub, $p = 2^d$, $n \gg p$, id $\equiv k$)

1. găseşte μ, minimul elementelor locale
2. **pentru** $i = d-1 : -1 : 0$
3. **dacă** $k_i = 1$ **atunci**
    1. **send**(μ, i), **stop**
4. **altfel recv**(μ', i)
5. **dacă** $\mu' < \mu$ **atunci** $\mu \leftarrow \mu'$

Să observăm că procesorul P_k calculează elementul minim în subhipercubul $\bar{\mathcal{H}}_d^{d-r}(k)$, unde r este numărul de biţi consecutivi de 0 de la începutul reprezentării binare a lui k pe d biţi.

Timpul său de execuţie este $T(n) = (\frac{n-1}{p} + \log p)\alpha + \log p(\sigma + \beta)$. Se constată că $T(p) = O(\log p)$, deci algoritmul este optimal şi în ce priveşte numărul de comparaţii şi în ce priveşte comunicaţia.

Dacă se doreşte ca fiecare procesor să cunoască minimul, procesorul care l-a calculat va face o difuzare, sau se aplică ideea de la calculul sumei globale, i.e. algoritmul 4.4.

Probleme

P 5.1.1 Care este principalul motiv pentru care algoritmul secvenţial curent de găsire a minimului este cel amintit în text şi nu unul asemănător cu cel paralel pentru EREW PRAM (tip turneu de tenis) ?

P 5.1.2 Să se scrie algoritmul de găsire a minimului în modelul EREW PRAM atunci când sunt doar $p \ll n$ procesoare. Care este timpul de execuţie ? Dar eficienţa algoritmului ?

P 5.1.3 Scrieţi algoritmul 5.1 astfel încât, după pasul i, minimul să fie între primele $n/2^i$ elemente ale secvenţei A.

P 5.1.4 În algoritmul 5.2, în linia 1, sunt necesare două operaţii pentru iniţializarea vectorului c (e drept că ne putem permite, câtă vreme nu numărăm decât comparaţiile !); se poate cu una singură ?

P 5.1.5 a) Dacă în secvenţa A pot fi elemente egale, algoritmul 5.2 funcţionează corect ? b) Dar dacă în linia 2 se înlocuieşte $a_i > a_j$ cu $a_i \geq a_j$?

P 5.1.6 Demonstraţi că pentru $T(n,p) = \log\log n - \log\log(2p/n) + O(1)$ este adevărat $T(n, n(n-1)/2) = O(1)$.

P 5.1.7 Scrieţi algoritmul de găsire a minimului pe un inel.

P 5.1.8 Cu care dintre cei doi algoritmi de găsire a minimului pe un EREW PRAM—5.1 sau cel din problema **P5.1.3**—seamănă algoritmul 5.3 ? Scrieţi un algoritm care să semene cu celălalt.

5.2 Selecţia şi găsirea rangului

Aceste probleme apar de obicei în legătură cu sortarea şi mult mai rar de sine stătătoare; de aceea vom considera mai ales cazurile în care despre ordinea din A există informaţii suplimentare.

5.2.1 Rangul într-o secvenţă sortată

Ne vom ocupa întâi de găsirea rangului unei valori v într-o secvenţă A sortată crescător (v nu este neapărat un element din A). Secvenţial, problema este simplă şi se rezolvă printr-o căutare binară în $\log n$ comparaţii (se compară v cu elementul aflat la mijlocul secvenţei; se află astfel în care jumătate de secvenţă se găseşte v; se continuă în acea jumătate).

Rangul pe CREW PRAM. Pe un astfel de model arhitectural, rangul poate fi găsit în timp constant, cu n procesoare. Se foloseşte o variabilă logică suplimentară c, de lungime n. Procesorul P_k compară elementul a_k cu v şi scrie în c_k valoarea 0 dacă a_k este mai mic şi 1 dacă este mai mare. Cum secvenţa A este ordonată, în acest moment în c se vor afla r valori consecutive de 0, urmate de $n-r$ valori de 1. Procesorul P_k citeşte acum valorile c_k şi c_{k-1}; un singur procesor, anume P_r, va găsi două valori diferite (facem convenţia că $c_{-1} = 0$). Deci algoritmul este

ALGORITM 5.4 (rangul lui v în secvenţa sortată A pe CREW PRAM, $p = n$, id $\equiv k$; în final r conţine rangul)

1. **dacă** $a_k < v$ **atunci** $c_k \leftarrow 0$
2. **altfel** $c_k \leftarrow 1$
3. **dacă** $c_k \neq c_{k-1}$ **atunci** $r \leftarrow k$

Rangul pe MIMD cu memorie distribuită. În acest context, modificăm puţin problema, astfel: procesorul P_k deţine o secvenţă ordonată crescător A_k de lungime $m = n/p$, dar despre ordinea în $A = \langle A_0, \ldots, A_{p-1} \rangle$ nu se mai poate spune nimic altceva. Presupunem că valoarea v este locală procesorului P_0, ceea ce nu este o particularizare.

Ideea algoritmului este simplă; procesorul P_0 difuzează valoarea v tuturor procesoarelor; fiecare procesor calculează rangul local r_k, deci rangul lui v în secvenţa locală A_k; apoi se execută un algoritm de sumă globală, după care fiecare procesor va deţine rangul lui v. Algoritmul se bazează pe faptul evident că rangul (global) este suma rangurilor locale $r = \sum_{k=0}^{p-1} r_k$.

ALGORITM 5.5 (calculează r, rangul lui v în secvenţa A sortată local, pe MIMD cu memorie distribuită, $mp = n$, id $\equiv k$)

1. **difuzare**: P_0 trimite tuturor valoarea v
2. $r_k \leftarrow$ rangul lui v în A_k (prin căutare binară)
3. $r \leftarrow$ suma globală a valorilor r_k (cu algoritmul 4.4)

Algoritmul determină corect rangul chiar dacă A conţine chei egale (se calculează numărul de elemente strict mai mici decât v).

5.2.2 Selecţie pe PRAM

Trecem acum la problema selecţiei, găsirea elementului din poziţia j în secvenţa A sortată, pe care o vom rezolva în cazul cel mai general: nici o informaţie iniţială despre ordinea din A.

Algoritmul secvenţial utilizat de obicei este inspirat din cel de sortare rapidă (quicksort). Ideea este de separa A în două părţi, de a vedea în care din ele se găseşte elementul căutat, apoi de a continua căutarea numai în acea parte; o strategie divide et impera, aşadar. Întâi se alege un element $v \in A$, numit pivot în quicksort, dar pe care l-am putea numi candidat. Apoi se împarte secvenţa A în două: în prima parte (stânga) vor fi elementele mai mici decât v, în a doua (dreapta), elementele mai mari sau egale cu v; ca o mică observaţie, numărul de elemente din stânga este chiar rangul lui v în A; dacă în stânga sunt exact j elemente, v este elementul căutat; dacă sunt mai puţin de j elemente, căutarea va fi continuată în dreapta (candidatul la selecţie a fost prea mic), altfel se va căuta în stânga. Forma generală a algoritmului este (valabilă şi la implementarea paralelă):

ALGORITM 5.6 (algoritm general de selecţie a elementului din poziţia j)

1. $s \leftarrow 0,\ d \leftarrow n-1$
2. **cât timp** $s \leq d$ {secvenţa este nevidă}
   1. *alege_pivot*(s, d, j)
   2. $i_v =$ *partiţionare*(s, d) {formează cele două subsecvenţe}
   3. **dacă** $i_v = j$ **atunci stop**
   4. **altfel dacă** $i_v < j$ **atunci** $s \leftarrow i_v + 1$ {continuă în dreapta}
   5. **altfel** $d \leftarrow i_v - 1$ {continuă în stânga}

În acest algoritm, s şi d reprezintă indicele primului element, respectiv cel al ultimului, în secvenţa în care se caută. Candidatul v este calculat de procedura *alege_pivot* şi adus în poziţia s, printr-o eventuală interschimbare, deci $a_s = v$; în general se procedează astfel: se alege un eşantion de t elemente din secvenţa în care se caută, se ordonează acest eşantion şi se alege de obicei mediana[2]; există multe euristici de alegere a pivotului; mediana e un candidat bun pentru că se speră împărţirea secvenţei de căutare în două părţi de dimensiuni apropiate, înjumătăţind la fiecare pas dimensiunea problemei; cu cât t este mai mare, cu atât mediana eşantionului aproximează mai bine mediana secvenţei; timpul de calcul al medianei creşte, dar scade mai repede dimensiunea problemei; practic se alege $t = 3$ sau chiar $t = 1$. Procedura *partiţionare* calculează poziţia i_v a pivotului, permutând elementele în A astfel încât $a_i < v$, când $i < i_v$ şi $a_i \geq v$, când $i > i_v$; algoritmul secvenţial de partiţionare merită atenţia: vezi problema 5.2.4. În cazul înjumătăţirii dimensiunii, numărul de operaţii respectă relaţia $T(n) = T(n/2) + n$, deci $T(n) = 2n$. În cel mai rău caz se poate ajunge la $O(n^2)$ operaţii, dar aceasta este foarte improbabil; se poate conta, în medie, pe $O(n)$.

Selecţie pe CREW PRAM, $p = n$. Putem să aflăm într-un singur pas ce elemente sunt mai mari sau mai mici decât pivotul; în schimb, este dificil să punem unele din aceste elemente într-o zonă contiguă de memorie, pentru a continua căutarea. Prezentăm întâi algoritmul şi apoi explicaţiile.

ALGORITM 5.7 (selecţie a elementului din poziţia j în secvenţa A, pe CREW PRAM, $p = n$, id $\equiv k$)

1. $n' \leftarrow n,\ r \leftarrow 0$
2. **cât timp** $n' \neq 1$
   1. se alege v şi se pune în a_0 (deci $v \equiv a_0$)
   2. **dacă** $a_k < v$ **atunci** $c_k \leftarrow 1$ $(P_0 :\ c_0 \leftarrow 0)$, $e_k \leftarrow 1$, $f_k \leftarrow 0$
   3. **altfel** $c_k \leftarrow 0$, $e_k \leftarrow 0$, $f_k \leftarrow 1$ $(P_0 :\ f_k \leftarrow 0)$
   4. calculează $c_k \leftarrow \sum_{i=0}^{k} c_i$, cu algoritmul 4.8 (suma prefixelor)
   5. calculează $f_k \leftarrow \sum_{i=0}^{k} f_i$, cu algoritmul 4.8 (suma prefixelor)

[2]Sau elementul din poziţia $\lfloor t(j-s)/(d-s+1) \rfloor$, cel care se află într-o poziţie proporţională, în eşantion, cu elementul din poziţia j în A, în secvenţa de căutare, presupunând că ar fi sortată; adică $i/(j-s) \approx t/(d-s+1)$; primul este raportul poziţiilor, al doilea cel a lungimilor.

6. **dacă** $r + c_{n'-1} = j$ **atunci stop**
7. **dacă** $r + c_{n'-1} > j$ **atunci**
   1. **dacă** $e_k = 1$ **atunci** $b_{c_k-1} \leftarrow a_k$
   $(P_0 : n' \leftarrow c_{n'-1})$
8. **altfel** $\{r + c_{n'-1} < j\}$
   1. **dacă** $e_k = 0$ **atunci** $b_{f_k-1} \leftarrow a_k$
   $(P_0 : \ n' \leftarrow f_{n'-1}, r \leftarrow r + c_{n'-1})$
9. **dacă** $k \geq n'$ **atunci stop**
10. $a_k \leftarrow b_k$

Pentru a nu lungi prea mult descrierea, am apelat la următoarea convenţie: instrucţiunile care încep cu "P_0 :" sunt executate de P_0 în locul celor imediat precedente. n' este numărul de elemente rămase în zona de căutare; r este numărul de elemente mai mici decât candidatul curent v, care nu mai sunt în zona de căutare (au fost eliminate la iteraţii anterioare); acestea sunt variabile globale, citite de toată lumea şi modificate doar de P_0. Instrucţiunea 2.1, alegerea pivotului, nu este explicitată; important e să dureze $O(1)$; pentru simplitate ne putem gândi că v e chiar a_0. În 2.2, c_k, copia sa e_k şi negatul său f_k exprimă sensul comparaţiei între a_k şi v. În 2.4 se aplică suma prefixelor pentru c_k; elementele mai mici decât v au acum valori c_k diferite, şi anume chiar poziţiile lor într-o subsecvenţă conţinând doar valorile mai mici decât v (prima poziţie fiind 1). Să exemplificăm pentru $n = 8$:

k	0	1	2	3	4	5	6	7
a_k	5	1	8	9	3	2	6	4
c_k, e_k	0	1	0	0	1	1	0	1
f_k	0	0	1	1	0	0	1	0
$\sum_{i=0}^{k} c_k$	0	1	1	1	2	3	3	4
$\sum_{i=0}^{k} f_k$	0	0	1	2	2	2	3	3

În mod analog, suma prefixelor pentru f_k aduce în aceste variabile (când $a_k > v$) poziţia într-o subsecvenţă conţinând valorile mai mari decât v. În exemplu, secvenţa elementelor mai mici decât v este $\langle 1, 3, 2, 4\rangle$ (rangurile sunt în penultima linie), iar $\langle 8, 9, 6\rangle$ cea a celor mai mari (rangurile în ultima linie). Să observăm că, după instrucţiunea (3), $c_{n'-1}$ este chiar rangul lui v în secvenţa de căutare[3]; rangul în A este obţinut adunând la $c_{n'-1}$ numărul de elemente mai mici decât v deja eliminate, adică r. Dacă rangul este j algoritmul se termină. Dacă este mai mare, rămân în cursă elementele mai mici decât v, cele pentru care $e_k = 1$, în număr de $c_{n'-1}$; ele sunt copiate într-un zonă de memorie suplimentară B. În linia 2.8.1 se procedează analog în cazul în care rangul este mai mic decât j; în plus, se actualizează r, pentru că sunt $c_{n'-1}$ elemente mai mici decât oricare dintre cele între care se va continua căutarea,

[3]în treacăt am rezolvat deci şi problema rangului atunci când secvenţa A nu este sortată; timpul necesar este $O(\log n)$, cât pentru suma prefixelor; de fapt e suficientă o simplă sumă.

acestea din urmă fiind mutate începând de la poziţia 0 în A. La fiecare iteraţie sunt necesare n' procesoare; în 2.9 îşi încheie activitatea procesoarele cu adresă $\geq n'$. În 2.10, se recopiază elementele în A. Sunt necesare câteva sincronizări: înainte de 2.6, 2.9 şi 2.10.

Timpul de execuţie este dictat de suma prefixelor; în cazul favorabil al înjumătăţirii lui n' la fiecare iteraţie, avem $T(n) < T(n/2) + O(\log n)$, adică $T(n) = O(\log^2 n)$[4]. Costul algoritmului este de $O(n \log^2 n)$. Numărul minim de operaţii pentru selecţie (de fapt pentru aflarea medianei) s-a demonstrat a fi $\Omega(\log n / \log\log n)$, pe un CRCW PRAM, cu un număr mărginit polinomial de procesoare.

Selecţie pe CREW PRAM, $p \ll n$. Schiţăm mai jos modificările aduse algoritmului 5.7, păstrând structura lui.

ALGORITM 5.8 (selecţie pe CREW PRAM, $p \ll n$, id $\equiv k$)

1. $n' \leftarrow n$, $r \leftarrow 0$, $m \leftarrow n/p$, $\tilde{A}$ se află între poziţiile km şi $(k+1)m - 1$
2. **cât timp** $n' \neq 1$
   1. se alege v şi se pune în a_0 (deci $v \equiv a_0$)
   2. aplică algoritmul din **P5.2.4** pt. partiţionarea $\tilde{A} = \langle \tilde{A}_0, \tilde{A}_1 \rangle$, $\tilde{A}_0 < v \leq \tilde{A}_1$
   3. $c_k \leftarrow |\tilde{A}_0|$, numărul de elemente din $\tilde{A}$ mai mici decât v
   4. $f_k \leftarrow |\tilde{A}_1|$, numărul de elemente din $\tilde{A}$ mai mari decât v
   5. $c_{k+1} \leftarrow \sum_{i=0}^{k} c_i$ şi $f_{k+1} \leftarrow \sum_{i=0}^{k} f_i$, cu algoritmul 4.8 (puţin modificat)
   6. **dacă** $r + c_p = j$ **atunci stop**
   7. **dacă** $r + c_p > j$ **atunci**
      1. copiază $\tilde{A}_0$ în B, în poziţiile de la c_k la dreapta ($c_0 = 0$)
      2. **dacă** $k = 0$ **atunci** $n' \leftarrow c_p$
   8. **altfel** $\{r + c_p < j\}$
      1. copiază $\tilde{A}_1$ în B, în poziţiile de la f_k la dreapta ($f_0 = 0$)
      2. **dacă** $k = 0$ **atunci** $n' \leftarrow f_p$, $r \leftarrow r + c_p$
   9. $m \leftarrow \lceil n'/p \rceil$, $\tilde{A}$ se află între indicii (din A) km şi $\min((k+1)m - 1, n' - 1)$
   10. copiază în $\tilde{A}$ elementele corespunzătoare din B

Să exemplificăm valorile c_k şi f_k dintr-o iteraţie, în tabelul 5.1, pentru $p = 4$. Se observă că c_4 este numărul de elemente mai mici decât pivotul 13, iar f_4 numărul de elemente mai mari; cum c_4 este rangul pivotului în A, merită să ne amintim că rangul global este egal cu suma rangurilor locale c_i, cu $0 \leq i \leq 3$. În ultimele două linii se găsesc poziţiile de scriere în B.

Într-o iteraţie, fiecare procesor se ocupă de aproximativ $m = n'/p$ elemente succesive din A (notăm secvenţa acestor elemente cu $\tilde{A}$). De menţionat că, în 2.9, unele

[4] Timpul poate fi redus dacă mutăm 2.5 imediat înainte de 2.8.1; în acest caz, a doua sumă a prefixelor se va face doar dacă este nevoie. De altfel, a doua sumă a prefixelor nu este necesară; notând $d_k = \sum_{i=0}^{k} c_i$ şi $g_k = \sum_{i=0}^{k} f_i$, observăm că avem întotdeauna doar una din situaţiile (i) $d_k = d_{k-1} + 1$ şi $g_k = g_{k-1}$, sau (ii) $d_k = d_{k-1}$ şi $g_k = g_{k-1} + 1$. În consecinţă, g_k rezultă din d_k, d_{k-1} şi g_{k-1}.

k	0	1	2	3	4
$\tilde{A}$	$\langle 13,1,23,25\rangle$	$\langle 18,9,7,16\rangle$	$\langle 11,6,10,0\rangle$	$\langle 15,20,17,5\rangle$	
$\tilde{A}_0,\ \tilde{A}_1$	$\langle 1\rangle,\ \langle 23,25\rangle$	$\langle 7,9\rangle,\ \langle 18,16\rangle$	$\langle 11,6,10,0\rangle,\ \langle\rangle$	$\langle 5\rangle,\ \langle 20,17,15\rangle$	
c_k	1	2	4	1	
f_k	2	2	0	3	
$\sum_{i=0}^{k} c_k$	0	1	3	7	8
$\sum_{i=0}^{k} f_k$	0	2	4	4	7

Tabelul 5.1: Exemplu de execuţie pentru algoritmul 5.8.

procesoare pot rămâne cu o secvenţă $\tilde{A}$ vidă, spre finalul execuţiei algoritmului; altfel, numai ultimul procesor are ceva mai puţine elemente decât celelalte; în orice caz, procesoarele sunt aproximativ la fel de încărcate. Numărul de elemente mai mari sau mai mici decât v deţinute de un procesor se calculează secvenţial în algoritmul din **P5.2.4**. Suma prefixelor (algoritmul 4.8, modificat pentru că se scriu rezultatele în elementul din poziţia următoare) durează acum $O(\log p)$. Timpul mare de calcul este consumat în partiţionarea secvenţială; dacă pivotul este bine ales, avem $T(n) \leq T(n/2) + n/p + \log p$, deci $T(n) = 2n/p + \log p \log n$, ceea ce este foarte bine, al doilea termen (cel suplimentar unei perfecte distribuiri a operaţiilor celor p procesoare) fiind nesemnificativ.

5.2.3 Selecţie pe MIMD cu memorie distribuită

Modificăm puţin problema selecţiei, în sensul că presupunem secvenţele locale ca fiind ordonate; dorim în continuare aflarea celei de-a j-a valori din secvenţa A ordonată, s-o numim mai departe x; presupunem că elementele din A sunt distincte. Schiţam întai ideea algoritmului, după care vom prezenta detalii.

Un procesor propune un candidat v pentru x, după care, cu algoritmul 5.5, se calculează rangul acestui candidat în secvenţa A. Fiecare procesor posedă două variabile care delimitează spaţiul în care se caută x, ele conţinând indicii s şi d în secvenţa locală $\tilde{A}$, pentru care este sigur că $\tilde{a}_s \leq x \leq \tilde{a}_d$; valorile iniţiale sunt $s = 0$, $d = m-1$; o dată calculat rangul candidatului, fiecare procesor poate micşora spaţiul de căutare. Apoi este propus un nou candidat, ş.a.m.d., până când se găseşte x; condiţia de terminare este aceeaşi ca la PRAM: rangul candidatului să fie chiar j.

Avem de dat răspunsuri mai multor întrebări.

Cine propune candidaţii ? Este clar că dacă un procesor are $s > d$, el nu deţine x, deci nu are sens să propună un candidat. Aşadar, toate procesoarele trebuie să participe la propuneri; unul singur deţine x, dar nu se ştie care. Pot fi luate în considerare mai multe variante. De exemplu, la iteraţia i, propune procesorul P_i; există o valoare specială, să-i spunem NIL, care este propusă atunci când $s > d$, şi care înseamnă "nu am candidat". Mecanismul pare destul de greoi; dacă rămân

doar puţine procesoare pentru care $s < d$, ele trebuie să le aştepte pe celelalte, care difuzează valori NIL. Vom alege o altă metodă: fiecare procesor propune câte un candidat la fiecare iteraţie; procesorul P_0 colectează aceşti candidaţi, elimină valorile NIL, apoi selectează unul singur, v, de exemplu mediana (elementul din poziţia din mijloc în secvenţa sortată a candidaţilor), pe care-l difuzează. Această metodă nu e evident superioară anterioarei; are totuşi avantajul că la fiecare iteraţie se propune un candidat; o adoptăm din motive ce se vor vedea abia în secţiunea dedicată sortării rapide.

Ce valori propune un procesor drept candidaţi ? Iniţial, cea mai plauzibilă valoare este cea din poziţia $\lfloor j/n \rfloor$ (reamintim că $n = mp$ este lungimea totală a secvenţei. Apoi, în iteraţiile următoare, fiecare procesor va propune valoarea mediană din spaţiul său de căutare, cea cu indicele $\lfloor (s+d)/2 \rfloor$ în secvenţa locală.

Cum se reduce spaţiul de căutare ? În procesul de detectare a rangului candidatului, un procesor determină un indice l astfel încât $\tilde{a}_{l-1} < v \leq \tilde{a}_l$; l este chiar rangul local. După calculul rangului global r, sunt posibile trei cazuri. Fie $r = j$ şi atunci soluţia a fost găsită. Fie $r < j$ şi atunci candidatul a fost prea mic; se elimină toate elementele mai mici decât candidatul, deci $s \leftarrow l$ (sau $l+1$ dacă $v = \tilde{a}_l$; mai concis, elementul cu cel mai mic indice, mai mare decât v). Fie $r > j$ şi deci candidatul a fost prea mare; atunci $d \leftarrow l - 1$.

Care este elementul din poziţia j ? Ultimul candidat v (deci valoarea v la sfârşitul algoritmului).

Vom numi P_0 procesorul *conducător* al selecţiei, datorită rolului său special în acest algoritm. Este evident că oricare procesor poate avea rolul de conducător într-o arhitectură simetrică (inel, tor, hipercub; dar nu şi grilă), fără a se modifica numărul de operaţii.

Să sintetizăm toate acestea într-un algoritm; r este rangul global, fiecare procesor deţinând câte o copie a acestei valori.

ALGORITM 5.9 (determină elementul din poziţia j în secvenţa A sortată local, pe MIMD cu memorie distribuită, $mp = n$, id $\equiv k$, procesor conducător P_0)

1. $s \leftarrow 0$, $d \leftarrow m-1$, $c \leftarrow \tilde{a}_{\lfloor j/p \rfloor}$, $r \leftarrow -1$ {Iniţializare}
2. **cât timp** $r \neq j$
   1. **colectare**: P_0 primeşte candidaţii c de la celelalte procesoare
   2. P_0 elimină valorile NIL, ordonează restul candidaţilor
      şi alege v mediana lor
   3. $r \leftarrow$ rangul global al lui v (se aplică algoritmul 5.5)
   4. **dacă** $r \neq j$ **atunci**
      {reduce spaţiul local de căutare}
      1. **dacă** $r < j$ **atunci** {candidat prea mic}
         1. $s \leftarrow$ cel mai mic indice pentru care $\tilde{a}_s > v$
      2. **altfel** $d \leftarrow$ cel mai mare indice pentru care $\tilde{a}_d < v$

      {propune un nou candidat}
      3. **dacă** $s \leq d$ **atunci**

1. $c \leftarrow \tilde{a}_{\lfloor (s+d)/2 \rfloor}$ {mediana în spaţiul de căutare}
4. **altfel** $c \leftarrow NIL$

În instrucţiunea 2.2, deşi nu am explicitat, este clar că procesorul P_k nu face nimic, când $k \neq 0$; în schimb, la 2.1 au loc comunicaţii globale, deci P_k participă.

Să apreciem calitativ timpul de execuţie pentru o singură iteraţie, pe un hipercub. Instrucţiunile 1, 2.4.1.1 şi 2.4.2 se execută în timp constant, instrucţiunea 2.1 în $O(p)$, 2.2 în $O(p \log p)$ (dar vom vedea că găsirea secvenţială a medianei necesită doar $O(p)$) şi 2.3 în $O(\log(n/p)$. Adunând, putem aproxima timpul necesar unei iteraţii la $O(\log n)$ pentru comparaţii şi $O(p)$ pentru comunicaţie.

A determina exact numărul de iteraţii este imposibil. La fiecare iteraţie, cel puţin un procesor îşi înjumătăţeşte spaţiul de căutare; deci, în cel mai defavorabil caz, au loc $p \log(n/p)$ iteraţii; aceasta este însă o apreciere foarte pesimistă. În general, spaţiul de căutare al fiecărui procesor este redus la fiecare iteraţie, deci se poate aprecia că, în medie, numărul de iteraţii este de $O(\log n)$. Să remarcăm că în cazul algoritmului 5.8, problema dezechilibrului de încărcare se rezolva la fiecare iteraţie; aici ar fi dificil, comunicaţia implicată fiind foarte complexă.

Complexitatea algoritmului este aşadar de $O(\log^2 n)$ pentru comparaţii şi de $O(p \log n)$ pentru comunicaţie.

Probleme

P 5.2.1 Cu ce amendamente este valabil algoritmul 5.4 pe un EREW PRAM ?

P 5.2.2 Scrieţi algoritmul de găsire a rangului unei valori v într-o secvenţă ordonată A de lungime n, în contextul MIMD cu memorie distribuită. Se presupune că procesorul P_i are în memoria locală secvenţa A_i de lungime $m = n/p$ şi că $A_i < A_j$, dacă $i < j$.

P 5.2.3 Scrieţi un algoritm pentru calculul rangului pe o arhitectură cu memorie distribuită, atunci când A este o secvenţă oarecare (nu este ordonată local).

P 5.2.4 Detaliaţi procedura *partiţionare* din algoritmul 5.6.

P 5.2.5 Care este condiţia de terminare a algoritmului 5.9 dacă în secvenţa A pot exista elemente cu aceeaşi valoare ? Ce modificări mai trebuie aduse algoritmului ?

P 5.2.6 Cum trebuie modificat algoritmul 5.9 dacă secvenţele locale nu sunt ordonate ?

5.3 Interclasarea

Fie secvenţele $\langle a_0, a_1, \ldots, a_{n-1} \rangle$ şi $\langle b_0, b_1, \ldots, b_{m-1} \rangle$, cu $n \leq m$. Algoritmul secvenţial de interclasare a lor decurge în felul următor; se folosesc doi indici, unul în A, unul în B, iniţializaţi cu 0; se compară elementele având aceşti indici; cel mai mic dintre ele este adăugat secvenţei $A \perp B$ (iniţial vidă); indicele corespunzător secvenţei în care s-a găsit elementul minimul este incrementat; se continuă până când primul dintre indici depăşeşte lungimea secvenţei. Se fac cel puţin $\min(n, m)$ comparaţii şi cel mult $n + m - 1$. (Deşi răspunsul va fi dat pe parcursul acestei secţiuni, gândiţi-vă de acum la un algoritm paralel de interclasare pentru două procesoare.)

5.3.1 Complexitate paralelă teoretică

Se pot interclasa două secvenţe în timp constant, pe un CREW PRAM ? Da, dacă folosim algoritmul 5.4 de găsire a rangului într-o secvenţă ordonată. Dacă se calculează rangul r_i al fiecărui element a_i din A între elementele din B, atunci poziţia lui a_i în $A \perp B$ este $i + r_i$ (sunt i elemente în A şi r_i elemente în B mai mici decât a_i); numărul de procesoare necesar este nm, câte m pentru calculul rangului fiecărui element din A. Se procedează analog cu elementele din B.

ALGORITM 5.10 (interclasare pe CREW PRAM, $p = nm$, în final $D = A \perp B$)

1. **pentru** $i = 0 : n-1$, **în paralel**
   1. câte m procesoare calculează r_i, rangul lui a_i în B, cu algoritmul 5.4
   2. un procesor din fiecare grup de m execută: $d_i \leftarrow a_{i+r_i}$
2. **pentru** $j = 0 : m-1$, **în paralel**
   1. câte n procesoare calculează r_j, rangul lui b_j în A, cu algoritmul 5.4
   2. un procesor din fiecare grup de n execută: $d_j \leftarrow b_{j+r_j}$

Dacă notăm cu P_{ij} procesoarele, o alocare a taskurilor pentru acest algoritm este simplă. Costul algoritmului este foarte mare, $O(n^2)$ comparaţii când $m = n$, ceea ce lasă acestui algoritm doar importanţă teoretică.

Algoritmul Valiant. Vom prezenta acum un algoritm de interclasare a două secvenţe, folosind cel mult $(n + m)/2$ procesoare, pe un CREW PRAM. Sursa este Kruskal [18].

ALGORITM 5.11 *(Valiant)* (interclasare $A \perp B$ pe CREW PRAM, $p = (n + m)/2$)

1. Împarte A în $\sqrt{n}$ secvenţe A_i de lungime $\sqrt{n}$ fiecare şi B în $\sqrt{m}$ secvenţe B_j de lungime $\sqrt{m}$ fiecare (ignorăm micile modificări necesare atunci când n şi m nu sunt pătrate perfecte).

2. Fie a_i^* ultimul element al secvenţei A_i şi b_j^* ultimul element al secvenţei B_j. **În paralel**, găseşte rangul fiecărui a_i^* în secvenţa $B^* = \langle b_0^*, \ldots, b_{\sqrt{m}-1}^* \rangle$, folosind algoritmul 5.4; altfel spus, pentru fiecare a_i^*, găseşte unicul indice $j(i)$ pentru care $b_{j(i)-1}^* < a_i^* < b_{j(i)}^*$ (convenim că $b_{-1}^* = -\infty$ şi $b_{\sqrt{m}}^* = \infty$).

3. **În paralel**, găseşte rangul fiecărui a_i^* în secvenţa $B_{j(i)}$.

4. repetă recursiv paşii 1-3, pentru interclasarea (în paralel) a fiecărei secvenţe A_i cu o secvenţă din B.

Să detaliem puţin. Scopul paşilor 1-3 este de găsi poziţia exactă a elementelor a_i^* în secvenţa B; în figura 5.1 secvenţele A şi B sunt simbolizate prin cele două linii orizontale; liniile oblice începând în a_i^* au capătul celălalt în b_i^a, care este cel mai mic element din B mai mare decât a_i^*; în figură, am considerat exemplul $a_0^* < b_0^* < a_1^* < b_1^* < b_2^* < a_2^* < a_3^* < b_3^*$.

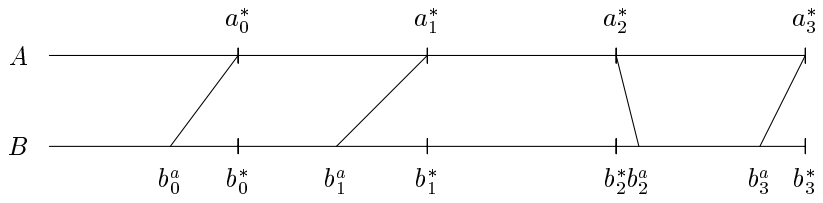


Figura 5.1: Separarea în subprobleme în algoritmul Valiant.

Găsirea fiecărui b_i^a se face în două etape; în prima se determină secvenţa B_j în care se găseşte b_i^a (o căutare la nivel de bloc, mai grosieră); este pasul 2 al algoritmului; el se execută în timp constant folosind $\sqrt{n}\sqrt{m}$ procesoare (se aplică de $\sqrt{n}$ ori, în paralel, algoritmul 5.4, care necesită $\sqrt{m}$ procesoare). În a doua (pasul 3 al algoritmului) se rafinează căutarea la nivel de element; efortul este identic cu cel de la pasul 2. În acest moment, problema iniţială a fost spartă în $\sqrt{n}$ probleme, în care fiecare A_i trebuie interclasat cu o secvenţă din B care începe cu elementul b_{i-1}^a (convenim că $b_{-1}^a = b_0$) şi se termină cu elementul precedent lui b_i^a; vom nota această secvenţă cu B_i'.

Numărul de comparaţii efectuate respectă regula recursivă $T(n) \leq T(\sqrt{n}) + O(1)$, ceea ce conduce la $T(n) = O(\log\log n)$. La primul nivel de recursie se folosesc $\sqrt{n}\sqrt{m} \leq (n+m)/2$ procesoare. În continuare, pentru fiecare subproblemă este adevărată o relaţie de acest gen, ceea ce conduce la un necesar de maximum $(n+m)/2$ procesoare.

Algoritmul acesta are dezavantajul de a nu împărţi în mod egal decât secvenţa A atunci când sunt formate subproblemele. Dacă aceasta nu afectează foarte mult timpul teoretic total (în general, dar nu şi în cel mai defavorabil caz), în schimb pune mari probleme în alocarea procesoarelor.

Se poate demonstra că problema interclasării a două secvenţe de lungime n se poate rezolva cu $O(n)$ procesoare în cel puţin $\Omega(\log\log n)$ operaţii, deci algoritmul Valiant este optim. În plus, costul lui este $O(n\log\log n)$, ceea ce e acceptabil.

5.3.2 Un algoritm pe CREW PRAM, pentru număr mic de procesoare

Să vedem cum se pot aplica ideile din algoritmul Valiant pentru cazul $p \ll n$. Să presupunem că împărţim A şi B în secvenţe de lungime egală, n/p, respectiv m/p; păstrăm notaţiile de mai sus; evident că acum $A^* = \langle a_0^*, \ldots, a_{p-1}^* \rangle$ şi $B^* = \langle b_0^*, \ldots, b_{p-1}^* \rangle$. Algoritmul are două etape.

ALGORITM 5.12 *(Valiant-Kruskal)* (interclasare $A \perp B$ pe CREW PRAM, $p \ll n$)

1. Găseşte poziţia exactă a fiecărui a_i^* în secvenţa B (formează secvenţele B_i').

2. Fiecare procesor P_i interclasează secvenţial secvenţele A_i şi B_i'.

Efortul mare de calcul se află în etapa a doua; putem spune că aceasta necesită $(n+m)/p$ comparaţii; numărul total de comparaţii care trebuie efectuat pentru interclasarea celor p perechi de secvenţe $A_i \perp B'_i$ este $n+m$ în cel mai defavorabil caz; presupunând secvenţele B'_i de lungimi aproximativ egale, rezultă complexitatea de mai sus.

Pentru prima etapă sunt mai multe variante posibile. Dacă fiecare P_i ar căuta separat poziţia lui a_i^* în B, aceasta s-ar face printr-o căutare binară, în $\log m$ comparaţii.

Se poate însă şi mai bine. Se interclasează A^* şi B^* folosind algoritmul Valiant, deci în timp $O(\log\log p)$ (cele două secvenţe au împreună $2p$ elemente; cu p procesoare se poate aplica algoritmul Valiant fără modificare). În acest moment s-a identificat, pentru fiecare a_i^*, segmentul $B_{j(i)}$ în care trebuie continuată căutarea. Fiecare procesor va căuta acum poziţia exactă pentru un a_i^* într-o secvenţă de lungime m/p, deci făcând $\log(m/p)$ comparaţii. Cum până acum am presupus că $n \leq m$, putem inversa rolurile între A şi B, şi deci timpul necesar primei etape va fi în acest caz $\log(n/p) + O(\log\log p)$, iar în total:

$$T(n,p) = (n+m)/p + \log(n/p) + O(\log\log p). \tag{5.1}$$

Vom face acum o mică paranteză, pentru a fi a mai clar cum se găseşte, după interclasarea secvenţelor A^* şi B^*, indicele $j(i)$, pentru fiecare a_i^*. Privind figura 5.1, reamintindu-ne că acolo $A^* \perp B^* = \langle a_0^*, b_0^*, a_1^*, b_1^*, b_2^*, a_2^*, a_3^*, b_3^* \rangle$ şi revăzând încă o dată pasul 2 al algoritmului Valiant, observăm că $j(0) = 0$, $j(1) = 1$, $j(2) = 3$, $j(3) = 3$; în general, $j(i)$ este indicele celui mai mic element din B^* mai mare decât a_i^* sau $j(i)$ este indicele primului element din B^* care se află în dreapta lui a_i^* în $A^* \perp B^*$; să notăm cu $rp(a_i^*)$ acest indice (rp vine de la *right pair*, perechea din dreapta). În mod analog, vom nota cu $lp(a_i^*)$ (*left pair*) indicele primului element din B^* care se află în stânga lui a_i^* în $A^* \perp B^*$. Se vede imediat că $rp(a_i^*) = lp(a_i^*) + 1$. Convenim ca $lp(a_i^*) = -1$ dacă nu există element din B^* mai mic decât a_i^*. Să mai observăm o relaţie interesantă (a cărei demonstraţie e lăsată cititorului):

$$rp(a_i^*) = rang(a_i^*) - i \tag{5.2}$$

unde rangul se calculează în secvenţa $A^* \perp B^*$.

Deci, în algoritmul Valiant, calculăm, pentru fiecare a_i^* rangul acestuia în $A^* \perp B^*$, ceea ce e echivalent cu sortarea acestei secvenţe (deci cu interclasarea), apoi $j(i) \equiv rp(a_i^*)$ se calculează cu (5.2). Acelaşi lucru se face şi în algoritmul Valiant-Kruskal.

5.3.3 Mediana secvenţei $A \perp B$ (partiţionare mediană)

Vom încerca să găsim un algoritm pentru calculul paralel al medianei secvenţei $A \perp B$, fără a face însă interclasarea; definim mediana secvenţei $A \perp B$ ca fiind valoarea cu rangul $\lceil (m+n)/2 \rceil - 1$. Soluţia prezentată este inspirată de [27].

Problema se poate formula altfel şi se numeşte partiţionare mediană. Să se partiţioneze A şi B în câte două secvenţe, $A = \langle A_0, A_1 \rangle$, respectiv $B = \langle B_0, B_1 \rangle$,

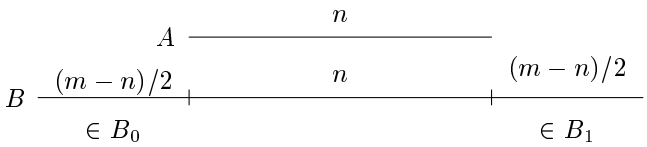


Figura 5.2: Reducerea căutării la $n = |A|$ elemente din B, când $m = |B| > n$ (aici $m - n$ este par).

astfel încât $|A_0| + |B_0| = \lceil (m+n)/2 \rceil$, $|A_1| + |B_1| = \lfloor (m+n)/2 \rfloor$ şi orice element din A_0 sau B_0 este mai mic decât orice element din A_1 sau B_1 În consecinţă, mediana secvenţei $A \perp B$ va fi ultima valoare (cea mai mare) din $A_0 \perp B_0$.

În acest fel se face legătura cu interclasarea. După aflarea partiţionării mediane, se pot găsi, în paralel, $A_0 \perp B_0$ şi $A_1 \perp B_1$, problema de interclasare fiind spartă în două probleme de dimensiuni mai mici şi *egale*.

Vom prezenta două idei care contribuie la găsirea rapidă a partiţionării mediane. Întâi, că se poate lucra cu numai n elemente din B, deci cu două secvenţe de dimensiuni egale. Apoi, că se pot utiliza comparaţii paralele.

Reducerea la secvenţe egale. Dacă ne uităm la figura 5.2, în care secvenţele A şi B sunt reprezentate prin segmente de lungimi proporţionale cu numărul de elemente, observăm că primele $m - \lfloor (m+n)/2 \rfloor$ din B, adică cele din $B_l = \langle b_0, b_1, \ldots, b_{m-\lfloor (m+n)/2 \rfloor - 1} \rangle$ aparţin în mod sigur lui B_0; aceasta rezultă imediat din faptul că în secvenţa B există $\lfloor (m+n)/2 \rfloor$ elemente mai mari decât cele din B_l, adică jumătate dintre elementele din $A \perp B$. În mod analog, este sigur că elementele din $\langle b_{\lceil (m+n)/2 \rceil}, \ldots, b_{m-1} \rangle$ aparţin secvenţei B_1. În acest fel este redusă dimensiunea problemei partiţionării.

Să încorporăm această idee într-o procedură. Vom nota cu la şi ua doi indici în secvenţa A, astfel încât toate elementele la stânga lui la sunt în A_0, iar toate elementele la dreapta lui ua sunt în A_1. Cele între la şi ua sunt încă în A; iniţial $la = 0$ şi $ua = n - 1$; la terminarea partiţionării (dar mai avem până acolo), $la = ua + 1$. Se definesc în mod analog indicii lb şi ub pentru secvenţa B (iniţial $ub = m - 1$). Algoritmul de reducere a dimensiunii este (pentru rigurozitate, uităm aici că am presupus $n < m$ şi considerăm cazul general):

ALGORITM 5.13 (reducerea dimensiunii problemei partiţionării mediane)

procedură *reduce*(la, ua, lb, ub)

1. **dacă** $n > m$ **atunci**
   1. $ua \leftarrow la + \lceil (m+n)/2 \rceil - 1$
   2. $la \leftarrow la + n - \lfloor (m+n)/2 \rfloor$
2. **altfel dacă** $n < m$ **atunci**
   1. $ub \leftarrow lb + \lceil (m+n)/2 \rceil - 1$
   2. $lb \leftarrow lb + m - \lfloor (m+n)/2 \rfloor$

Apelul procedurii se face prin *reduce*$(0, n-1, 0, m-1)$.

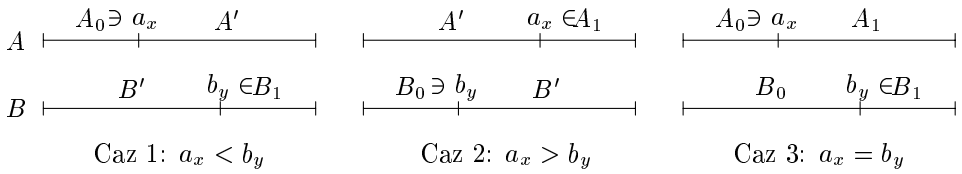


Figura 5.3: Comparaţie între elemente cu indici complementari; concluzii posibile.

Pentru a introduce ideea care permite efectuarea mai multor comparaţii în paralel în căutarea partiţionării mediane, vom putea considera $n = m$.

Reducerea căutării prin utilizarea indicilor complementari. Vom încerca să extragem maximul de informaţie din comparaţii între a_x şi b_y, unde $0 \leq x \leq n-1$ şi $y = n-1-x$; numim y indicele *complementar* al lui x; se observă că a_x şi b_y sunt egal distanţate de începutul secvenţei A, respectiv sfârşitul secvenţei B.

Dacă $a_x < b_y$, atunci putem pune $\langle a_0, \ldots, a_x \rangle$ în A_0 şi $\langle b_y, \ldots, b_{n-1} \rangle$ în B_1; într-adevăr, sunt $n-1-x$ elemente în A mai mari decât a_x şi $n-y = x+1$ elemente în B mai mari decât a_x, în total n elemente din $A \perp B$ mai mari decât a_x, deci mediana este mai mare decât a_x. În mod analog se demonstrează că mediana este mai mică decât b_y. În figura 5.3, cazul 1, este ilustrată această situaţie; cu A' şi B' sunt notate subsecvenţele din A, respectiv B, în care trebuie continuată căutarea medianei.

Dacă $a_x > b_y$, situaţia se inversează: $\langle a_x, \ldots, a_{n-1} \rangle$ aparţine lui A_1 şi $\langle b_0, \ldots, b_y \rangle$ aparţine lui B_0 (vezi figura 5.3, cazul 2).

În fine, dacă $a_x = b_y$, am descoperit partiţionarea mediană şi mediana însăşi, care este a_x.

Reducere paralelă a căutării. Pentru a face mai multe procesoare să participe la reducerea secvenţelor A' şi B', vom face comparaţii în mai multe puncte. Fie $x_0, \ldots, x_{t-1}$ un număr de t indici diferiţi, în ordine crescătoare, şi indicii complementari $y_0, \ldots, y_{t-1}$. Comparaţiile între a_{x_i} şi b_{y_i} pot fi făcute în paralel. Dacă notăm $c_i = 0$ pentru $a_{x_i} < b_{y_i}$ şi $c_i = 1$ altfel, atunci în secvenţa $\langle c_0, \ldots, c_{t-1} \rangle$ există cel mult două elemente vecine diferite. Deci există un singur j pentru care $a_{x_j} < b_{y_j}$ şi $a_{x_{j+1}} \geq b_{y_{j+1}}$. Această afirmaţie este evident adevărată deoarece secvenţa $\langle a_{x_0}, \ldots, a_{x_t - 1} \rangle$ este crescătoare, în timp ce $\langle b_{y_0}, \ldots, b_{y_t - 1} \rangle$ este descrescătoare; ele se pot "intersecta" cel mult o dată.

În concluzie, dacă notăm A' şi B' părţile din A şi B care mai pot conţine mediana după efectuarea celor t comparaţii, se poate ajunge într-unul din următoarele patru cazuri.

1. Dacă există cel puţin un j pentru care $a_{x_j} = b_{y_j}$, atunci:

$$\begin{array}{lll} A_0 = \langle a_0, \ldots, a_{x_j} \rangle & B_0 = \langle b_0, \ldots, b_{y_j - 1} \rangle & A' = \emptyset \\ A_1 = \langle a_{x_j+1}, \ldots, a_{n-1} \rangle & B_1 = \langle b_{y_j}, \ldots, b_{n-1} \rangle & B' = \emptyset \end{array}$$

2. Dacă $a_{x_0} < b_{y_0}$ şi $a_{x_{t-1}} < b_{y_{t-1}}$, atunci ($a_{x_{t-1}}$ şi $b_{y_{t-1}}$ sunt ca în figura 5.3, cazul 1):

$$\begin{array}{lll} A_0 = \langle a_0, \ldots, a_{x_{t-1}} \rangle & B_0 = \emptyset & A' = \langle a_{x_{t-1}+1}, \ldots, a_{n-1} \rangle \\ A_1 = \emptyset & B_1 = \langle b_{y_{t-1}}, \ldots, b_{n-1} \rangle & B' = \langle b_0, \ldots, b_{y_{t-1}-1} \rangle \end{array}$$

3. Dacă $a_{x_0} < b_{y_0}$, şi există un indice j pentru care $a_{x_{j-1}} < b_{y_{j-1}}$, dar $a_{x_j} > b_{y_j}$, atunci:

$$\begin{array}{lll} A_0 = \langle a_0, \ldots, a_{x_{j-1}} \rangle & B_0 = \langle b_0, \ldots, b_{y_j} \rangle & A' = \langle a_{x_{j-1}+1}, \ldots, a_{x_j-1} \rangle \\ A_1 = \langle a_{x_j}, \ldots, a_{n-1} \rangle & B_1 = \langle b_{y_{j-1}}, \ldots, b_{n-1} \rangle & B' = \langle b_{y_j+1}, \ldots, b_{y_{j-1}} \rangle \end{array}$$

4. Dacă $a_{x_0} > b_{y_0}$ (a_{x_0} şi b_{y_0} sunt ca în figura 5.3, cazul 2), atunci:

$$\begin{array}{lll} A_0 = \emptyset & B_0 = \langle b_0, \ldots, b_{y_0} \rangle & A' = \langle a_0, \ldots, a_{x_0-1} \rangle \\ A_1 = \langle a_{x_0}, \ldots, a_{n-1} \rangle & B_1 = \emptyset & B' = \langle b_{y_0+1}, \ldots, b_{n-1} \rangle \end{array}$$

Să mai observăm că partiţionarea mediană a fost găsită în cazul 1 întotdeauna, în cazul 2 dacă $x_{t-1} = n-1$, în cazul 3 dacă $x_{j-1} = x_j - 1$ şi în cazul 4 dacă $x_0 = 0$.

Procedura următoare înglobează toate aceste rezultate. Toate notaţiile se păstrează; vectorul c este folosit pentru memorarea sensului inegalităţilor.

ALGORITM 5.14 (partiţionare mediană, pe CRCW PRAM, id $\equiv k$)

procedură *partiţionare_mediană*(la, ua, lb, ub)

1. **dacă** $k = 0$ **atunci** *reduce*(la, ua, lb, ub)
2. **cât timp** $la \leq ua$
   1. $c_k \leftarrow 1$, $exit \leftarrow 0$, $d \leftarrow -1$
   2. $x_k \leftarrow la + (k+1)\lceil (ua - la + 1)/(p+1) \rceil$ {se aleg indicii x_i echidistanţi}
   3. $y_k \leftarrow la + ub - x_k$ {complementarul lui x_k}
   4. **dacă** $a_{x_k} > b_{y_k}$ **atunci**
      1. **dacă** $k = 0$ {cazul 4}
         1. $ua \leftarrow x_0 - 1$, $lb \leftarrow y_0 + 1$, $exit \leftarrow 1$
   5. **altfel dacă** $a_{x_k} < b_{y_k}$ **atunci**
      1. **dacă** $k = p - 1$ {cazul 2}
         1. $la \leftarrow x_{p-1} + 1$, $ub \leftarrow y_{p-1} - 1$, $exit \leftarrow 1$
      2. **altfel** $c_k \leftarrow 0$ {cazul 3}
   6. **altfel** {cazul 1, $a_{x_k} = b_{y_k}$}
      1. $d \leftarrow k$
   7. **dacă** $exit = 0$ **atunci**
      1. **dacă** $d = k$ **atunci** {din nou cazul 1}
         1. $la \leftarrow x_k + 1$, $ua \leftarrow x_k$, $lb \leftarrow y_k$, $ub \leftarrow y_k - 1$
      2. **dacă** $c_k = 0$ şi $c_{k+1} = 1$ **atunci** {din nou cazul 3}
         1. $la \leftarrow x_k + 1$, $ua \leftarrow x_{k+1} - 1$, $lb \leftarrow y_{k+1} + 1$, $ub \leftarrow y_k - 1$

În final, se obţin $A_0 = \langle a_0, \ldots, a_{ua} \rangle$, $B_0 = \langle b_0, \ldots, b_{ub} \rangle$, $A_1 = \langle a_{la}, \ldots, a_{n-1} \rangle$, $B_1 = \langle b_{lb}, \ldots, b_{n-1} \rangle$.

Pot apărea scrieri simultane în instrucţiunile 2.1 şi 2.6.1; în prima, aceasta se evită foarte uşor lăsând un singur procesor să scrie variabilele *exit* şi d; în 2.6.1, indiferent ce procesor scrie (în modelele ARBITRARY sau PRIORITY CRCW, desigur), algoritmul funcţionează corect (vezi şi problema 5.3.6). În variabila *exit* scrie cel mult un procesor (P_0 sau P_{p-1}). De asemenea, în la, ua, lb, ub, scrie un singur procesor, în general (vezi, totuşi, problema 5.3.8).

Procesoarele lucrează în paralel. Totuşi, unele pot sta ceva mai mult timp într-o iteraţie (în instrucţiuni **dacă**), de unde necesitatea unor sincronizări. De exemplu, înainte de 2.7, sincronizarea este necesară pentru a asigura actualizarea variabilelor *exit*, C şi d în iteraţia curentă. De asemenea este necesară sincronizarea la începerea fiecărei iteraţii a buclei 2, pentru a permite actualizarea variabilelor la, ua, lb, ub.

Algoritmul poate fi uşor adaptat unui EREW PRAM. Se lasă căutarea să continue chiar dacă $a_{x_k} = b_{y_k}$ (de altfel nici nu se mai verifică egalitatea). În acest scop se elimină variabila *exit* şi tot ce urmează după instrucţiunea 2.7 inclusiv. De asemenea, se elimină instrucţiunile 2.6, 2.6.1 şi se înglobează cazul $a_{x_k} = b_{y_k}$ în instrucţiunea 2.5, transformând semnul $<$ în $\leq$.

Să vedem care este timpul de execuţie al acestui algoritm; se observă că în fiecare iteraţie se execută în paralel o singură comparaţie. Cum A şi B sunt împărţite în $p+1$ secvenţe de lungimi egale, iar în iteraţia următoare căutarea este continuată numai într-una din aceste perechi de secvenţe, însemnă că dimensiunea problemei se reduce de $p+1$ ori. Dacă z este numărul total de iteraţii (după care $ua < la$), atunci z este cel mai mic întreg pentru care $n/(p+1)^z < 1$. Adică se obţine $z = T(n,p) = \lfloor \log n / \log(p+1) \rfloor + 1$.

Cum complexitatea secvenţială a acestui algoritm este $T(n,1) = \log n / \log 2$, eficienţa sa este $\varepsilon(n,p) \approx \log(p+1)/p$, deci nu grozavă; algoritmul este bun pentru un număr mic de procesoare; pentru $p = 2$, eficienţa este 0.79, însă pentru $p = 32$ ajunge la 0.16. (Acesta este însă cel mai bun algoritm paralel pentru partiţionare mediană cunoscut deocamdată.)

5.3.4 Interclasare prin multipartiţionare

Să mai prezentăm un algoritm de interclasare, tot pe CREW PRAM şi tot pentru cazul $p \ll n$. Acesta are marele avantaj de a crea partiţii egale, în număr de $t \leq p$, şi pentru A şi pentru B, ceea ce implică şi un timp de execuţie ceva mai scurt (aproximativ acelaşi, indiferent de valorile din B), şi o planificare mai simplă a procesoarelor în comparaţie cu algoritmul 5.12 (Valiant-Kruskal). În acest scop este folosit algoritmul de partiţionare mediană descris în secţiunea anterioară.

Problema multipartiţionării este de a găsi secvenţele A_i, B_i, astfel încât $|A_i| + |B_i| = \lceil (m+n)/t \rceil$ şi $A \perp B = \langle A_0 \perp B_0, \ldots, A_{t-1} \perp B_{t-1} \rangle$; deci, de a partiţiona $A \perp B$ în secvenţe de dimensiuni egale (eventual mai puţin ultima), fără însă a face interclasarea.

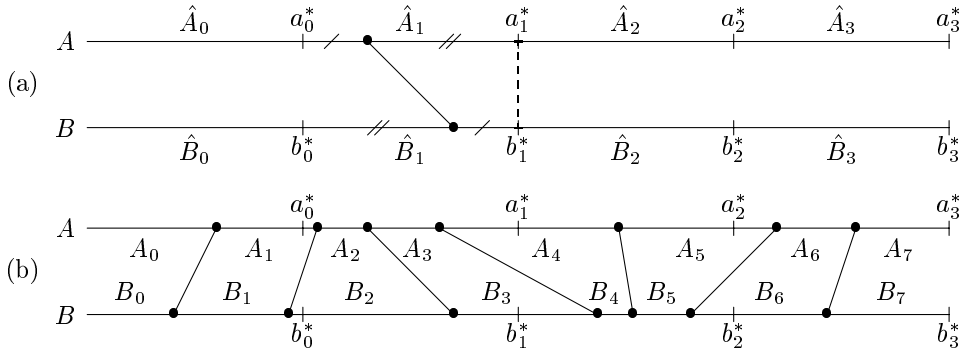


Figura 5.4: (a) linii de separare; exemplu de partiţionare mediană pentru o pereche $\hat{A}_i$, $\hat{B}_j$; (b) exemplu de multipartiţionare în cazul $A^* \perp B^* = \langle a_0^*, b_0^*, a_1^*, b_1^*, b_2^*, a_2^*, a_3^*, b_3^* \rangle$.

Considerăm, pentru început, A şi B compuse din subsecvenţe de dimensiuni egale $s = \lceil (m+n)/t \rceil$ (ultimele fiind mai scurte, eventual):

$$\hat{A}_0 = \langle a_0, \ldots, a_{s-1} \rangle,\ \hat{A}_1 = \langle a_s, \ldots, a_{2s-1} \rangle,\ \ldots$$
$$\hat{B}_0 = \langle b_0, \ldots, b_{s-1} \rangle,\ \hat{B}_1 = \langle b_s, \ldots, b_{2s-1} \rangle,\ \ldots$$

Definim, ca în algoritmul 5.11, $A^* = \langle a_i^* \rangle = \langle a_{is-1} \mid i = 1, \ldots, \lfloor n/s \rfloor \rangle$, secvenţa formată din ultimele elemente ale subsecvenţelor A_i; B^* se defineşte analog. Secvenţa $A^* \perp B^*$ poate fi determinată în timp $\log \log t$, utilizând algoritmul Valiant (sunt cel puţin t procesoare, pentru a interclasa secvenţe de lungime $\lfloor n/s \rfloor + \lfloor m/s \rfloor \approx (m+n)/s \approx t$, deci se poate atinge timpul optim).

Vom face multipartiţionarea prin executarea în paralel a $t-1$ procese de partiţionare mediană pe perechi de subsecvenţe $\hat{A}_i$, $\hat{B}_j$, cu $j = rp(a_i^*)$ sau $i = rp(b_j^*)$ (pentru noţiunea de pereche din dreapta, $rp()$, vezi secţiunea 5.3.2; tot acolo se află modul de calcul al acestor entităţi, care se obţin imediat din algoritmul Valiant).

Să pregătim terenul cu câteva definiţii şi observaţii. În reprezentarea grafică din figura 5.4a, în care secvenţele sunt figurate prin segmente de dreaptă, numim punct de separare un indice care separă o secvenţă în două subsecvenţe (elementul cu indicele respectiv fiind în secvenţa din stânga sau în cea din dreapta), şi linie de separare, o dreaptă care uneşte două puncte de separare a două secvenţe; în figura 5.4a, sunt reprezentate două astfel de linii, una continuă, alta punctată.

OBSERVAŢIA 5.1 *Linia de separare prin a_i^* şi b_j^*, cu $j = rp(a_i^*)$, lasă în stânga ei cel puţin $(i+j+1)s$ elemente din $A \perp B$ (exemplu: linia punctată din figura 5.4a).*

Demonstraţie. Deoarece $a_i^* \leq b_j^*$, vom număra elementele din $A \perp B$ mai mici decât a_i^*. Sunt $(i+1)s$ elemente din A mai mici decât a_i^*. Cum $b_{j-1}^* < a_i^*$, atunci sunt js elemente din B mai mici decât a_i^*; nu putem spune nimic despre elementele din B între b_{j-1}^* şi b_j^*. ■

OBSERVAŢIA 5.2 *Linia de separare mediană a secvenţei $\hat{A}_i \perp \hat{B}_j$, cu $j = rp(a_i^*)$, lasă în stânga sa exact $(i+j+1)s$ elemente din $A \perp B$ (exemplu: linia continuă oblică din figura 5.4).*

Demonstraţie. Imediată, din observaţia 5.1 şi din figură, unde segmentele notate cu acelaşi număr de linii mici oblice au acelaşi număr de elemente (din cele $(i+j+2)s$ elemente din stânga liniei de separare $a_i^* b_j^*$, se extrag s; observaţia 5.1 este de ajutor pentru că arată că linia continuă este în stânga celei punctate). ■

Prin simetrie, cele două observaţii sunt valabile şi pentru perechile b_i^*, a_j^*, cu $j = rp(b_i^*)$. În sfârşit, o ultimă precizare:

OBSERVAŢIA 5.3 *Este adevărată următoarea relaţie:*

$$\{i + rp(a_i^*) \mid i = 0, \ldots, \lfloor n/s \rfloor - 1\} \cup \{i + rp(b_i^*) \mid i = 0, \ldots, \lfloor m/s \rfloor - 1\} = \{0, 1, \ldots, \lfloor n/s \rfloor + \lfloor m/s \rfloor - 1\} \tag{5.3}$$

Demonstraţie. Prin inducţie, vezi problema 5.3.9. ■

Această observaţie arată că sumele dintre indici şi perechile lor din dreapta formează exact şirul primelor $\lfloor n/s \rfloor + \lfloor m/s \rfloor - 1$ numere naturale. Împreună cu celelalte observaţii, aceasta permite reducerea problemei multipartiţionării în t segmente la $t-1$ partiţionări mediane. Figura 5.4b ilustrează acest lucru; pe scurt, observaţiile de mai sus arată că între oricare două linii de separare consecutive se află exact s elemente din $A \perp B$. Algoritmul general de multipartiţionare este următorul:

ALGORITM 5.15 (multipartiţionare prin partiţionare mediană)

1. găseşte $A^* \perp B^*$ cu algoritmul 5.11
2. **în paralel**
   1. **pentru** $i = 0 : \lfloor n/s \rfloor - 1$, **în paralel**
      1. partiţionează median perechea $\hat{A}_i$, $\hat{B}_{rp(a_i^*)}$ utilizând $\lfloor p/(t-1) \rfloor$ procesoare
   2. **pentru** $j = 0 : \lfloor m/s \rfloor - 1$, **în paralel**
      1. partiţionează median perechea $\hat{B}_j$, $\hat{A}_{rp(b_j^*)}$ utilizând $\lfloor p/(t-1) \rfloor$ procesoare

Timpul necesar pentru multipartiţionare este

$$\lfloor \log((n+m)/t) / \log(\lfloor p/(t-1) \rfloor + 1) + O(\log \log p).$$

Interclasarea prin multipartiţionare este acum uşor de făcut. Se utilizează $t = p/2$ în algoritmul 5.15, pentru a obţine $p/2$ perechi disjuncte de subsecvenţe, care se vor interclasa, fiecare de către două procesoare (după un algoritm banal: fiecare procesor interclasează începând de la un capăt; eficienţa este 1).

ALGORITM 5.16 *(Xiong)* (interclasare prin multipartiţionare pe CREW PRAM)

1. apelează algoritmul 5.15 de multipartiţionare cu $t = p/2$
2. **pentru** $i = 0 : p/2 - 1$, **în paralel**
   1. interclasează A_i şi B_i utilizând două procesoare

Au loc citiri simultane deoarece procesoare diferite pot lucra cu aceeaşi subsecvenţă $\hat{A}_i$ sau $\hat{B}_j$ (în medie două procesoare operează pe o subsecvenţă). Complexitatea algoritmului este simplu de calculat:

$$T(n,p) = \frac{m+n}{p} + \frac{1}{\log 3}\log\frac{m+n}{p} + O(\log\log p). \tag{5.4}$$

Primul termen corespunde interclasării secvenţiale (şi e optim), iar următorii multipartiţionării. Comparând cu complexitatea algoritmului Valiant-Kruskal (5.1) se constată diferenţa doar la al doilea termen, ceva mai mic aici.

Marele avantaj al algoritmului rămâne însă faptul că se crează subprobleme de dimensiuni egale, ceea ce implică o încărcare echilibrată a procesoarelor.

5.3.5 Interclasare pe arhitecturi cu memorie distribuită

Titlul acestei secţiuni este foarte promiţător; din păcate, cum veţi vedea, conţinutul nu va fi prea generos.

Începem cu partea mai consistentă şi care ne va fi de mare folos mai departe, anume interclasarea cu două procesoare. S-ar părea că nu e foarte mult de discutat pe tema asta. Algoritmul cel mai simplu a fost deja sugerat şi el este următorul: P_0 trimite secvenţa sa A_0 de lungime n lui P_1 şi recepţionează de la acesta secvenţa A_1 (tot de lungime n); apoi P_0 interclasează A_0 şi A_1, oprindu-se atunci când a obţinut primele n elemente; în acelaşi timp, şi P_1 interclasează A_0 şi A_1, dar începând de la sfârşitul secvenţelor, oprindu-se şi el atunci când a obţinut n elemente; în acest moment secvenţa $A_0 \perp A_1$ este distribuită celor două procesoare. Cum P_0 are acum toate cele n elemente mai mici decât cele ale lui P_1, putem asimila operaţia executată cu $exch(A_0, A_1)$, o generalizare a aceleiaşi operaţii la nivel de element. Presupunând A_i şi A_j ordonate, după executarea $exch(A_i, A_j)$ se obţine $A_i < A_j$, ca în procedura care urmează; ca şi în cazul operaţiei la nivel de element, vom nota $exch(i,j)$ atunci când nu există posibilitatea de confuzie; presupunem că procesoarele i şi j sunt vecine.

ALGORITM 5.17 (interclasare cu două procesoare, pe arhitecturi cu memorie distribuită, id $\equiv k$, secvenţa locală este notată $\tilde{A}$)

procedură $exch(i,j)$

1. **dacă** $i = k$ **atunci**
   1. **în paralel** **send**$(\tilde{A}, j)$, **recv**(B, j)
   2. $\tilde{A} \leftarrow$ primele n elemente din $\tilde{A} \perp B$, prin interclasare secvenţială
2. **dacă** $j = k$ **atunci**
   1. **în paralel** **send**$(\tilde{A}, i)$, **recv**(B, i)
   2. $\tilde{A} \leftarrow$ ultimele n elemente din $\tilde{A} \perp B$, prin interclasare secvenţială (în ordine inversă)

Timpul de execuţie este de $n\alpha$ pentru interclasarea propriu-zisă şi de $(\sigma + n\beta)$ pentru comunicaţie. Numărul de comparaţii este jumătate din cel necesar, în cel mai

rău caz, interclasării secvenţiale, şi mai bine de atât nu se poate. Comunicaţia însă, poate fi îmbunătăţită.

Să presupunem că procesoarele nu îşi comunică de la început secvenţele în întregime, ci o fac în pachete de lungime ν. Ideea care se ascunde aici e de a transmite astfel de pachete până când procesoarele sunt sigure (şi vor fi amândouă în acelaşi timp) că au fiecare suficiente elemente din secvenţa celuilalt pentru a reuşi să obţină partea din rezultat pe care trebuie s-o calculeze. Speranţa este de a ajunge aici înainte de a se comunica întregile secvenţe.

Se începe cu P_0 trimiţând lui P_1 ultimul pachet din A_0 şi primind de la acesta primul pachet din A_1; este normal să se procedeze aşa, din moment ce P_0 începe interclasarea de la început, adică cu elementele mai mici, în timp ce P_1 începe cu elementele mai mari. Să raţionăm din punctul de vedere al lui P_0 (partenerul său va proceda în mod analog); ca în algoritmul 5.17, notăm secvenţa locală cu $\tilde{A}$ ($= \langle \tilde{A}_0, \ldots, \tilde{A}_{n/\nu-1} \rangle$) şi cu B copia locală a lui A_1, conţinând deocamdată numai ν elemente. Să comparăm $\tilde{a}_{n-\nu}$ cu $b_{\nu-1}$, uitându-ne la figura 5.3 şi amintindu-ne că indicii acestor două elemente sunt complementari; dacă $\tilde{a}_{n-\nu} \leq b_{\nu-1}$, atunci este sigur că primele n elemente din $\tilde{A} \perp B$ sunt deţinute de P_0. Dacă nu, atunci se continuă schimbând încă un pachet; se compară acum $\tilde{a}_{n-2\nu}$ şi $b_{2\nu-1}$, ş.a.m.d.

ALGORITM 5.18 (interclasare cu două procesoare, id $\equiv k$, comunicaţie prin pachete)

procedură *exch*1(i, j)
1. $l \leftarrow 1$, $gata \leftarrow 0$, $B \leftarrow \emptyset$
2. **cât timp** $l \leq n/\nu$ şi $gata = 0$ {schimbă un pachet}
   1. **dacă** $i = k$ **atunci**
      1. **în paralel** **send**$(\tilde{A}_{n/\nu-l}, j)$, **recv**$(\tilde{B}, j)$
      2. $B \leftarrow \langle B, \tilde{B} \rangle$ {concatenează în B, la dreapta}
      3. **dacă** $\tilde{a}_{n-l\nu} \leq b_{l\nu-1}$ **atunci** $gata \leftarrow 1$
   2. **dacă** $j = k$ **atunci**
      1. **în paralel** **send**$(\tilde{A}_{l-1}, i)$, **recv**$(\tilde{B}, i)$
      2. $B \leftarrow \langle \tilde{B}, B \rangle$ {concatenează în B, la stânga}
      3. **dacă** $\tilde{a}_{l\nu-1} \geq b_0$ **atunci** $gata \leftarrow 1$
   3. $l \leftarrow l + 1$

3 **dacă** $i = k$ **atunci**
   1. $\tilde{A} \leftarrow$ primele n elemente din $\tilde{A} \perp B$, prin interclasare secvenţială

4. **altfel**
   1. $\tilde{A} \leftarrow$ ultimele n elemente din $\tilde{A} \perp B$, prin interclasare secvenţială (în ordine inversă)

Avantajele şi dezavantajele unei astfel de abordări se pot evidenţia pe cazul simplu $\nu = n/2$. Probabilitatea ca un singur pachet comunicat să fie suficient este $1/2$; într-adevăr: mediana secvenţei $A_0 \perp A_1$ se află în jumătatea din stânga a lui A_1 sau în jumătatea din dreapta a lui A_0, cu aceeaşi probabilitate de a se afla în una din

celelalte jumătăţi. În primul caz e suficient un pachet, în al doilea trebuie comunicate ambele. Atunci, în medie, timpul de comunicaţie al variantei $exch1()$ este: $\frac{1}{2}(\sigma + \frac{1}{2}n\beta) + \frac{1}{2}(2\sigma + n\beta) = \frac{3}{2}\sigma + \frac{3}{4}n\beta$, deci, grosso modo, puţin peste trei sferturi din cel în care se comunicau secvenţele întregi, şi, în concluzie, metoda este benefică. Nu trebuie mizat însă foarte mult pe ea; dacă există multe perechi de procesoare care fac astfel de interclasări, iar apoi urmează, într-un fel sau altul, o sincronizare (se schimbă partenerii de comunicaţie, de pildă), timpul total de execuţie va fi dictat mai degrabă de timpul cel mai mare de interclasare şi nu de timpul mediu.

Pe tema algoritmului 5.18 se poate broda mult. Se poate trimite întâi un pachet mai mare, de lungime $n/2$, de exemplu, apoi altele mai mici. Probabil că problema nu merită un studiu foarte amănunţit; ideea merită pusă în practică, acordând experimental valoarea (valorile) lui ν.

Din nefericire, de îndată ce sunt mai mult de două procesoare, algoritmul de interclasare se complică, în sensul că nici una dintre topologiile curente nu se potriveşte structurii comunicaţiei. Adaptarea multipartiţionării la arhitecturile cu memorie distribuită implică o structură neregulată a comunicaţiei. Totuşi, pentru o posibilă soluţie paralelă, vezi problema 5.4.17.

Probleme

P 5.3.1 Scrieţi (totuşi...) algoritmul secvenţial de interclasare a două secvenţe.

P 5.3.2 Să se demonstreze că în algoritmul 5.11 sunt suficiente $\sqrt{nm}$ procesoare.

P 5.3.3 Demonstraţi că $T(n) = \log\log n$ verifică $T(n) = T(\sqrt{n}) + 1$.

P 5.3.4 Găsiţi încă un argument pentru aproximarea cu $(n+m)/p$ a complexităţii etapei a doua din algoritmul Valiant-Kruskal.

P 5.3.5 Demonstraţi relaţia (5.2).

P 5.3.6 Dacă x şi y sunt indici complementari şi $a_x = b_y$, demonstraţi că valoarea medianei secvenţei interclasate $A \perp B$ este a_x, chiar dacă în fiecare dintre secvenţele A şi B pot exista elemente egale.

P 5.3.7 Scrieţi un algoritm de partiţionare mediană a secvenţelor ordonate A şi B, de lungime egală n, pe un CREW PRAM, folosind $p = n$ procesoare.

P 5.3.8 În algoritmul 5.14, este posibil ca două procesoare să scrie simultan în variabilele la, ua, lb, ub ?

P 5.3.9 Dacă $X = \langle x_0, \ldots, x_{n-1}\rangle$ şi $Y = \langle y_0, \ldots, y_{m-1}\rangle$ sunt două secvenţe ordonate crescător, să se arate că $\{i + rp(x_i) \mid i = 0, \ldots, n-1\} \cup \{j + rp(y_j) \mid j = 0, \ldots, m-1\} = \{0, \ldots, n+m-1\}$. Altfel zis, să se demonstreze (5.3).

P 5.3.10 O generalizare a noţiunii de partiţionare mediană este k-separarea. Problema este definită astfel: dându-se secvenţele ordonate A şi B, să se găsească A_0, A_1, B_0, B_1 astfel

încât $|A_0|+|B_0| = k$, $|A_1|+|B_1| = n+m-k$ şi $A \perp B = \langle A_0 \perp B_0, A_1 \perp B_1 \rangle$. Să se demonstreze că, pentru $n \leq m$, k-separarea este echivalentă cu partiţionarea mediană pentru:

$$\begin{cases} A_{[k]} \perp B_{[k]}, & \text{dacă } 1 \leq k \leq n \\ A \perp (B_{[k]} - B_{[k-n]}), & \text{dacă } n < k \leq m \\ (A - A_{[k-n]}) \perp (B - B_{[k-m]}), & \text{dacă } m < k \leq n+m \end{cases}$$

unde am notat prin $A_{[k]}$ secvenţa primelor k elemente din A.

P 5.3.11 Presupunând comunicaţia realizată half duplex, care este cel mai bun mod de a implementa $exch(a_i, a_j)$, cele două elemente aparţinând unor procesoare vecine ?

P 5.3.12 Pe un CREW PRAM, $exch(a_i, a_j)$ se poate implementa prin instrucţiunea: **dacă** $a_i > a_j$ **atunci** $b \leftarrow a_i$, $a_i \leftarrow a_j$, $a_j \leftarrow b$. Se poate reduce numărul de atribuiri ?

5.4 Sortare

Vom considera numai sortarea *internă*, în care datele se presupun stocate în memoria internă a calculatorului; spre deosebire, în sortarea externă datele se află într-o memorie mult mai lentă, externă calculatorului (disc, etc.).

Complexitatea secvenţială a sortării este $\Omega(n \log n)$ şi există mai multe metode care ating această limită: sortarea rapidă, pe care o vom numi *quicksort* aşa cum e deja consacrat, sortarea prin interclasare (mergesort) sau sortarea de ansamble (heapsort). Spre deosebire de ultimele două, care au aceeaşi complexitate (ca ordin de mărime) în toate cazurile, quicksort se comportă excelent în medie, dar poate ajunge la $O(n^2)$ în cel mai defavorabil caz; totuşi, cu unele mici precauţii explicate în secţiunea dedicată ei, sortarea rapidă este într-adevăr cea mai rapidă şi deci cea mai răspândită, fiind pe deasupra şi relativ uşor de programat. În cele ce urmează vom considera pur şi simplu $n \log n$ complexitatea sortării.

Vom prezenta algoritmi secvenţiali la începutul fiecărei secţiuni următoare, ei fiind în majoritatea cazurilor sursa de inspiraţie, mai mult sau mai puţin directă, pentru algoritmii paraleli.

Atunci când spunem (pentru calculatoarele cu memorie distribuită, de obicei) că un procesor ordonează o secvenţă, vom presupune tacit că o face utilizând cel mai bun algoritm secvenţial cu putinţă; acesta depinde de mulţi factori, dar în primul rând de lungimea secvenţei de sortat.

În cazul arhitecturilor secvenţiale este uşor de spus unde se va găsi secvenţa A după sortare: în exact aceeaşi zonă de memorie ca la început, ordinea crescătoare fiind de la stânga la dreapta. Pentru arhitecturile paralele cu memorie comună situaţia este aceeaşi. În cazul memoriei distribuite, însă, ordinea finală nu mai este atât de evidentă; fiecare procesor deţine o parte din secvenţă şi nu e obligatoriu ca aceste părţi să aibă dimensiuni egale, deşi ar fi de preferat. Vom considera, în mod natural, că A este sortată dacă secvenţa A_i locală procesorului P_i este ordonată crescător, $0 \leq i \leq p-1$, şi dacă $A_i < A_j$, când $i < j$, $0 \leq i, j \leq p-1$.

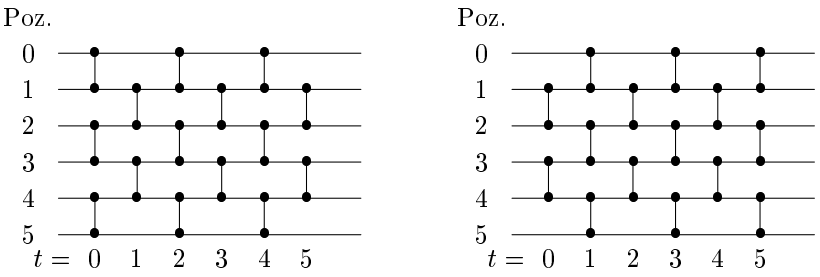


Figura 5.5: Reţele de sortare par-impar.

5.4.1 Sortare par-impar

Există o mulţime de algoritmi secvenţiali de sortare în timp $O(n^2)$ (vezi, de exemplu, problema 5.4.1); ei sunt utili din două motive: sunt extrem de simplu de programat şi sunt cei mai eficienţi pentru număr mic de elemente. În continuare vom prezenta un astfel de algoritm, numit sortare *par-impar* din motive care vor fi imediat evidente.

Reţele de comparatoare. Introducem întâi noţiunea de reţea de comparatoare, cu ajutorul figurii 5.5; liniile orizontale reprezintă elementele din secvenţa de intrare A, iar segmentele verticale unind două orizontale reprezintă comparatoarele, care efectuează operaţii $exch(i, j)$, cu $i < j$ (i şi j sunt numerele de ordine ale orizontalelor); aşadar, circulând de la stânga la dreapta, elementele mai mici au tendinţa de a "urca", cele mai mari de a "coborî". Reţeaua se numeşte *de sortare* dacă la ieşirea ei secvenţa A este ordonată, oricare ar fi fost ordinea iniţială. Puteţi vedea o reţea ca o structură hardware, în care pe orizontale circulă datele, iar comparatoarele au două intrări şi două ieşiri, sau pur şi simplu ca pe o descriere grafică a unui algoritm, cu o bună scoatere în evidenţă a paralelismului.

Pentru a păstra tradiţia, introducem câteva notaţii specifice. Notăm $[i : j]$ comparatorul care execută $exch(i, j)$; dacă α este o reţea, atunci $\alpha[i : j]$ este o reţea formată din concatenarea reţelei α cu comparatorul $[i : j]$ (acesta se adaugă la dreapta lui α); dacă X este o secvenţă de intrare, atunci $X\alpha$ este secvenţa la ieşirea din α, iar $(x\alpha)_i$ elementul din poziţia i al acesteia.

Reţele de sortare par-impar. Figura 5.5 prezintă două astfel de reţele; se observă imediat ca la fiecare pas (sau moment de timp) se execută maximul de comparaţii posibile, alternând paritatea poziţiei elementului mai mic (în urma comparaţiei). Rezultatul se obţine în timp n, cu gradul de paralelism $n/2$. Se utilizează $\binom{n}{2} = n(n-1)/2$ comparatoare; altfel spus, costul algoritmului este $O(n^2)$. Se observă că reţeaua din dreapta, în figura 5.5 este cea din stânga văzută în sens invers; pentru n par, reţelele sunt diferite, pentru n impar identice. Deşi e intuitiv că aceste reţele sortează, demonstraţia nu e tocmai simplă; cititorul grăbit poate sări până la sfârşitul ei.

Proprietăţi ale reţelelor de sortare. O reţea în care toate comparatoarele

sunt de forma $[i : i+1]$ (compară elemente adiacente) se numeşte *simplă*; astfel sunt şi reţelele de sortare par-impar. Multe din proprietăţile demonstrate în continuare sunt valabile *numai* pentru reţele simple.

Într-o secvenţă A, numim *inversare* perechea (a_i, a_j) dacă $a_i > a_j$ şi $i < j$. O secvenţa poate avea cel mult $\binom{n}{2}$ inversări (dacă e ordonată descrescător).

PROPOZIŢIA 5.1 *O reţea simplă de sortare are cel puţin* $\binom{n}{2}$ *comparatoare.*

Demonstraţie. Un comparator $[i : i+1]$ reduce numărul de inversări din secvenţă cu 0 sau 1. Secvenţa sortată nu are inversări; maximul de inversări este $\binom{n}{2}$. ■

Deci, din acest punct de vedere, reţeaua par-impar stă bine: conţine numărul minim de comparatoare.

Notăm cu Z o secvenţă care conţine toate inversările posibile: pentru orice $i < j$, avem $z_i > z_j$; sau, dacă vreţi, putem considera $Z = \langle n, n-1, \ldots, 1 \rangle$.

PROPOZIŢIA 5.2 *(Floyd) Dacă o reţea* α *nu sortează o secvenţă oarecare* X *pentru două poziţii* $i < j$*, adică* $(x\alpha)_i > (x\alpha)_j$*, atunci nu sortează nici* Z *pentru acele poziţii, deci* $(z\alpha)_i > (z\alpha)_j$*.*

Demonstraţie. Prin inducţie după numărul de comparatoare din α. Dacă α are un singur comparator $[r : r+1]$ şi presupunem că $(z\alpha)_i < (z\alpha)_j$, atunci $i = r$, $j = r+1$ şi deci $(x\alpha)_i < (x\alpha)_j$, ceea ce contrazice ipoteza.

Dacă β este o reţea pentru care lema este adevărată, $\alpha = \beta[r : r+1]$ şi $(x\alpha)_i > (x\alpha)_j$, să demonstrăm că $(z\alpha)_i > (z\alpha)_j$.

Dacă $r = i$, atunci $j > r+1$ ($j = r+1$ implică $(x\alpha)_i < (z\alpha)_j$). Cum $(x\alpha)_r < (x\alpha)_{r+1}$ şi $(x\alpha)_r > (x\alpha)_j$, atunci $(x\alpha)_{r+1} > (x\alpha)_j$, deci $(x\beta)_{r+1} > (x\beta)_j$ şi $(x\beta)_r > (x\beta)_j$; din ipoteza de inducţie rezultă că $(z\beta)_{r+1} > (z\beta)_j$ şi $(z\beta)_r > (z\beta)_j$ şi în final $(z\alpha)_i = \min((z\beta)_r, (z\beta)_{r+1}) > (z\alpha)_j$.

Dacă $r = i-1$, atunci $(x\beta)_r$ sau $(x\beta)_{r+1}$ mai mare decât $(x\beta)_j$ (sau amândouă), deci $(z\beta)_r$ sau $(z\beta)_{r+1}$ mai mare decât $(z\beta)_j$ (din ipoteza de inducţie) şi atunci $(z\alpha)_i = \max((z\beta)_r, (z\beta)_{r+1}) > (z\alpha)_j$.

Lăsăm cititorului completarea demonstraţiei pentru $r = j$ şi $r = j-1$. Pentru alţi i, j e banal, deoarece $[r : r+1]$ nu afectează aceste poziţii. ■

PROPOZIŢIA 5.3 *Dacă o reţea simplă sortează* Z*, atunci sortează orice secvenţă.*

Demonstraţie. Consecinţă directă a Prop. 5.2. ■

Pentru a demonstra că reţeaua de sortare par-impar funcţionează corect este suficient să demonstrăm următoarea afirmaţie.

PROPOZIŢIA 5.4 *Reţeaua par-impar sortează* Z*.*

Demonstraţie. Cum reţeaua conţine numărul minim de comparatoare, înseamnă că la fiecare comparaţie urmează interschimbarea, atunci când Z parcurge reţeaua. Să presupunem că se întâmplă aşa. Ne vom referi la reţeaua din figura 5.5, dreapta; pentru cea din stânga, vezi **P**5.4.4.

Considerăm că efectul unei comparaţii la momentul t se manifestă la momentul $t+1$; deci, la $t=0$, z_0 e pe poziţia 0, z_1 pe poziţia 1; la momentul 1, z_1 e pe poziţia 0, z_0 pe poziţia 1. Prima comparaţie are loc la momentul $t=0$, ultima la $t=n-1$; poziţia finală e cea de la momentul $t=n$.

Să încercăm să vedem care este poziţia elementului z_k în momentul t dacă toate comparatoarele fac interschimbări.

$$\text{Pentru } k \text{ impar}: \quad \begin{cases} k-t, & \text{pentru } t \le k \\ t-1-k, & \text{pentru } t \ge k+1 \end{cases}$$

z_k urcă de la $t=1$ la $t=k$, ajungând în poziţia 0; acolo stă un pas; apoi coboară de la $t=k+2$ la $t=n$, ajungând pe poziţia $n-1-k$, adică cea bună.

$$\text{Pentru } k \text{ par}: \quad \begin{cases} k+t, & \text{pentru } t \le n-1-k \\ 2n-1-k-t, & \text{pentru } t \ge n-k \end{cases}$$

z_k coboară de la $t=1$ la $t=n-1-k$, ajungând în ultima poziţie; stă acolo un pas; apoi urcă de la $t=n-k+1$ la $t=n$, ajungând pe poziţia $n-1-k$.

Să demonstrăm prin inducţie după t că această descriere a poziţiei funcţie de timp este corectă. Vom face doar un sfert din demonstraţie, celelalte cazuri fiind asemănătoare ca tehnică.

Pentru t=0, funcţia este evident corectă.

Presupunem că este corectă la momentul t şi vom arăta că, în acest moment, au loc toate interschimbările, ceea ce face funcţia corectă şi la momentul $t+1$. Presupunem k impar şi t par. Atunci:

a) Dacă z_k se află în poziţia $k-t$ (impară, $k<t$), comparaţia se face cu elementul din poziţia $k-t-1$, fie el z_j. Pentru ca interschimbarea să aibă loc este necesar ca $j<k$. Sunt posibile următoarele situaţii:

Pentru j impar: $k-t-1=j-t \Rightarrow j=k-1$, imposibil ($j$ ar fi par)
$k-t-1=t-1-j \Rightarrow j=2t-k<k$
Pentru j par: $k-t-1=j+t \Rightarrow j=k-2t-1<k$
$k-t-1=2n-1-j-t \Rightarrow j=2n-k$, imposibil

b) Dacă z_k se află pe poziţia $t-1-k$ (pară, $t \ge k+1$), comparaţia se face cu elementul din poziţia $t-k$, fie el z_j. Interschimbarea are loc dacă $j>k$.

Pentru j impar: $t-k=j-t \Rightarrow j=2t-k>k$
$t-k=t-1-j \Rightarrow j=k-1$, imposibil
Pentru j par: $t-k=j+t \Rightarrow j=-k$, imposibil
$t-k=2n-1-j-t \Rightarrow j=2n-2t+k-1>k$

Mai trebuie verificat că, din cele patru variante bune, exact una are loc la fiecare moment de timp (ceea ce e simplu) şi că, la momente de timp diferite, nu se obţin

soluţii j identice (ceea ce iarăşi e simplu, dar trebuie coroborat cu cazul t impar). Din nou lăsăm cititorului aceste detalii[5]. ■

Aici se termină demostraţia corectitudinii reţelelor de sortare par-impar. Extensia acestui algoritm la arhitecturi MIMD cu memorie partajată este lăsată ca exerciţiu (vezi **P5.4.2**).

Sortare par-impar pe MIMD cu memorie distribuită

Ne ocupăm direct de cazul $p \ll n$. Fiecare dintre cele p procesoare are în memoria locală $m = n/p$ elemente din A.

Aplicarea principiului lui Brent. Am putea aplica ideea din figura 5.5 fără nici o modificare; închipuiţi-vă că $p = 2$ şi că tragem o linie orizontală care separă trei poziţii sus pentru procesorul P_0 şi trei poziţii jos pentru procesorul P_1; fiecare procesor execută comparaţiile şi interschimbările care îi sunt locale, pentru cele în care un element e la P_0, iar celălalt la P_1, efectuându-se comunicaţie (în cadrul unui $exch()$); se comunică de $n/2$ ori, de fiecare dată numai câte un element (deci vor apărea foarte mulţi σ în timpul de comunicaţie); în plus, timpul de execuţie rămâne $O(n^2)$. Nu am făcut altceva decât să aplicăm principiul lui Brent.

Sortare par-impar cu subsecvenţe. O idee mult mai bună e aceea de a aplica sortarea par-impar la nivel de subsecvenţe; ca imagine grafică, în figura 5.5, pe liniile orizontale nu ar mai circula elemente ci subsecvenţe din A. Fiecare procesor se ocupă de o astfel de "linie". Un "comparator" va trebui acum, dintre cele $2m$ elemente de pe cele două linii, să le aleagă pe cele mai mici m şi să le dirijeze pe prima linie, şi pe cele mai mari m, pe cealaltă linie. Dacă subsecvenţele sunt ordonate, atunci aceasta este, într-o arhitectură cu memorie distribuită, chiar o operaţie $exch(A_i, A_{i+1})$, aşa cum a fost definită în secţiunea 5.3.5. Nu este greu de văzut că, dacă un algoritm sortează utilizând $exch()$ la nivel de element, el va sorta şi cu $exch()$ la nivel de subsecvenţă, cu singura condiţie ca subsecvenţele să fi fost iniţial sortate; implementarea $exch()$ din algoritmul 5.17 ne asigură că subsecvenţele rămân permanent sortate, iar faptul că algoritmul sortează ne spune că, la sfârşitul execuţiei sale, subsecvenţele sunt în ordine crescătoare; deci, întreaga secvenţă este sortată.

Deoarece în sortarea par-impar comparaţiile sunt adiacente, topologia suficientă în cazul implementării pe o arhitectură cu memorie distribuită este inelul. Scrierea algoritmului necesită doar un pic de atenţie la parităţi şi la cazurile când procesorul este la una din extreme.

ALGORITM 5.19 (sortare par-impar pe inel, $p \ll n$, id $\equiv k$)

1. sortează crescător secvenţa locală (cu quicksort)
2. **pentru** $i = 0 : p - 1$
    1. **dacă** i este par **atunci**
        1. **dacă** k este par şi $k \neq p - 1$ **atunci** $exch(k, k + 1)$

[5] Important e să fie bine înţeles drumul unui element prin reţea.

2. **dacă** k este impar **atunci** $exch(k-1,k)$
2. **altfel** {i este impar}
   1. **dacă** k este par şi $k \neq 0$ **atunci** $exch(k,k-1)$
   2. **dacă** k este impar şi $k \neq p-1$ **atunci** $exch(k,k+1)$

În acest caz, numărul de comparaţii este (primul termen pentru sortarea locală, al doilea pentru cele p interclasări):

$$T_a(n,p) = \frac{n}{p}\log\frac{n}{p} + p\frac{n}{p} = \frac{n}{p}\log\frac{n}{p} + n.$$

Timpul de comunicaţie este $T_c(n,p) = p(\sigma + (n/p)\beta) = O(n)$. Aceşti timpi nu mai sunt atât de răi ca aceia rezultaţi din aplicarea principiului lui Brent; cauza este că metoda de sortare locală a fost cea mai bună posibil, şi nu tot cea par-impar. Astfel, eficienţa asimptotică a algoritmului este totuşi 1, deşi algoritmul secvenţial de origine este în $O(n^2)$ operaţii.

Generalizare. Nu am fi insistat atât de mult dacă situaţia nu ar fi tipică; e prima dată când ne întâlnim cu aşa ceva, deci merită să generalizăm. Să zicem că rezolvarea unei anume probleme, folosind un algoritm secvenţial rapid $\mathcal{A}_1$, necesită mult mai puţin timp decât dacă se foloseşte un alt algoritm secvenţial $\mathcal{A}_2$. Totuşi, $\mathcal{A}_2$ se paralelizează foarte uşor; în plus are proprietatea că, într-o arhitectură multiprocesor, înainte sau/şi după o fază de cooperare între procesoare, reduce dimensiunea problemei de la n, la n/p, în aşa fel încât fiecare procesor se vede pus în situaţia de a rezolva secvenţial problema sa. Este clar că el o va rezolva folosind cel mai bun algoritm secvenţial cu putinţă, adică $\mathcal{A}_1$. Dacă n/p este suficient de mare şi $\mathcal{A}_2$ nu consumă prea mult timp pentru a reduce dimensiunea problemei, atunci algoritmul paralel inspirat de $\mathcal{A}_2$ are o complexitate de acelaşi ordin de mărime ca o ipotetică paralelizare ideală a algoritmului $\mathcal{A}_1$, pentru că efortul cel mai mare este rezolvarea *locală* a problemei ! Aparent am făcut dintr-un slab algoritm secvenţial, un bun algoritm paralel. În fapt am folosit fiecare algoritm acolo unde era mai bun; din $\mathcal{A}_2$ am luat paralelismul, din $\mathcal{A}_1$ rapiditatea secvenţială. Totuşi, prin tradiţie şi, de ce să nu recunoaştem, pe merit, numele algoritmului paralel este dat de $\mathcal{A}_2$.

În cazul nostru, $\mathcal{A}_1$ este quicksort şi are o complexitate de $O(n\log n)$; $\mathcal{A}_2$ este sortarea par-impar, are complexitate $O(n^2)$, dar combină p sortări de dimensiune n/p în timp $O(n)$ pentru a face o sortare de dimensiune n. Hibridul lor are o complexitate paralelă de $O(n\log n)$.

Nu trebuie însă să rămâneţi cu ideea că quicksort nu se pretează paralelizării. Va fi demostrat imediat contrariul. Totuşi, algoritmul 5.19 se distinge prin simplitate, chiar dacă eficienţa lui este ceva mai slabă, cum se va dovedi după studierea altor metode de sortare paralelă.

Reţele sistolice. În final, o ultimă observaţie privitoare la reţelele de sortare în general, ca structuri hardware. Putem să ne închipuim că există un ceas global, la fiecare tact, comparatoarele citind datele de intrare, efectuând interschimbarea,

dacă e cazul, şi scoţând datele pe liniile de ieşire, pentru a fi citite de următoarele comparatoare, la tactul următor; există o mică dificultate în realizarea sincroniei, anume că unele date nu sunt utilizate la tactul următor, ci mai târziu (în reţelele par-impar, aşa ceva se întâmplă doar pentru p impar); se pot însă introduce module de întârziere cu un tact astfel încât, pe fiecare "linie" orizontală (a unui element din secvenţă) dintr-o reţea de sortare, la fiecare moment de timp (deci pe fiecare verticală), să se afle fie un comparator, fie un modul de întârziere. În acest datele vor circula "aliniate" pe verticală (frontul de undă va fi vertical).

Avantajul acestui mod de a privi lucrurile este că se pot sorta mai multe secvenţe în regim pipeline. Dacă se introduce la fiecare tact câte o secvenţă la intrarea în reţea, nu va exista nici un conflict pe parcursul străbaterii reţelei, decalajul de un tact păstrându-se până la ieşire. Deci, dacă sunt suficiente date, o reţea poate realiza o sortare la fiecare tact, eficienţa fiind foarte mare. Astfel se poate realiza o funcţionare numită *sistolică*, deoarece datele înaintează prin reţea ca fluxul de sânge prin vase, câte puţin la fiecare bătaie a inimii—ceasul global al reţelei. (Termenul de sistolic este mult ulterior reţelelor de sortare; noutatea nu constă însă decât în ideea funcţionării sincrone, în ritmul impus de un ceas global.)

5.4.2 Quicksort paralel

Algoritmul secvenţial. Ideea sortării rapide e simplă şi ingenioasă: se împarte secvenţa A în două părţi; într-una elementele sunt mai mici decât o valoare pivot v, în celalaltă, mai mari; sortându-se aceste două secvenţe, A va rezulta şi ea sortată. Forma generală a algoritmului este cea care urmează.

ALGORITM 5.20 (quicksort, forma generală secvenţială)

procedură *quicksort*(s, d)

1. **cât timp** $s < d$ {secvenţa este nevidă}
   1. *alege_pivot*(s, d)
   2. *partiţionare*(s, d, i_v) {formează cele două subsecvenţe}
   3. *quicksort*$(s, i_v - 1)$ {şi le sortează}
   4. *quicksort*$(i_v + 1, d)$

Variabilele s şi d sunt indicii elementului cel mai din stânga, în secvenţă, respectiv al celui mai din dreapta. Despre alegerea pivotului nu spunem nimic nou faţă de ceea ce am discutat în secţiunea 5.2; reamintim că pivotul se va găsi în poziţia s, adică $v \equiv a_s$. De asemenea, ştim acum cum se realizează secvenţial partiţionarea, din **P5.2.4**, şi chiar paralel, din algoritmul 5.8 (în care ocupă cea mai mare parte); i_v este poziţia pivotului în secvenţa A. După realizarea partiţionării, sortarea subsecvenţelor rezultate poate decurge independent. Apelul se face cu *quicksort*$(0, n-1)$.

Algoritm paralel simplu. O paralelizare naivă a acestui algoritm ar fi următoarea: un procesor partiţionează secvenţa iniţială A; apoi i se mai asociază un procesor, pentru a partiţiona cele două secvenţe rezultate; se mai adaugă două procesoare, fiind

acum patru secvenţe, etc., până când se obţin p secvenţe şi atunci fiecare procesor sortează secvenţial (cu quicksort) o secvenţă. Cum pentru partiţionarea de către un procesor a unei secvenţe de lungime n sunt necesare n comparaţii, şi presupunând că se obţin secvenţe egale la fiecare partiţionare, numărul de comparaţii necesar va fi

$$n + n/2 + n/4 + \ldots + 2n/p + n/p \log(n/p) \approx n/p \log n + 2n$$

ceea ce nu e convenabil; gândiţi-vă numai la cazul $p = \log n$, care e uşor de întâlnit în practică, şi pentru care eficienţa este $1/3$. De fapt va fi şi mai rău, pentru că secvenţele pot fi inegale, şi atunci unele procesoare vor lucra mai mult.

Quicksort pe PRAM

Quicksort bazat pe partiţionare paralelă. Defectul ideii de mai sus este că nu se efectuează partiţionarea în paralel, de către mai multe procesoare. Pe o arhitectură cu memorie comună, o structura generală a unui algoritm quicksort paralel care să înlăture acest neajuns este foarte asemănătoare cu cea a algoritmului secvenţial.

ALGORITM 5.21 (quicksort, forma generală paralelă pe un PRAM)

procedură *pquicksort*(s, d, q)

1. **dacă** $q = 1$ **atunci**
   1. sortează secvenţial folosind algoritmul quicksort 5.20
2. **altfel**
   1. **cât timp** $s < d$ {secvenţa este nevidă}
      1. *alege_pivot*(s, d, q)
      2. *partiţionare*(s, d, i_v, q) {formează cele două subsecvenţe}
      3. $q' \leftarrow \lceil q(i_v - s)/(d - s) \rceil$ {număr de proc. pentru secvenţa din stânga}
      4. *pquicksort*$(s, i_v - 1, q')$ {sortează în stânga cu q' procesoare}
      5. *pquicksort*$(i_v + 1, d, q - q')$ {sortează în dreapta cu $q - q'$ procesoare}

Faţă de forma secvenţială a mai fost adăugat un parametru, numărul de procesoare q; iniţial, avem $q = p$; apoi, după crearea celor două subsecvenţe, procesoarele se împart şi ele în două grupe, de mărime proporţională cu lungimea secvenţelor; de aici mai rezultă un avantaj al acestei abordări, anume că se echilibrează mai bine eforturile procesoarelor, cel puţin câtă vreme q nu devine foarte mic.

Partiţionare paralelă. Întrucât alegerea pivotului e o operaţie care durează puţin şi poate fi făcută în multe feluri, trecem direct la punctul cel mai important: partiţionarea; de fapt implementarea sortării rapide în paralel se reduce la un bun algoritm paralel de partiţionare, o dată adoptată forma 5.21. Cum am menţionat, algoritmul 5.8 conţine în mare măsură ideile de care avem nevoie; acolo se extrăgea doar una dintre cele două secvenţe rezultate; aici avem nevoie de amândouă. Pe scheletul acelui algoritm vom obţine (procesoarele participante sunt $P_0, \ldots, P_{q-1}$, după o eventuală renumerotare; secvenţa de care se ocupă un procesor este notată $\tilde{A}$; pivotul este $v \equiv a_0$):

ALGORITM 5.22 (algoritm de partiţionare pe CREW PRAM, $p \ll n$, pentru procesorul P_k aparţinând unui grup de q procesoare)

procedură *partiţionare*(s, d, i_v, q)

1. $m \leftarrow \lceil (d-s)/q \rceil$
2. $\tilde{A}$ se află între poziţiile $l = s + km$ şi $u = \min[s + (k+1)m - 1, d]$ din A
3. aplică algoritmul din **P5.2.4** pt. partiţionarea $\tilde{A} = \langle \tilde{A}_0, \tilde{A}_1 \rangle$, $\tilde{A}_0 < v \leq \tilde{A}_1$

3'. (în plus, P_0 pune v pe prima poziţie în $\tilde{A}_1$)

4. $c_k \leftarrow |\tilde{A}_0|$, numărul de elemente din $\tilde{A}$ mai mici decât v
5. $f_k \leftarrow |\tilde{A}_1|$, numărul de elemente din $\tilde{A}$ mai mari decât v
6. $c_{k+1} \leftarrow \sum_{i=0}^{k} c_i$ şi $f_{k+1} \leftarrow \sum_{i=0}^{k} f_i$, cu algoritmul 4.8 (suma prefixelor)
7. copiază $\tilde{A}_0$ în B, în poziţiile de la $s + c_k$ la dreapta ($c_0 = 0$)
8. copiază $\tilde{A}_1$ în B, în poziţiile de la $s + c_q + f_k$ la dreapta ($f_0 = 0$)
9. copiază în $\tilde{A}$ elementele corespunzătoare din B (de la l la u)
10. **dacă** $k = 0$ **atunci** $i_v \leftarrow s + c_q$

În instrucţiunea 2, procesorul calculează indicii care delimitează zona de care se ocupă. Partiţionarea ei şi calculul poziţiilor în B ale subsecvenţelor locale $\tilde{A}_0$ şi $\tilde{A}_1$ se face ca în algoritmul 5.8. La copiere, intervine o diferenţă; în 8, secvenţele de tip $\tilde{A}_1$ se copiază într-o zonă din B care începe cu poziţia $s + c_q$; în această poziţie va fi chiar pivotul (în 3', P_0 l-a pus pe prima poziţie în $\tilde{A}_1$, iar acum se copiază $\tilde{A}_1$ începând de la $s + c_q$); cum sunt $c_q = \sum_{i=0}^{q-1} c_k$ elemente mai mici decât v, aceasta este poziţia corectă. În 9, fiecare procesor recopiază aproximativ m elemente din B în A, dar ele nu sunt aceleaşi cu cele deţinute iniţial. Se observă că se poate folosi porţiunea din B cuprinsă între indicii s şi d. Timpul de calcul este bine repartizat între procesoare, deci se poate estima la $(d-s)/q + O(\log q)$ primul termen fiind pentru partiţionarea secvenţială, al doilea pentru suma prefixelor.

Să considerăm acum algoritmul 5.21 în care se utilizează partiţionarea din procedura 5.22. Repartizarea procesoarelor în funcţie de lungimea subsecvenţelor face ca termenul $O(\log q)$ să fie întotdeauna neglijabil faţă de $(d-s)/q$ (pentru $p \ll n$); astfel, timpul de execuţie al algoritmului va fi $O((n/p)\log n)$, de unde o eficienţă asimptotic egală cu 1; al doilea termen ca importanţă în timpul de execuţie este cel mult $O(\log p \log n)$, dar mai mic decât la algoritmul 5.8 (acolo toate cele p procesoare lucrau pe secvenţe din ce în ce mai mici, aici numărul de procesoare scade o dată cu dimensiunea secvenţei). Memoria suplimentară folosită este de n locaţii, adică o întreagă copie a lui A, ceea ce poate însemna destul de mult.

Reducerea memoriei suplimentare. O metodă de a elimina parţial acest inconvenient va fi sugerată în continuare, pentru că este instructivă; ideea apare în [6]. Să presupunem că fiecare procesor nu se ocupă de o secvenţă contiguă, ca în algoritmul 5.22 pasul 2, ci elementele sunt împărţite ciclic procesoarelor, ca în exemplul din tabelul 5.6. În primul grup de linii marcate cu numele procesoarelor este arătată repartizarea elementelor pe procesoare; în al doilea, ordinea locală a elementelor după ce fiecare procesor aplică o partiţionare secvenţială ca în **P5.2.4** (pivotul este 13);

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a_k	13	1	23	25	18	9	7	16	11	6	10	0	15	20	17	5
P_0	13				18				11				15			
P_1		1				9				6				20		
P_2			23				7				10				17	
P_3				25				16				0				5
P_0	11				13				18				15			
P_1		1				9				6				20		
P_2			10				7				23				17	
P_3				5				0				16				25
a_k	11	1	10	**5**	13	9	7	0	18	6	**23**	16	15	20	17	25

Figura 5.6: Reducerea memoriei suplimentare prin repartizarea ciclică a secvenţei A.

ultima linie a tabelului arată ordinea indusă pe A de această partiţionare locală. Se observă că există trei zone în A: una în care elementele sunt mai mici decât pivotul, cea de la valoarea **5** în stânga; o a doua în care elementele sunt mai mari decât pivotul, la dreapta de **23**; între ele, o zonă cu elemente amestecate. Doar pentru această zonă trebuie acum făcută o reordonare, într-un mod oarecum asemănător cu partiţionarea din algoritmul 5.22 (partea de după (2')); nu intrăm în amănunte tehnice şi ne mulţumim de data aceasta numai cu ideea că modul de repartizare a elementelor pe procesoare poate aduce neaşteptate avantaje.

Quicksort pe hipercub

Să vedem acum modul de implementare a sortării rapide pe un hipercub, în care fiecare procesor deţine $m = n/p$ elemente din A ($p \ll n$). Din capul locului se vede că algoritmul 5.21 nu poate fi folosit în litera lui datorită alocării unui număr diferit de procesoare pentru sortarea celor două secvenţe rezultate după partiţionare; pentru o arhitectură cu memorie distribuită aceasta implică o regrupare a elementelor care necesită multă comunicaţie; mai grav, comunicaţia într-un grup de procesoare care se ocupă de aceeaşi secvenţă trebuie să fie regulată; e greu de imaginat, de exemplu, cum vor coopera 7 procesoare dintr-un hipercub cu $p = 16$.

Va trebui deci să ne mulţumim cu o înjumătăţire a procesoarelor la fiecare partiţionare; 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 secvenţa pe care o deţine. O mai slabă echilibrare a efortului va fi compensată de facilitatea comunicaţiei; am ales totuşi răul cel mai mic. Pentru ca nici acest rău să nu fie semnificativ, e necesară ceva mai multă grijă în alegerea pivotului. Este evident care ar fi cea mai bună alegere: chiar mediana secvenţei A; astfel, partiţionarea s-ar face chiar la jumătate; din păcate, calcul exact al medianei e destul de costisitor, deci ne vom mulţumi cu o aproximare a ei.

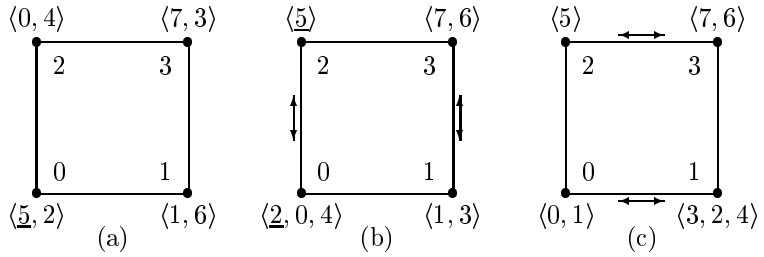


Figura 5.7: Exemplu de desfăşurare a sortării rapide pe un hipercub cu patru procesoare: (a) situaţia iniţială; (b) după comunicaţia pe direcţia 1; (c) după comunicaţia pe direcţia 0.

Pe un hipercub cu $p = 2^d$, acest mod de operare se poate pune uşor în practică. Să spunem că P_0 alege pivotul v dintre elementele locale şi îl difuzează tuturor. Fiecare procesor partiţionează secvenţa proprie: $\tilde{A} = \langle \tilde{A}_0, \tilde{A}_1 \rangle$ astfel încât $\tilde{A}_0 < v \leq \tilde{A}_1$. Până aici totul e banal. Acum procesoarele vecine pe direcţia $d-1$ (care au diferit doar primul bit al reprezentării binare a adresei; pentru P_k acest bit este k_{d-1}) schimbă între ele câte o secvenţă, astfel încât secvenţele mai "mici" $\tilde{A}_0$ să ajungă în jumătatea inferioară a hipercubului (unde $k_{d-1} = 0$), iar cele mai "mari" $\tilde{A}_1$ în jumătatea superioară (unde $k_{d-1} = 1$); astfel procesoarele P_i, cu $0 \leq i \leq p/2 - 1$, au acum 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 ($\mathcal{H}_d^{d-1}(0)$, jumătatea inferioară şi $\mathcal{H}_d^{d-1}(2^{d-1})$, jumătatea superioară), în paralel; se comunică pe direcţia $d-2$, de această dată. Se procedează similar mai departe, la fiecare iteraţie comunicând pe direcţia imediat inferioară, până la direcţia 0; de fiecare dată, numărul grupurilor de procesoare care cooperează se dublează, iar numărul procesoarelor dintr-un grup se înjumătăţeşte. În final, sunt p grupuri formate dintr-un singur procesor; atunci secvenţa A este sortată la nivel mare (P_i are elementele mai mici decât cele ale lui P_j, dacă $i < j$) şi fiecare procesor va avea doar de sortat secvenţa locală.

O ilustrare a algoritmului este prezentată în figura 5.7, pentru $p = 4$ şi $n = 8$; elementele pivot sunt cele subliniate (primele din secvenţele locale); ultima fază, de sortare locală, nu a mai fost reprezentată.

Această primă implementare a algoritmului quicksort are forma următoare.

ALGORITM 5.23 (quicksort pe hipercub, $p = 2^d \ll n$, id $\equiv k$)

1. **pentru** $l = d-1 : -1 : 0$
   1. **dacă** ultimii $l+1$ biţi din k sunt egali cu 0 **atunci**
      1. alege pivotul v dintre elementele locale
      2. **difuzează** v în $\mathcal{H}_d^{l+1}(k)$
   2. aplică algoritmul din **P5.2.4** pentru partiţionarea $\tilde{A} = \langle \tilde{A}_0, \tilde{A}_1 \rangle$, $\tilde{A}_0 < v \leq \tilde{A}_1$
   3. **dacă** $k_l = 0$ **atunci** $\{P_k$ e în subhipercubul "inferior"$\}$

1. **în paralel** **send**$(\tilde{A}_1, l)$, **recv**(X, l)
2. $\tilde{A} \leftarrow \langle \tilde{A}_0, X \rangle$ {concatenează secvenţa primită cu cea rămasă}

4. **altfel** {$k_l = 1$, subhipercubul "superior"}
   1. **în paralel** **send**$(\tilde{A}_0, l)$, **recv**(X, l)
   2. $\tilde{A} \leftarrow \langle \tilde{A}_1, X \rangle$

2. sortează secvenţa proprie $\tilde{A}$

Timpul de execuţie poate fi estimat astfel, în ipoteza unei încărcări echilibrate (fiecare procesor are în permanenţă aproximativ n/p elemente):

$$T(n,p) = \frac{n}{p}\log\frac{n}{p}\alpha + \sum_{i=0}^{d-1}\left[\frac{n}{p}\alpha + (\sigma + \frac{n}{2p}\beta)\right] = \frac{n}{p}(\log n)\alpha + (\sigma + \frac{n}{2p}\beta)\log p. \quad (5.5)$$

Primul termen este pentru sortarea locală, iar în sumă, n/p este numărul de comparaţii necesare partiţionării, iar celălalt termen corespunde comunicaţiei; am neglijat timpul pentru difuzări, acestea făcându-se în paralel pe hipercuburi din ce în ce mai mici. Rezultatul este promiţător, comparaţiile fiind perfect distribuite procesoarelor, deci existând numai overhead de comunicaţie. În practică lucrurile nu stau chiar aşa; dacă alegerea pivotului nu este făcută cu atenţie, nu numai că încărcarea poate deveni foarte inegală, dar algoritmul poate nici să nu funcţioneze. De exemplu, ca un caz extrem, dacă P_0 alege ca pivot cel mai mic element al său, care, întâmplător, este mai mic şi decât orice element din $P_{2^{d-1}}$, după prima iteraţie P_0 rămâne fără nici un element, deci nu mai poate propune un pivot (ar putea propune o valoare aleatoare, dar asta nu are decât o logică: să nu se blocheze execuţia; nici vorbă de performanţă); vezi şi problema 5.4.6.

Hyperquicksort. Un algoritm asemănător, dar cu o mult mai stabilă comportare, a fost propus de Wagar şi numit hiperquicksort [26]. Să presupunem că fiecare procesor începe prin a sorta secvenţa proprie şi să vedem ce avantaje decurg de aici. Alegerea pivotului este acum imediată: P_0 va alege elementul său median; este vorba deci de mediana unui eşantion de n/p elemente, care este o bună aproximare a medianei secvenţei A. P_0 face difuzarea pivotului. Partiţionarea este şi ea simplă; trebuie găsită poziţia pivotului în secvenţa locală, care este ordonată; aceasta se face prin căutare binară, deci rapid. Se comunică, exact ca în algoritmul precedent, secvenţele "mici" în jumătatea inferioară a hipercubului şi secvenţele "mari" în jumătatea superioară. Fiecare procesor va avea acum două secvenţe ordonate, pe care le interclasează, pentru a avea din nou o secvenţă ordonată. După care se continuă în cele două subhipercuburi, apoi în patru, etc., structura comunicaţiei fiind aceeaşi ca la algoritmul dinainte. Toate aceste consideraţii ne conduc la:

ALGORITM 5.24 *(Wagar)* (hiperquicksort, pe hipercub, $p = 2^d \ll n$, id $\equiv k$)

1. sortează secvenţa locală $\tilde{A}$
2. **pentru** $l = d-1 : -1 : 0$

1. **dacă** ultimii $l+1$ biţi din k sunt egali cu 0 **atunci**
   1. alege mediana secvenţei locale $\tilde{A}$ ca pivot v
   2. **difuzează** v în $\mathcal{H}_d^{l+1}(k)$
2. partiţionează $\tilde{A} = \langle \tilde{A}_0, \tilde{A}_1 \rangle$, $\tilde{A}_0 < v \leq \tilde{A}_1$, prin căutare binară
3. **dacă** $k_l = 0$ **atunci** {P_k e în subhipercubul "inferior"}
   1. **în paralel** **send**$(\tilde{A}_1, l)$, **recv**(X, l)
   2. $\tilde{A} \leftarrow \tilde{A}_0 \perp X$ {interclasează secvenţa primită cu cea rămasă}
4. **altfel** {$k_l = 1$, subhipercubul "superior"}
   1. **în paralel** **send**$(\tilde{A}_0, l)$, **recv**(X, l)
   2. $\tilde{A} \leftarrow \tilde{A}_1 \perp X$

Făcând din nou ipoteza unei încărcări echilibrate (mult mai aproape de realitate, acum), timpul de execuţie este

$$T(n,p) = \frac{n}{p}\log\frac{n}{p}\alpha + \sum_{i=0}^{d-1}\left[(\log\frac{n}{p} + \frac{n}{p})\alpha + (\sigma + \frac{n}{2p}\beta)\right] \approx \frac{n}{p}(\log n)\alpha + (\sigma + \frac{n}{2p}\beta)\log p. \tag{5.6}$$

O comparaţie rapidă cu relaţia (5.5) ne dezvăluie un singur termen în plus, în sumă, cel corespunzător căutării binare, de $\log(n/p)$, pe care-l putem şi neglija, de altfel. În rest, sortarea locală este făcută tot o singură dată, comunicaţia este aceeaşi, şi, coincidenţă plăcută, partiţionarea unei secvenţe neordonate (de lungime n/p) din algoritmul 5.23 are aceeaşi complexitate ca interclasarea a două secvenţe (de lungime aproximativ $n/(2p)$) în hiperquicksort. Astfel algoritmul hiperquicksort are aceeaşi complexitate, în cel mai bun caz, ca şi algoritmul 5.23, dar o va atinge cu probabilitate mult mai mare decât acesta din urmă. Este şi motivul pentru care a fost preferat; de asemenea, el este şi relativ simplu.

5.4.3 Sortare echilibrată

Algoritmii de sortare pentru arhitecturi cu memorie distribuită inspiraţi de quicksort au, după cum am am văzut, un dezavantaj: procesoarele nu au acelaşi număr de operaţii de efectuat, de unde existenţa unui overhead de încărcare inegală.

Vom prezenta în această secţiune un algoritm, preluat din [1] care înlătură acest dezavantaj, desigur, cu preţul unor operaţii suplimentare. Notăm cu A secvenţa de sortat, de lungime n şi cu A_k secvenţa locală procesorului P_k, de lungime $m = n/p$. Primul lucru pe care-l face fiecare procesor este să-şi ordoneze secvenţa locală.

Ideea: multiselecţie şi schimb complet. Să încercăm să facem astfel încât procesorul P_0 să aibă cele mai mici m elemente, P_1 pe următoarele m, etc. După aceea va fi suficientă o nouă sortare locală. Pentru aceasta este necesară parcurgerea următoarelor etape:

1. Se efectuează selecţia elementelor din poziţiile im, cu $i = 1, \ldots, p-1$, în secvenţa A ordonată; sunt necesare $p-1$ astfel de elemente, notate v_i, pentru a partiţiona A în p secvenţe.

2. Fiecare procesor împarte secvenţa locală în p părţi, în concordanţă cu cheile v_i; i.e. P_k găseşte subsecvenţele A_k^i ale lui A_k astfel încât $A_k^{i-1} < v_i \leq A_k^i$.

3. Procesorul P_k colectează secvenţele A_l^k de la celelalte procesoare; pe ansamblu, aceasta este o operaţie de schimb complet. În acest moment, P_k deţine toate subsecvenţele pentru care $v_{k-1} \leq A_l^k < v_k$, adică exact m elemente, cele dintre poziţiile km şi $(k+1)m-1$, în secvenţa A ordonată (pentru rigurozitate, $v_0 = -\infty$).

Algoritm de multiselecţie. Am văzut în secţiunea 5.2 un algoritm de selecţie a unei valori pentru arhitecturi cu memorie distribuită. Acum însă este vorba de selecţia a $p-1$ valori; desigur că aceste valori ar putea fi selectate una după alta, aplicând de $p-1$ ori succesiv algoritmul 5.9. Să vedem însă dacă nu se poate face totul în paralel.

În algoritmul 5.9, unul dintre procesoare avea rolul conducător: era cel care, în instrucţiunea 2.2, alegea unul singur dintre cei p candidaţi la selecţie; în acest timp, restul procesoarelor erau inactive. Vom executa de $p-1$ ori în paralel algoritmul 5.9, procesorul P_k având rolul conducător în căutarea valorii v_k. Pentru o mai bună înţelegere, rescriem detaliile algoritmului 5.9 şi, în cadrul său, ale algoritmului 5.5 de calcul al rangului. Notăm $\tilde{A}$ variabila locală în care P_k stochează A_k.

ALGORITM 5.25 *(Abali)* (determină cheile de partiţionare a secvenţei A sortată local în p subsecvenţe egale, pe MIMD cu memorie distribuită, $mp = n$, id $\equiv k$)

1. **pentru** $i = 1 : p-1$
   1. $s_i \leftarrow 0$, $d_i \leftarrow m-1$, $c_i \leftarrow \tilde{a}_{\lfloor im/p \rfloor}$ ($c_0 \leftarrow NIL$), $r_i \leftarrow -1$
2. **cât timp** $r_i \neq im$, pentru cel puţin un i, $1 \leq i \leq p-1$
   1. **schimb complet**: P_k primeşte candidaţii c_k de la celelalte procesoare
   2. P_k elimină valorile NIL, ordonează restul candidaţilor şi alege v_k mediana lor
   3. **difuzare generală**: P_k trimite tuturor valoarea v_k
   4. **pentru** $i = 1 : p-1$
      1. $l \leftarrow$ rangul lui v_i în $\tilde{A}$
      2. $r_i \leftarrow$ suma globală a valorilor l (cu algoritmul 4.4)
      3. **dacă** $r_i \neq im$ **atunci**
         1. **dacă** $r_i < im$ **atunci** $s_i \leftarrow$ cel mai mic indice pentru care $\tilde{a}_{s_i} > v_i$
         2. **altfel** $d_i \leftarrow$ cel mai mare indice pentru care $\tilde{a}_{d_i} < v_i$
         3. **dacă** $s_i \leq d_i$ **atunci** $c_i \leftarrow \tilde{a}_{\lfloor (s_i+d_i)/2 \rfloor}$
         4. **altfel** $c_i \leftarrow NIL$

Se observă transformări tipice în comunicaţia globală atunci când se simetrizează un algoritm: în instrucţiunea 2.1, colectarea s-a transformat în schimb complet, în 2.3 difuzarea în difuzare generală. La pasul 2.2 toate procesoarele lucrează acum în paralel. La ceilalţi paşi totul se repetă de $p-1$ ori. Să remarcăm că P_0 are o poziţie specială; din cauză că nu se caută o cheie v_0, el primeşte numai valori NIL, deci este practic inactiv.

Să vedem cum se modifică timpul de execuţie. Ne referim tot la hipercub. Faţă de algoritmul 5.9, toate complexităţile se înmulţesc cu p, mai puţin la pasul 2.1, unde $O(p)$ se transformă în $O(p \log p)$, şi la pasul 2.2, unde rămâne valabil $O(p)$. În total, cu aceleaşi consideraţii asupra numărului de iteraţii, putem aprecia timpul de lucru al algoritmului la $O(p \log^2 n)$ pentru operaţii aritmetice (comparaţii) şi la $O(p \log p \log n)$ pentru comunicaţie. Cu toate că algoritmul 5.9 necesită $O(\log^2 n)$ operaţii, se poate spune că algoritmul 5.25, deşi de complexitate $O(p \log^2 n)$, nu implică de p ori mai multe operaţii, ci ceva mai puţine.

Algoritmul de sortare echilibrată este următorul.

ALGORITM 5.26 (sortare echilibrată, pe MIMD cu memorie distribuită, $mp = n$, $\mathrm{id} \equiv k$)

1. sortează secvenţa locală A_k
2. aplică algoritmul 5.25 pentru găsirea cheilor de partiţionare $v_1, \ldots, v_{p-1}$
3. separă A_k în secvenţe A_k^i, cu $0 \leq i \leq p-1$, a.î. $A_k^{i-1} < v_i \leq A_k^i$
4. **schimb complet**: P_k colectează A_l^k de la celelalte procesoare P_l $(0 \leq l \leq p-1)$
5. ordonează cele p secvenţe A_l^k $(0 \leq l \leq p-1)$,
   prin interclasare după un arbore binar

La pasul 3 separarea secvenţei locale în p părţi se poate face fie prin $p-1$ căutări binare ale poziţiilor cheilor $v_1, \ldots, v_{p-1}$, în timp $O(p \log(n/p))$, fie printr-o simplă parcurgere secvenţială a secvenţei locale (ca la interclasare, secvenţa $\langle v_i \rangle$ fiind şi ea ordonată) în timp $O(n/p)$. La pasul 4, considerând că lungimea medie a mesajelor este n/p^2, timpul necesar schimbului complet va fi $O((n/p) \log p)$.

La pasul 5, fiecare procesor P_k deţine elementele din poziţiile de la km la $(k+1)m-1$ în secvenţa A sortată, deci exact cele care îi trebuiau, dar sub forma a p secvenţe, fiecare ordonată crescător. Pentru a profita de această structură, se poate face o interclasare după un arbore binar: cele p secvenţe se interclasează câte două, obţinându-se $p/2$ secvenţe care, la rândul lor, se interclasează câte două, etc. (este un algoritm asemănător sortării prin interclasare, doar că acolo secvenţele sunt de lungimi egale). La fiecare înjumătăţire a numărului de secvenţe, deoarece numărul total de elemente este n/p, se fac cel mult n/p comparaţii; deci, în total, $O((n/p) \log p)$ comparaţii, semnificativ mai puţin decât pentru o sortare obişnuită.

Sumând, obţinem pentru algoritmul 5.26 de sortare echilibrată un număr de comparaţii de $O((n/p) \log n + p \log^2 n)$ şi un timp de comunicaţie de $O((n/p) \log p)$ (dacă $(n/p) \log p > p \log p \log n$, adică $p^2 < n/\log n$, ceea ce poate fi rezonabil); cea mai mare pondere o au sortarea locală iniţială şi găsirea cheilor de partiţionare. Timpul de execuţie al acestui algoritm depinde în destul de slabă măsură de ordinea din secvenţa A. Comparaţii experimentale între el şi hiperquicksort au arătat că, pentru o repartiţie uniformă a elementelor din A, hiperquicksort se comportă mult mai bine; pentru diverse ordini particulare, cum ar fi de pildă cea în care $A_i < A_j$ pentru $i < j$, dar A_i nesortate, sortarea echilibrată are o execuţie mai rapidă.

Sortare cvasi-echilibrată. Trecem acum la un algoritm care, fără a realiza o încărcare perfect echilibrată a procesoarelor, furnizează mijloace pentru ca dezechilibrul să nu fie prea mare; el a fost publicat în [24].

Algoritmul 5.26 căuta cheile $v_1, \ldots, v_{p-1}$ aflate pe poziţiile $m, \ldots, (p-1)m$ în secvenţa sortată A. În locul determinării precise a acestora se pot căuta chei care să se afle aproape de poziţiile dorite. Ideea prezentată în continuare este foarte intuitivă; vom arăta că ea conduce la rezultate promiţătoare.

Descriere generală. Ca de obicei, fiecare procesor ordonează secvenţa locală. Apoi, din fiecare din aceste p secvenţe sunt extrase $p-1$ eşantioane; în mod natural, sunt alese cele din poziţiile $\lfloor im/p \rfloor$, cu $i = 1, \ldots, p-1$; dacă vă amintiţi, acestea sunt chiar valorile propuse de procesoare ca prim set de candidaţi în pasul (1) al algoritmului 5.25. Se obţin în total $p(p-1)$ valori, într-o secvenţă pe care o vom boteza Y. Se sortează Y şi se aleg cele $p-1$ chei de partiţionare, iarăşi destul de natural, ca fiind cele din poziţiile $i(p-1)$, cu $1 \leq i \leq p-1$ (se mai pot alege, de exemplu, cele din poziţiile $p/2$, $p/2+p$, $\ldots$, $p/2+(p-2)p$, adică mediana celor mai mici p chei, mediana următoarelor p, etc.). După care totul decurge ca în algoritmul 5.26: fiecare procesor partiţionează secvenţa proprie în acord cu $v_1, \ldots, v_{p-1}$, urmează un schimb complet în care un procesor colectează subsecvenţe cu aceeaşi poziţie în secvenţele locale şi le sortează apoi prin interclasare după un arbore binar.

Proprietăţi. Să vedem acum cât de echilibrată este încărcarea procesoarelor; aceasta este dictată de numărul de elemente care ajung în final la un anumit procesor.

PROPOZIŢIA 5.5 *In secvenţa A există cel puţin $[i(p-1)+1]m/p$ elemente mai mici decât v_i şi cel puţin $(p-i)(p-1)m/p$ elemente mai mari decât v_i.*

Demonstraţie. v_i este elementul din poziţia $i(p-1)$ în secvenţa eşantioanelor Y; aşadar sunt $i(p-1)$ elemente în Y mai mici decât v_i şi $p(p-1) - i(p-1) - 1 = (p-i)(p-1) - 1$ elemente mai mari. Dar fiecare element din Y poate fi privit ca reprezentantul a m/p elemente mai mici decât el; în plus, există m/p elemente mai mici decât v_i în secvenţa locală din care provine acesta, al căror reprezentant este v_i; cu totul, $[i(p-1)+1]m/p$ elemente mai mici decât v_i. La fel de bine, un element din Y este reprezentantul a m/p elemente mai mari decât el, de unde $(p-i)(p-1)m/p$ elemente mai mari decât v_i. ■

PROPOZIŢIA 5.6 *Numărul de elemente x din A pentru care $v_{i-1} < x \leq v_i$ este de cel mult $2m$.*

Demonstraţie. Pentru $1 \leq i \leq p-1$, numărul de elemente mai mari decât v_i este cel puţin $(p-i)(p-1)m/p$, iar numărul de elemente mai mici decât v_{i-1} este cel puţin $[(i-1)(p-1)+1]m/p$; numărul de elemente a căror valoare este între v_{i-1} şi v_i va fi cel mult $n - [(p-i)(p-1) + (i-1)(p-1) + 1]m/p = 2m - 2m/p < 2m$. Se poate verifica imediat că propoziţia este adevărată şi în cazurile $i = 0$ şi $i = p$. ■

PROPOZIŢIA 5.7 *Propoziţia 5.6 rămâne valabilă şi dacă $v_i = y_{p/2+(i-1)p}$, cu $1 \leq i \leq p-1$.*

Cum elementele $x \in A$ pentru care $v_{i-1} < x \leq v_i$ au destinaţia finală P_{i-1}, din propoziţia 5.6 rezultă că un procesor va deţine mai puţin de $2n/p$ elemente; deci, în cel mai rău caz, de două ori mai multe decât a avut iniţial.

Multiselecţie aproximativă. Algoritmul de sortare va fi deci identic cu 5.26, cu singura mică (dar importantă) diferenţă că, la pasul 2, găsirea cheilor de partiţionare se face cu un alt algoritm. Pe o arhitectură cu memorie distribuită, detalierea acestui pas ar putea fi următoarea.

ALGORITM 5.27 *(Shi)* (determină cheile de partiţionare a secvenţei A sortată local în p subsecvenţe de lungime $\leq 2m$, pe MIMD cu memorie distribuită, $mp = n$, id $\equiv k$, $\tilde{A}$ secvenţa locală)

1. formează secvenţa $X = \langle \tilde{a}_{\lfloor im/p \rfloor} \mid i = 1, \ldots, p-1 \rangle$
2. **colectare**: P_0 colectează secvenţele X şi le concatenează în secvenţa Y
3. P_0 sortează Y şi extrage $v_i = y_{i(p-1)}$ (sau $v_i = y_{p/2+(i-1)p}$), cu $1 \leq i \leq p-1$
4. **difuzare**: P_0 trimite tuturor valorile v_i

Colectarea şi difuzarea durează $O(p^2)$, respectiv $O(p \log p)$ (mesajele au lungime $p-1$), iar numărul de comparaţii este cel necesar sortării secvenţei Y, deci $p^2 \log p^2$ dacă aplicăm un algoritm uzual de sortare, sau, dacă ţinem seama de faptul că secvenţele Y sunt ordonate, $p^2 \log p$ aplicând sortarea prin interclasare după un arbore binar. Se vede deci că sortarea este cea care consumă cel mai mult timp. Problema **P5.4.11** arată cum se poate îmbunătăţi acest timp.

Considerând acum complexitatea algoritmului (să-l numim de sortare cvasi-echilibrată) 5.26 combinat cu algoritmul 5.27 (Shi), vom constata că ea nu se modifică decât la pasul 2, aşa cum am arătat. La pasul 5, dacă partiţionarea este aproape echilibrată, putem aprecia că se fac tot $(n/p) \log p$ comparaţii; în cel mai rău caz, când există secvenţe de lungime aproape $2n/p$, vor fi de două ori mai multe operaţii (dar numai dacă există $p/2$ partiţii de lungime $2n/p$ şi $p/2$ de lungime 0, iar la interclasare se combină cele lungi între ele, la fiecare pas). În medie sunt $O((n/p) \log n + p \log n)$ comparaţii iar timpul de comunicaţie este $O((n/p) \log p)$.

Algoritmul de sortare cvasi-echilibrată (Shi) este uşor adaptabil pentru o arhitectură cu memorie comună.

5.4.4 Sortare bitonică

Algoritmul de sortare ce va fi prezentat în continuare este un exemplu tipic de algoritm paralel, care nu se inspiră în nici un fel din cele secvenţiale (având, de altfel, o complexitate secvenţială mai mare de $O(n \log n)$). El este datorat, ca idee, lui Batcher [2].

Definiţii şi proprietăţi. Fie $A = \langle a_0, a_1, \ldots, a_{n-1} \rangle$ 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

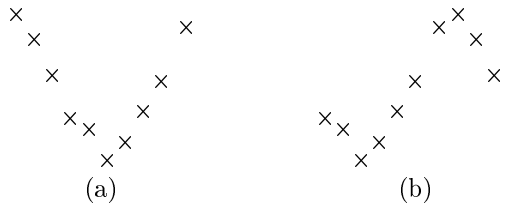


Figura 5.8: Secvenţe (a) unimodală (cu minim) (b) bitonică.

local sau maxim local şi *bitonică* dacă este o rotaţie a unei secvenţe unimodale; vezi figura 5.8. Următoarea lemă deschide calea către algoritm.

LEMA 5.1 *(Batcher) Fie* $A = \langle a_0, a_1, \ldots, a_{2N-1} \rangle$ *o secvenţă bitonică de lungime pară. Se definesc secvenţele* $m(A)$ *şi* $M(A)$ *astfel:*

$$m(A) = \langle \min(a_0, a_N),\ \min(a_1, a_{N+1}),\ \ldots,\ \min(a_{N-1}, a_{2N-1}) \rangle$$

$$M(A) = \langle \max(a_0, a_N),\ \max(a_1, a_{N+1}),\ \ldots,\ \max(a_{N-1}, a_{2N-1}) \rangle.$$

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

Demonstraţie. Putem presupune că secvenţa A este unimodală şi conţine un minim local; într-adevăr, dacă A este rotită cu p paşi la stânga (dreapta), atunci şi $m(A)$, $M(A)$ vor fi rotite cu p paşi la stânga (dreapta), deci îşi vor păstra proprietatea de bitonicitate; pe de altă parte, din orice secvenţă bitonică se poate obţine prin rotaţie o secvenţă unimodală cu minim (vezi problema 5.4.12).

Fie a_j minimul local din A. Să presupunem că $j < N$ (dacă $j \geq N$, considerăm secvenţa A în ordine inversă, ceea ce inversează ordinea în $m(A)$, $M(A)$, dar nu modifică bitonicitatea sau unimodalitatea).

Fie k indicele pentru care: $a_0 > a_N,\ \ldots,\ a_{k-1} > a_{N+k-1},\ a_k < a_{N+k},\ \ldots,\ a_{N-1} < a_{2N-1}$. Este evident că $0 \leq k \leq j$. Atunci

$$m(A) = \langle a_N, \ldots,\ a_{N+k-1},\ a_k,\ a_{k+1}, \ldots,\ a_j,\ a_{j+1}, \ldots,\ a_{N-1} \rangle$$

este o rotaţie a secvenţei $\langle a_k,\ a_{k+1}, \ldots,\ a_j,\ a_{j+1}, \ldots,\ a_{N-1},\ a_N, \ldots,\ a_{N+k-1} \rangle$ care este unimodală (cu a_j minim), deci $m(A)$ este bitonică.

$M(A)$ este o rotaţie a secvenţei unimodale $\langle a_0, \ldots, a_{k-1}, a_{N+k}, \ldots, a_{2N-1} \rangle$ (cu a_{k-1} sau a_{N+k} minim).

Pe de alta parte, pentru orice $x \in M(A)$, $y \in m(A)$ sunt adevărate inegalităţile $x > a_{k-1}$ sau $x > a_{N+k}$, $y < a_k$ sau $y < a_{N+k-1}$; în plus, avem: $a_{k-1} > a_k$, $a_{N+k-1} < a_{N+k}$ (din unimodalitate) şi $a_{k-1} > a_{N+k-1}$, $a_k < a_{N+k}$ (din ipotezele de mai sus); aşadar obţinem $x > y$. ■

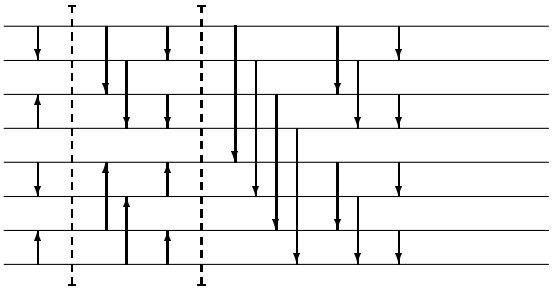


Figura 5.9: Sortare bitonică pentru 8 procesoare.

Ideea sortării bitonice. Algoritmul decurge în doi paşi. Întâi se transformă A într-o secvenţă bitonică, 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.

Revenind la primul pas, să observăm 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 listă de $n/2$ secvenţe bitonice 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 bitonică 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 a unei secvenţe bitonice (sau un alt algoritm, pe care-l vom prezenta mai jos, mai eficient în execuţia secvenţială).

Reţea de sortare bitonică. În figura 5.9 sunt prezentate comparaţiile care se efectuează într-o arhitectură cu 8 procesoare, 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 de paralelism este $n/2$ (au loc $n/2$ comparaţii în paralel).

Să deducem complexitatea algoritmului. Dacă $B(n)$ este numărul de comparaţii necesar pentru sortarea unei secvenţe bitonice folosind $n/2$ procesoare, atunci $B(2^i) = i$ (din lema lui Batcher). Atunci timpul de sortare a unei secvenţe oarecare este

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

pe un EREW PRAM (nu există conflicte nici la scriere, nici la citire). Costul algoritmului este deci $O(n \log^2 n)$; cum costul secvenţial este $O(n \log n)$, se pune problema

dacă algoritmul lui Batcher poate fi îmbunătăţit; răspunsul este pozitiv: se poate construi un algoritm paralel de sortare în $O(\log n)$, folosind $n/2$ procesoare, dar constanta ascunsă de $O(\cdot)$ este imensă. Practic, algoritmul lui Batcher rămâne deocamdată optim.

Sortare bitonică pe hipercub

Cazul $p = n$. Să vedem cum se poate implementa algoritmul pe o arhitectură cu memorie distribuită, mai precis pe un hipercub. Considerăm întâi cazul în care există un singur element pentru fiecare procesor: $p = n$. Dacă ne uităm puţin pe figura 5.9, imaginându-ne acum că fiecare linie reprezintă un procesor, observăm că un procesor k comunică doar cu procesoare $k \pm 2^i$, dar întotdeauna astfel încât într-un hipercub au loc doar comunicaţii între vecini. Algoritmul este următorul:

ALGORITM 5.28 (sortare bitonică pe hipercub, $p = n = 2^d$, id $\equiv k$)

1. **pentru** $l = 0 : d-1$
   1. **pentru** $i = l : -1 : 0$
      1. **dacă** $k_{l+1} = 0$ **atunci**
         1. **dacă** $k_i = 0$ **atunci** $exch(k, k + 2^i)$
         2. **altfel** $exch(k - 2^i, k)$
      2. **altfel** $\{k_{l+1} = 1\}$
         1. **dacă** $k_i = 0$ **atunci** $exch(k + 2^i, k)$
         2. **altfel** $exch(k, k - 2^i)$

Singura problemă este stabilirea sensului în care se fac interschimbările; mecanismul din algoritm este uşor de înţeles dacă observăm că, la pasul l, elementul dintr-un procesor k face parte dintr-o secvenţă care trebuie ordonată crescător dacă bitul $l+1$ al reprezentării binare a lui k este 0 şi descrescător dacă acesta este 1 (prin convenţie $k_{d+1} = 0$). Pentru un l fixat, interschimbările se fac astfel încât, pentru o pereche de procesoare P_k, P_j (cu $k = j \pm 2^i$) 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.

Complexitatea acestui algoritm este clar $\sum_{l=0}^{d-1} \sum_{i=0}^{l} O(1) = O(\log^2 p)$, la fiecare pas având loc o comunicaţie între vecini şi o comparaţie.

Cazul $p \ll n$. Ar fi naiv să invocăm principiul lui Brent şi deci (pe un PRAM) să păstrăm algoritmul neschimbat, modificând doar alocarea procesoarelor; timpul de execuţie ar fi de $O((n/p)\log^2 p)$, ori trebuie să tindem către $O((n/p)\log p)$ pentru a spera o eficienţă apropiată de 1.

Rămânem în cadrul arhitecturilor cu memorie distribuită. Să presupunem că fiecare procesor se ocupă de $m = n/p$ elemente. În primul rând, fiecare procesor P_k sortează elementele sale într-o secvenţă A_k crescătoare dacă k este par şi descrescătoare dacă e impar; aceasta durează $O((n/p)\log(n/p))$. În continuare, se poate aplica algoritmul de sortare bitonică, înlocuind elementele a_i cu secvenţe ordonate A_i, toate de aceeaşi lungime; formarea secvenţelor de tipul $m(A)$ şi $M(A)$ se face simplu,

$\min(A_i, A_j)$ fiind acum o secvenţă obţinută prin alegerea fiecărui minim dintre elementele aflate pe aceeaşi poziţie în secvenţele A_i, A_j. Şi totuşi mai trebuie adăugat ceva.

Să exemplificăm pentru un grup de 4 procesoare; presupunem că secvenţa $\langle A_0, A_1\rangle$ este ordonată crescător (adică A_0 şi A_1 sunt fiecare ordonate crescător şi, în plus, $A_0 < A_1$) şi că secvenţa $\langle A_2, A_3\rangle$ este ordonată descrescător. Deci secvenţa $A = \langle A_0, A_1, A_2, A_3\rangle$ este bitonică (chiar unimodală); în algoritmul de sortare bitonică aceasta este situaţia după primul pas ($l = 0$ în algoritmul 5.28, vezi şi figura 5.9). Urmează acum operaţiile (pasul al doilea pentru cele patru procesoare):

$$\begin{array}{cccc} A_0 \leftarrow \min(A_0, A_2) & A_1 \leftarrow \min(A_1, A_3) & A_2 \leftarrow \max(A_0, A_2) & A_3 \leftarrow \max(A_1, A_3) \\ A_0 \leftarrow \min(A_0, A_1) & A_1 \leftarrow \max(A_0, A_1) & A_2 \leftarrow \min(A_2, A_3) & A_3 \leftarrow \max(A_2, A_3). \end{array}$$

Lema lui Batcher ne asigură că $A_0 < A_1 < A_2 < A_3$; dacă ar fi vorba doar de elemente (adică $|A_i| = 1$) am fi satisfăcuţi cu atât; problema e că elementele din nici un A_i nu sunt ordonate crescător. Deci, pentru ca secvenţa A să fie global ordonată, mai trebuie ordonate (încă o dată) elementele din fiecare A_i. Din fericire nu trebuie să mai pierdem $O((n/p)\log(n/p))$ pentru aceasta; aceeaşi lemă ne spune că secvenţele A_i sunt bitonice; ori ele pot fi ordonate mai repede (dar nu aplicând sortarea bitonică—aceasta conduce la un timp şi mai mare: $O((n/p)\log^2(n/p))$; aici ne-ar aduce respectarea ad litteram a principiului lui Brent).

Sortarea secvenţială a unei secvenţe unimodale. (Sortarea unei secvenţe bitonice este lăsată ca exerciţiu.) O secvenţă unimodală cu minim este compusă din două subsecvenţe, una descrescătoare, alta crescătoare. Dacă o privim pe cea descrescătoare în sens invers, obţinem una crescătoare. Secvenţa sortată se obţine prin interclasarea celor două subsecvenţe crescătoare, deci în timp $O(n/p)$; mai detaliat: cel mult $\log(n/p)$ pentru găsirea minimului şi cel mult n/p pentru interclasare.

ALGORITM 5.29 (sortarea secvenţei unimodale A cu minim, secvenţial)

1. găseşte j, indicele elementului minim
2. interclasează $\langle a_j, a_{j-1}, \ldots, a_0\rangle$ cu $\langle a_{j+1}, a_{j+2}, \ldots, a_{m-1}\rangle$

Sortare bitonică pe hipercub. Putem scrie acum algoritmul de sortare bitonică pe un hipercub, în cazul general; notăm A' secvenţa pe care o primeşte la un moment dat P_k de la un vecin al său (sunt necesare deci n/p locaţii de memorie suplimentare).

ALGORITM 5.30 (sortare bitonică pe hipercub, $p = 2^d \ll n$, varianta 1, id $\equiv k$)

1. sortează elementele din A_k, crescător dacă $k_0 = 0$, descrescător dacă $k_0 = 1$
2. **pentru** $l = 0 : d-1$
    1. **pentru** $i = l : -1 : 0$
        1. **în paralel** **send**(A_k, i), **recv**(A', i)
        2. **dacă** $k_{l+1} = k_i$ **atunci** $A_k \leftarrow \min(A_k, A')$
        3. **altfel** $A_k \leftarrow \max(A_k, A')$
        4. sortează secvenţa bitonică A_k, crescător dacă $k_{l+1} = 0$, descrescător dacă $k_{l+1} = 1$

Numărul de comparaţii efectuat de acest algoritm este (termenul $2n/p$ corespunde complexităţii sortării unei secvenţe bitonice, vezi problemele)

$$\frac{n}{p}\log\frac{n}{p}+\sum_{l=0}^{d-1}\left(\frac{2n}{p}+\sum_{i=0}^{l}\frac{n}{p}\right)\approx\frac{n}{p}\log n+\frac{n}{p}\log p+\frac{n}{2p}\log^2 p=O(\frac{n}{p}\log n) \quad (5.7)$$

şi deci obiectivul propus a fost atins. În ce priveşte comunicaţia, aceasta ocupă timpul

$$\sum_{l=0}^{d-1}\sum_{i=0}^{l}(\sigma+\frac{n}{p}\beta)=(\sigma+\frac{n}{p}\beta)\log^2 p=O(\frac{n}{p}\log^2 p).$$

Se poate concluziona că algoritmul propus are o eficienţă asimptotic egală cu 1. Vom vedea că se pot găsi algoritmi cu o complexitate mai bună a comunicaţiei, deci cu un overhead datorat comunicaţiei mai mic. În ce priveşte overhead-ul de încărcare inegală, algoritmul 5.30 stă foarte bine: procesoarele au aceeaşi cantitate de operaţii de efectuat; singura etapă la care pot interveni diferenţe—însă nesemnificative—este cea de ordonare locală a secvenţelor bitonice.

Alt algoritm de sortare bitonică. Să deducem acum o variantă mai simplă şi mai eficientă a sortării bitonice pe hipercub. Pentru claritate, să precizăm câţiva termeni. Spunem că, în secvenţa $A=\langle A_0,\ldots,A_{p-1}\rangle$, A_i este un minim local dacă $A_i<A_{i-1}$ şi $A_i<A_{i+1}$; definiţia este aceeaşi ca pentru cazul în care A_i ar fi elemente, doar semnificaţia semnului $<$ s-a schimbat. În continuare, noţiunile de unimodalitate şi bitonicitate se definesc la nivel de subsecvenţă la fel ca la nivel de element.

În definirea secvenţelor $m(A)$ şi $M(A)$, subsecvenţele componente au forma $\min(A_i,A_j)$, respectiv $\max(A_i,A_j)$; observaţia esenţială este că, în acest fel, secvenţele $m(A)$ şi $M(A)$ sunt definite în mod *unic* din punct de vedere al elementelor componente (nu şi al ordinii lor, evident). Indiferent de ordinea elementelor în interiorul unei subsecvenţe, dacă A este bitonică la nivel de subsecvenţe, atunci $m(A)$ şi $M(A)$ din lema lui Batcher vor fi bitonice la nivel de subsecvenţe. Atunci nu mai trebuie să avem grijă ca secvenţa A să fie în permanenţă bitonică la nivel de element, aşa cum am făcut în algoritmul 5.30, ci doar bitonică la nivel de subsecvenţă.

Mai rămâne de stabilit metoda de obţinere a subsecvenţelor de genul $\min(A_i,A_j)$. Ideea cea mai la îndemână e de a ţine în permanenţă secvenţele A_i sortate crescător, şi de a obţine minimul prin interclasare. În acest fel, dacă secvenţele A_i, A_j au fiecare câte m elemente, subsecvenţa minim se calculează cu doar m comparaţii şi este ordonată crescător.

Toate aceste consideraţii ne arată că putem aplica foarte bine algoritmul 5.28 şi în cazul $p\ll n$, cu două amendamente: elementele locale unui procesor să fie iniţial sortate crescător şi operaţiile de tip $exch(i,j)$ să fie acum la nivel de subsecvenţă şi să fie efectuate aşa cum a fost descris în secţiunea dedicată interclasării.

ALGORITM 5.31 (sortare bitonică pe hipercub, $p=2^d\ll n$, varianta 2, id $\equiv k$)

1. sortează crescător elementele din A_k (locale)
2. apelează algoritmul 5.28, cu $exch()$ la nivel de subsecvenţă

Numărul de comparaţii efectuat de acest algoritm este

$$T(n,p) = \frac{n}{p}\log\frac{n}{p} + \sum_{l=0}^{d-1}\sum_{i=0}^{l}\frac{n}{p} \approx \frac{n}{p}\log n + \frac{n}{2p}\log^2 p \tag{5.8}$$

şi se dovedeşte mai mic decât cel din (5.7). Deci, nu numai că algoritmul 5.31 este mai simplu decât 5.30, dar are şi un număr de comparaţii mai mic; în ce priveşte comunicaţia, algoritmii implică acelaşi efort.

Vom vedea, în problema 5.4.16, că efortul deducerii algoritmului 5.30 nu a fost totuşi inutil. În cazul memoriei partajate, acesta este cel mai eficient.

5.4.5 Sortare prin interclasare

Algoritm secvenţial. Sortarea prin interclasare (mergesort) are o descriere recursivă foarte scurtă: se împarte secvenţa A în două jumătăţi, se sortează fiecare dintre ele şi apoi se interclasează secvenţele ordonate; înjumătăţirea secvenţelor are loc până când lungimea secvenţelor este 1, deci ele sunt fiecare ordonate.

ALGORITM 5.32 (mergesort, forma generală secvenţială)

procedură *mergesort*(s,d)

1. **cât timp** $s \neq d$ {secvenţa are mai mult de un element}
   1. $x \leftarrow \lfloor (s+d)/2 \rfloor$ {poziţia mediană}
   2. *mergesort*(s,x) {sortează prima jumătate}
   3. *mergesort*$(x+1,d)$ {sortează a doua jumătate}
   4. interclasează $A(s:x)$ cu $A(x+1:d)$ {cele două jumătăţi}

Dacă nu vă place descrierea recursivă, puteţi gândi altfel: se interclasează $n/2$ perechi de secvenţe de un element, apoi $n/4$ perechi de secvenţe de două elemente, etc. Complexitatea secvenţială a algoritmului respectă $T(n) \leq T(n/2)+n$ (interclasarea a două secvenţe de lungime $n/2$ se face cu cel mult n comparaţii), deci $T(n) \leq 2n\log n$. Sortarea prin interclasare implică de două ori mai multe operaţii decât quicksort, în cazul cel mai favorabil pentru acesta din urmă; marea deosebire în ce priveşte timpii de execuţie este că pentru mergesort variaţiile sunt destul de mici, deci $2n\log n$ este o bună aproximare, în timp ce pentru quicksort sunt mai mari. Totuşi, în general, quicksort este mai rapid.

Algoritmi pe PRAM. Din forma algoritmului 5.32 se vede că o variantă paralelă a sa necesită numai un bun algoritm paralel de interclasare. Altfel, toate interclasările de secvenţe de aceeaşi lungime pot decurge în paralel, începând de la cele $n/2$ interclasări de secvenţe de lungime 1, şi până la unica interclasare de secvenţe de lungime $n/2$. Este acelaşi gen de paralelism ca şi la quicksort, numai că aici gradul mare de paralelism este la început, dimensiunile secvenţelor de interclasat crescând, în timp ce la quicksort era invers: prin partiţionare se obţineau secvenţe din ce în ce mai mici; mai există o diferenţă; aici dimensiunile se dublează după fiecare interclasare, în timp ce acolo subsecvenţele obţinute prin partiţionare nu erau egale.

Din punct de vedere teoretic, pe un CREW PRAM, folosind $O(n)$ procesoare, putem aplica pentru interclasare algoritmul Valiant 5.11; cu acesta, pentru interclasarea a două secvenţe de lungime t cu $O(t)$ procesoare este nevoie de $O(\log\log t)$ comparaţii. Deoarece pe tot timpul sortării prin interclasare suma elementelor din secvenţele care se interclasează în paralel este n, rezultă că sunt necesare $O(n)$ procesoare. Numărul de comparaţii va fi $O(\log\log 1 + \log\log 2 + \ldots + \log\log(n/4) + \log\log(n/2))$, unde fiecare termen corespunde unui nivel de recursie (primii sunt evident greşiti, primul chiar matematic absurd; ei pot fi oricum neglijaţi, ca fiind mici; am vrut să fie subliniată dimensiunea secvenţelor de interclasat); efectuând suma (vezi **P5.4.20**) se obţine o complexitate de $O(\log n \log\log n)$ pentru problema sortării. Costul acestui algoritm nu este totuşi optim, ci $O(n \log n \log\log n)$; există un algoritm optim, care asigură sortarea în timp $O(\log n)$, cu $O(n)$ procesoare, dar el este destul de complicat; totuşi, se bazează tot pe interclasare.

În cazul $p \ll n$, pe un CREW PRAM se poate aplica algoritmul de interclasare 5.16 (Xiong). Fiecare procesor sortează secvenţial cele $m = n/p$ elemente ale sale. Apoi, procesoarele se grupează câte două pentru a interclasa două secvenţe de lungime m, câte patru pentru a interclasa secvenţe de lungime $2m$, câte opt pentru a interclasa secvenţe de lungime $4m$, etc. Timpul necesar acestor operaţii este

$$\begin{aligned} T(n,p) &= \frac{n}{p}\log\frac{n}{p} + \sum_{i=1}^{\log p}\left[\frac{2\cdot 2^{i-1}m}{2^i} + \frac{\log(2\cdot 2^{i-1}m/2^i)}{\log(\lfloor 2^i/2^{i-1}\rfloor+1)} + O(\log\log p)\right] \\ &= \frac{n}{p}\log n + \frac{1}{\log 3}\log\frac{n}{p}\log p + O(\log p\log\log p). \end{aligned} \tag{5.9}$$

Primul termen este pentru sortarea secvenţială, iar cei din sumă sunt luaţi din (5.4). Pentru p mic, eficienţa acestui algoritm va fi foarte bună.

Cazul memoriei distribuite. După cum am văzut în secţiunea dedicată interclasării, pe o arhitectură cu memorie distribuită nu există un algoritm eficient pentru a rezolva această problemă. Pe un hipercub, după sortarea secvenţelor locale, doar prima etapă de interclasare decurge uşor, cea în care perechi procesoare îşi interclasează secvenţele locale printr-un $exch()$; apoi ar urma ca grupuri de patru să interclaseze două secvenţe ordonate distribuite pe câte două procesoare, grupuri de opt să interclaseze secvenţe ordonate distribuite pe patru procesoare, etc. Nu se poate obţine un mod eficient de comunicaţie (sau, cel puţin, el nu a fost găsit până în prezent). Dacă nu aţi făcut-o deja, vedeţi acum problema 5.4.17 pentru o frumoasă reţea de sortare prin interclasare, şi pentru comunicaţia pe care ar implica-o într-un hipercub.

5.4.6 Ce algoritm de sortare alegem ?

Am prezentat o mulţime de algoritmi care rezolvă aceeaşi problemă, fiecare în alt mod; şi să nu vă închipuiţi că aceştia sunt toţi. Pe care dintre ei îl alegem ? De ce nu ne-am mulţumit să descriem unul singur, cel mai bun ?

Să restrângem discuţia la calculatoarele cu memorie distribuită. În tabelul 5.2 sunt recapitulate metodele paralele de sortare din acest capitol, împreună cu complexităţile

Algoritm	Comparaţii	Comunicaţie
Sortare par-impar (alg. 5.19)	$O(\frac{n}{p}\log n + n)$	$O(n)$
Hiperquicksort (Wagar, alg. 5.24)	$O(\frac{n}{p}\log n)$	$O(\frac{n}{2p}\log p)$
Sortare echilibrată (Abali, alg. 5.26+5.25)	$O(\frac{n}{p}\log n + p\log^2 n)$	$O(\frac{n}{p}\log p)$
Sortare cvasi-echilibrată (Shi, alg. 5.26+5.27)	$O(\frac{n}{p}\log n + p\log n)$	$O(\frac{n}{p}\log p)$
Sortare bitonică (Batcher, alg. 5.31)	$O(\frac{n}{p}\log n + \frac{n}{2p}\log^2 p)$	$O(\frac{n}{p}\log^2 p)$

Tabelul 5.2: Complexităţile metodelor de sortare pentru calculatoare MIMD cu memorie distribuită, prezentate în acest capitol.

lor—număr de comparaţii şi număr de elemente comunicate, în medie. Dintre cele 5 metode, una singură este potrivită topologiei de inel, sortarea par-impar. Deci, dacă procesoarele calculatorului pe care îl avem sunt conectate în inel, aceasta e metoda aleasă; ceilalţi algoritmi se pot şi ei implementa pe inel, dar va creşte complexitatea comunicaţiei, ceea ce elimină interesul pentru ei. Pe de altă parte, dacă avem un hipercub, ar fi păcat să alegem sortarea par-impar, care nu profită decât de puţine dintre conexiunile existente.

Alte criterii sunt dificil de generalizat; este vorba despre valoarea raportului viteză de comunicaţie/viteză de calcul pentru calculatorul respectiv; de comportarea algoritmilor în medie şi în cazul cel mai defavorabil; de cât de aproape de medie se află în general timpul de execuţie; de constantele ascunse de notaţia $O(\cdot)$; de caracteristici hardware sau software ale calculatorului, care pot fi speculate mai uşor sau mai greu de către un algoritm.

De obicei decizia vine pe baza posibilităţilor calculatorului şi, nu în ultimul rând, a timpului disponibil pentru scrierea programului. Scopul acestei cărţi e de a oferi o gama mare de variante, nu numai pentru a avea de unde alege, dar şi pentru a dezvolta instrumente aplicabile altor probleme, netratate aici.

Probleme

P 5.4.1 Doi algoritmi secvenţiali foarte populari de sortare cu complexitate $O(n^2)$ sunt metoda bulelor (bubblesort) şi sortarea prin inserţie. Îi prezentăm mai jos în forma lor cea mai simplă (se pot introduce unele teste care diminuează numărul de operaţii):

BUBBLESORT
pentru $i = n-1 : -1 : 1$
  pentru $j = 0 : i-1$
    $exch(j, j+1)$

SORTARE PRIN INSERŢIE
pentru $i = 1 : n-1$
  pentru $j = i-1 : -1 : 0$
    $exch(j, j+1)$

Desenaţi reţelele de sortare corespunzătoare acestor algoritmi şi apreciaţi timpul de execuţie, nivelul de paralelism, etc.

P 5.4.2 Scrieţi algoritmul de sortare par-impar pentru un EREW PRAM, în cazurile $p = n/2$, $p \ll n$.

P 5.4.3 Pentru n impar, desenaţi reţelele analoage celor din figura 5.5. Sunt ele identice aşa cum am afirmat în text ?

P 5.4.4 Fie α o reţea de sortare simplă cu $\binom{n}{2}$ comparatoare. Atunci α^R, reţeaua conţinând aceleaşi comparatoare, dar în ordine inversă, este o reţea de sortare.

P 5.4.5 Ce modificări trebuie aduse algoritmului 5.21 şi procedurii 5.22 pentru adaptarea la un EREW PRAM ?

P 5.4.6 Să presupunem că, în algoritmul 5.23, este aleasă ca pivot mediana unui eşantion de t elemente. Care este valoarea minimă pentru t astfel încât algoritmul să nu se oprească înainte de sortarea locală (aceasta se poate întâmpla dacă un procesor care trebuie să aleagă un pivot rămâne fără nici un element din secvenţă) ?

P 5.4.7 Este algoritmul de sortare echilibrată 5.26 într-adevăr echilibrat dacă în A pot exista elemente egale ?

P 5.4.8 Fie $X = \langle X_0, \ldots, X_{p-1} \rangle$ o secvenţă de lungime n, fiecare X_i fiind ordonată crescător şi având lungime n_i. Scrieţi algoritmul (secvenţial) de sortare a lui X, prin interclasare după un arbore binar. De câtă memorie suplimentară este nevoie ?

P 5.4.9 Scrieţi algoritmul de sortare cvasi-echilibrată (Shi) pentru un PRAM.

P 5.4.10 Presupunem că trei procesoare deţin secvenţele $\langle 0, 1, 2, 9, 16, 17, 24, 25, 27, 28, 30, 33 \rangle$, $\langle 7, 8, 11, 12, 13, 18, 19, 21, 23, 29, 34, 35 \rangle$, respectiv $\langle 3, 4, 5, 6, 10, 14, 15, 20, 22, 26, 31, 32 \rangle$, fiecare de lungime 12, deja sortate. Care sunt cheile de partiţionare calculate de algoritmul 5.27 ? Câte elemente va deţine în final fiecare procesor ?

P 5.4.11 Pe un hipercub, algoritmul 5.27 poate fi îmbunătăţit după următoarea idee: interclasarea mesajelor X poate fi făcută în timpul colectării; pe un arbore de acoperire binomial, fiecare procesor care primeşte un X de la un procesor mai depărtat decât el de P_0 interclasează secvenţa primită cu cea proprie şi trimite rezultatul mai departe. Scrieţi algoritmul şi apreciaţi timpul de execuţie.

P 5.4.12 Să se arate că dintr-o secvenţă bitonică se pot obţine, prin rotaţie, secvenţe unimodale cu minim, respectiv cu maxim.

P 5.4.13 Scrieţi algoritmul secvenţial de găsire a minimului unei secvenţe unimodale A, de lungime m, cu minim. Se poate adapta acest algoritm în cazul în care A este bitonică ?

P 5.4.14 Detaliaţi algoritmul 5.29 pentru cazul unei secvenţe bitonice.

P 5.4.15 Găsiţi o planificare a procesoarelor pentru sortarea bitonică pe un EREW PRAM, în cazul $p = n/2$.

P 5.4.16 Găsiţi o variantă eficientă pentru sortarea bitonică pe un EREW PRAM, când $p \ll n$.

P 5.4.17 Un alt algoritm paralel de sortare propus de Batcher poate fi descris recursiv după cum urmează; în esenţă, este un algoritm de sortare prin interclasare, numit sortare cu interclasare par-impară (nici o legătură cu sortarea par-impar). Se împarte secvenţa de sortat C în două subsecvenţe, $C = \langle A, B \rangle$; se sortează A şi B; apoi acestea se interclasează tot în mod recursiv: se interclasează secvenţele "pare" $\langle a_0, a_2, \ldots \rangle$ şi $\langle b_0, b_2, \ldots \rangle$, rezultatul fiind $\langle x_0, x_1, \ldots \rangle$; se interclasează secvenţele "impare" $\langle a_1, a_3, \ldots \rangle$ şi $\langle b_1, b_3, \ldots \rangle$, obţinându-se

$\langle y_0, y_1, \ldots\rangle$; se consideră secvenţa $Z = \langle x_0, y_0, x_1, y_1, x_2, y_2, \ldots\rangle$, căreia i se aplică operaţiile de comparare-interschimbare $exch(y_0, x_1)$, $exch(y_1, x_2)$, ...; în urma acestor operaţii $Z = A \perp B$.

Demostraţi corectitudinea acestui algoritm folosind următoarea teoremă, deosebit de utilă, formulată iniţial pentru reţelele de sortare:

TEOREMĂ (principiul zero-unu). Dacă o reţea cu n linii de intrare (un algoritm cu $n = |C|$) sortează nedescrescător toate cele 2^n secvenţe de zero şi unu, atunci va sorta orice secvenţă de n numere în ordine nedescrescătoare.

P 5.4.18 Desenaţi o reţea de sortare conformă metodei din problema anterioară. Apreciaţi timpul de execuţie. Studiaţi posibilitatea de adaptare a acestui algoritm la o arhitectură MIMD cu memorie distribuită, comparând apoi cu sortarea bitonică.

P 5.4.19 Algoritmul de sortare bitonică poate fi descris şi altfel, tot sub formă recursivă. Dacă $A = \langle a_0, a_1, \ldots\rangle$ este unimodală, atunci se sortează secvenţele unimodale $\langle a_0, a_2, \ldots\rangle$ ("pară") şi $\langle a_1, a_3, \ldots\rangle$ ("impară") şi se efectuează operaţiile $exch(a_0, a_1)$, $exch(a_2, a_3)$, ... Demonstraţi corectitudinea algoritmului folosind principiul zero-unu. Verificaţi că reţeaua de sortare care rezultă este identică cu cea pentru sortarea bitonică.

P 5.4.20 Demonstraţi că $\log\log n + \log\log(n/2) + \ldots + \log\log 2 = O(\log n \log\log n)$.

Capitolul 6

Algoritmi numerici

Primii algoritmi pentru care s-a pus acut problema paralelizării au fost cei numerici, deoarece ei sunt folosiţi în rezolvarea problemelor ştiinţifice de mari dimensiuni, pentru care nevoia de viteză de calcul e imperioasă. De aceea, pe primele calculatoare paralele, algoritmii numerici au fost implementaţi cu prioritate.

Să precizăm – deşi e probabil inutil – că algoritmii numerici sunt cei în care se efectuează calcule aritmetice cu numere reale (sau complexe, caz pe care nu-l abordăm, el neimplicând mari diferenţe), reprezentate în format virgulă mobilă. Exemplele de probleme numerice sunt numeroase: începând de la înmulţirea matrice-vector, tratată într-un capitol anterior, şi de la rezolvarea de sisteme de ecuaţii liniare de diverse tipuri, până la, să zicem, rezolvarea de ecuaţii diferenţiale, sau cu derivate parţiale. Spre deosebire, sortarea unor numere reale nu este o problemă numerică, deoarece, deşi se efectuează operaţii (comparaţii, adică scăderi), rezultatul lor nu este folosit mai departe; valorile reale nu se modifică pe întreg parcursul algoritmilor de sortare, ci numai locurile lor în memorie, adică ordinea lor.

Scopul acestei lucrări fiind în primul rând de iniţiere, ne vom opri la prezentarea de algoritmi pentru probleme simple din punct de vedere al tratărilor matematică şi algoritmică. Este vorba de înmulţirea de matrice şi de rezolvarea de sisteme liniare triunghiulare, dense (cele obişnuite; în acest context se va vorbi în mod natural şi despre factorizarea LU) şi tridiagonale; deci, capitole de bază ale algebrei liniare. Există, în plus, şi un alt interes pentru algoritmi paraleli ce rezolvă aceste probleme; ei sunt folosiţi în numeroase alte probleme, care se reduc la, sau conţin ca etape intermediare, aceşti algoritmi numerici fundamentali. Vom insista îndeosebi asupra algoritmilor dedicaţi arhitecturilor cu memorie distribuită, aici intervenind o serie de tehnici de paralelizare de mare interes.

Pentru a înţelege acest capitol cititorului îi sunt necesare cunoştinţe de matematică la nivelul anului I de facultate (cel mult) şi parcurgerea capitolelor referitoare la sisteme liniare ale unui manual de metode numerice, pentru familiarizarea cu algoritmii secvenţiali. Nu vom aborda mai deloc problema stabilităţii numerice a algoritmilor

(cât de "aproape" de soluţia exactă a problemei este cea calculată numeric) – care este, de fapt, piatra de încercare a algoritmilor numerici; aceasta nu înseamnă că am uitat-o, ci că punem accentul pe aspectele legate de paralelism.

Deoarece în toate problemele pe care le vom prezenta se lucrează cu matrice, să prezentăm întâi câteva notaţii, păstrate de-a lungul întregului capitol. Fie $A \in \mathbf{R}^{n\times n}$, deci o matrice pătrată, de dimensiune $n \times n$; nu vom lucra cu matrice dreptunghiulare; în singurul loc unde ar fi într-adevăr cazul, înmulţirea de matrice, generalizările sunt uşoare. Notăm prin a_{ij}, sau prin $A(i,j)$, elementul din linia i, coloana j, al matricei A; vom numerota liniile şi coloanele începând de la zero.

Pentru a descrie algoritmi la nivel de blocuri de matrice (şi nu la nivel de element), se consideră următoarele partiţionări ale matricei A:

$$A = \begin{bmatrix} A_{00} & A_{01} & \cdots & A_{0,n_b-1} \\ A_{01} & A_{11} & \cdots & A_{1,n_b-1} \\ \vdots & \vdots & & \vdots \\ A_{n_b-1,0} & A_{n_b-1,1} & \cdots & A_{n_b-1,n_b-1} \end{bmatrix} = [A_0 A_1 \ldots A_{n_b-1}] = \begin{bmatrix} A_0^T \\ A_1^T \\ \vdots \\ A_{n_b-1}^T \end{bmatrix} \tag{6.1}$$

în care: $A_{ij} \in \mathbf{R}^{r\times r}$ este un bloc pătrat; $A_j \in \mathbf{R}^{n\times r}$ este o bloc-coloană; $A_i^T \in \mathbf{R}^{r\times n}$ este o bloc-linie. În notaţie MATLAB:

$$\begin{aligned} A_{ij} &\equiv A(ir : (i+1)r-1, jr : (j+1)r-1), \\ A_j &\equiv A(:, jr : (j+1)r-1), \\ A_i^T &\equiv A(ir : (i+1)r-1, :). \end{aligned}$$

Presupunem că n se divide cu r, şi deci numărul de blocuri pe o dimensiune este $n_b = n/r$. Desigur că presupunerea că blocurile A_{ij} sunt pătrate este simplificatoare, dar ea uşurează înţelegerea, fără a diminua prea mult generalitatea. Atenţie, notaţiile A_i şi A_i^T pot genera confuzii; aceste două matrice nu sunt una transpusa celeilalte; ele nu vor fi folosite niciodată împreună.

6.1 Înmulţirea de matrice

Înmulţirea de matrice este operaţia cea mai folosită în algoritmii din algebra liniară, în special în variantele bazate pe apeluri la rutine BLAS (Basic Linear Algebra Subroutines) de nivel 3 (în aceste variante, algoritmii sunt descrişi la nivel de bloc, şi reduşi la câteva operaţii simple, implementate foarte eficient; vom prezenta o astfel de versiune pentru factorizarea LU). Ne vom ocupa în această secţiune de prezentarea câtorva algoritmi bazaţi pe idei diferite şi având în vedere mai multe arhitecturi ţintă.

Vom expune întâi problema şi vom deduce complexitatea ei paralelă, apoi vom prezenta algoritmi pentru arhitecturi MIMD cu memorie distribuită, pentru topologiile de inel şi tor, iar în final algoritmi pentru arhitecturi MIMD cu memorie partajată.

Referinţele de bază în alcătuirea acestei secţiuni au fost [12, 13, 5].

6.1.1 Complexitatea problemei

Fie A, $B \in \mathbf{R}^{n\times n}$ două matrice. Notăm prin $MM(n)$ problema calculului matricei produs $C = A \cdot B \in \mathbf{R}^{n\times n}$. Prin definiţie:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}, \quad \text{pentru} \quad i,j \in 0 : n-1 \tag{6.2}$$

Denumim folosirea explicită a acestei formule *metoda standard* pentru rezolvarea $MM(n)$ (spre a o diferenţia de metodele rapide, care nu vor fi însă studiate în această lucrare; vezi [11]).

La nivel de blocuri, calculul produsului $C = A \cdot B$ se poate face folosind partiţionările din relaţia (6.1). Vom prezenta mai multe variante, fiecare sugerând o altă paralelizare. O primă formulă este similară cu cea la nivel de element (6.2):

$$C_{ij} = \sum_{k=0}^{n_b-1} A_{ik}B_{kj}, \quad \text{pentru} \quad i,j \in 0 : n_b - 1 \tag{6.3}$$

în care pentru calculul fiecărui produs de blocuri se aplică formula la nivel de element.

O a doua formă reprezintă calculul unei bloc coloane din C:

$$C_j = A \cdot B_j = \sum_{i=0}^{n_b-1} A_i \cdot B_{ij}, \quad \text{pentru} \quad j \in 0 : n_b - 1. \tag{6.4}$$

În fine, în a treia, se calculează o bloc linie din C:

$$C_i^T = A_i^T \cdot B = \sum_{j=0}^{n_b-1} A_{ij} \cdot B_j^T, \quad \text{pentru} \quad i \in 0 : n_b - 1. \tag{6.5}$$

Şi în ultimele două forme, produsele de blocuri se calculează cu formula la nivel de element, cu observaţia că acum se înmulţesc matrice (blocuri) dreptunghiulare.

Întorcându-ne la formula (6.2), aceasta arată numărul de operaţii necesare în rezolvarea prin metoda standard a problemei $MM(n)$, anume $T_1(n) = 2n^3$; calculul fiecărui element din C implică $2n$ operaţii (este un produs scalar), şi sunt n^2 elemente. Dacă se calculează în paralel, se observă că toate cele n^3 (câte n pentru cele n^2 elemente din C) produse de tipul $a_{ik}b_{kj}$ care apar se pot calcula simultan, ele depinzând numai de datele de intrare; apoi, pentru obţinerea fiecărui element al rezultatului trebuie calculată o sumă de n valori; această problemă a fost studiată anterior şi se poate rezolva în timp $O(\log n)$, utilizând $O(n^3)$ procesoare (câte $O(n)$ pentru fiecare dintre cele n^2 elemente ale rezultatului); aceasta este şi complexitatea algoritmului paralel standard de înmulţire de matrice.

Este interesant că şi complexitatea problemei $MM(n)$ este $O(\log n)$, dar numărul de procesoare cu care se poate obţine aceasta este mai mic. Nu s-a descoperit încă

limita inferioară a acestui număr de procesoare; se ştie că este mai mare decât $O(n^2)$ – ceea ce se poate demonstra cu uşurinţă – şi mai mică sau egală cu $O(n^{2.37})$, care este cel mai mic majorant descoperit până acum, în 1987 [9]. De remarcat legătura între complexitatea (în timp) a algoritmilor secvenţiali rapizi şi complexitatea "în spaţiu" (număr de procesoare) a algoritmilor paraleli ($O(n^{2.37})$ este cel mai bun timp secvenţial pentru $MM(n)$); este aşa numita *teză a calculului paralel*, vezi [16].

O soluţie facilă de a construi algoritmi pentru produsul de matrice este aceea de a utiliza ca operaţie de bază produsul matrice-vector, operaţie deja discutată. De exemplu, folosind ca bază relaţia (6.4), cu $r = 1$ (deci $n_b = n$), şi ţinând seama că produsele de tipul AB_j se pot calcula în paralel, o descriere simplă este următoarea:

ALGORITM 6.1 ($MM(n)$ paralel, o soluţie generală)
pentru $j = 0 : n-1$, **în paralel**
calculează $C(:, j) \leftarrow A \cdot B(:, j)$

Desigur că pentru acest algoritm mai trebuie detaliată planificarea operaţiilor, descrierea comunicaţiei, etc. O versiune asemănătoare se poate obţine pornind de la relaţia (6.5).

În cele ce urmează, vom descrie algoritmi relativ apropiaţi de această idee, deşi forma va putea părea mult diferită. În fond, aspectele specifice înmulţirii de matrice se referă la gruparea mesajelor comunicate, pentru a se micşora timpii de start-up, şi la alternarea calcule-comunicaţie care este cea mai avantajoasă. Totuşi, vor apare şi idei noi.

6.1.2 Algoritmi pe inel

Vom porni analiza de la ipoteza că datele sunt repartizate echilibrat între procesoare, fără repetiţii, şi, de asemenea, că toate matricele – şi cele de intrare şi cea de ieşire – au aceeaşi repartizare.

O primă soluţie, şi singura pe care o detaliem, este aceea de a lucra pe linii, fiecare procesor deţinând deci $r = n/p$ linii; aşadar, în formulele anterioare, $n_b = p$. Presupunem o repartizare bloc linii, cea mai simplu de manipulat; procesorul P_i va avea în memoria sa blocurile A_i^T, B_i^T (sau, dacă ne referim la blocuri $r \times r$, blocurile A_{ij}, B_{ij}, cu $j = 0 : p-1$); în final, P_i va avea bloc linia corespunzătoare a rezultatului, C_i^T.

Un algoritm simplu este sugerat de prima egalitate din relaţia (6.5), adică $C_i^T = A_i^T \cdot B$. De aici se vede că procesorul P_i are toate elementele din A necesare calculului părţii sale din rezultat, dar are nevoie de întreaga matrice B. O difuzare generală în care fiecare procesor participă cu bloc linia sa din B ar aduce tuturor procesoarelor datele necesare în memoria proprie; acesta este echivalentul algoritmului 4.15. Din păcate, defectul pe care l-am subliniat în cazul produsului matrice-vector este încă şi mai important acum. În loc de $3rn$ elemente (câte o bloc linie din A, B şi C), un procesor ar trebui să aloce spaţiu pentru mai mult de n^2 elemente în memoria locală, ceea ce este foarte mult. De aceea soluţia este, în general, inacceptabilă.

Trebuie deci să imaginăm un mod de comunicaţie prin care bloc liniile matricei B să circule pe la toate procesoarele; desigur, A rămâne nemişcată, ca de altfel şi C. De exemplu, se translează spre stânga bloc liniile din B: la fiecare etapă un procesor trimite către stânga bloc linia sa şi recepţionează o bloc linie din dreapta, aceasta până când bloc liniile revin la procesoarele care le-au deţinut iniţial; evident, în fiecare etapă procesoarele efectuează calcule cu datele locale din etapa respectivă. Se poate observa că paradigma de comunicaţie este cea studiată într-un exemplu din secţiunea **??**; de asemenea, este modul de lucru din algoritmul 4.16.

Vom nota cu l indicele liniilor aparţinând unui procesor, şi cu c indicele liniilor din B primite la un moment dat. În fiecare dintre cele p etape, procesorul P_i calculează un produs de tipul $A_{ij}B_j^T$, ca în termenul din dreapta al relaţiei (6.5), deci, între un bloc local din A (pătrat) şi bloc linia din B recepţionată anterior. În algoritmul ce urmează, instrucţiunile sunt scrise în stil MATLAB; în chip de comentariu este prezentată şi o versiune folosind notaţia din (6.5).

ALGORITM 6.2 ($MM(n)$ pe inel, $n \gg p$, repartizare bloc linii pentru A, B, C; pentru procesorul P_i)

$l \leftarrow ir : (i+1)r - 1$, $c \leftarrow l$, $C(l,:) \leftarrow 0$, $j \leftarrow i$ $\quad \{C_i^T \leftarrow 0\}$
pentru $k = 0 : p-1$
  $C(l,:) \leftarrow C(l,:) + A(l,c)B(c,:)$ $\quad \{C_i^T \leftarrow C_i^T + A_{ij}B_j^T\}$
  în paralel
    send($B(c,:)$, stânga) $\quad \{$**send**(B_j^T, stânga)$\}$
    recv(D, dreapta) $\quad \{$**recv**($B^T_{(j+1) \bmod p}$, dreapta)$\}$
  $j \leftarrow (j+1) \bmod p$
  $c \leftarrow jr : (j+1)r - 1$, $B(c,:) \leftarrow D$

Trebuie remarcat că, atunci când este vorba de o implementare efectivă, $A(l,:)$, $C(l,:)$ şi $B(c,:)$ (ca şi variabila auxiliară D) sunt matrice de dimensiune $r \times n$ în memoria locală, şi nu blocuri ale unor matrice $n \times n$ memorate integral. Algoritmul a fost scris astfel pentru a fi mai sugestiv. Doar aici, pentru exemplificare, vom prezenta şi forma efectivă, apropiată de sursa într-un limbaj de programare; A_{local}, B_{local} şi C_{local} sunt variabilele locale conţinând bloc linii ale matricelor A, B, respectiv C.

ALGORITM 6.3 ($MM(n)$ pe inel, $n \gg p$, pentru procesorul P_i)

$j \leftarrow i$, $C_{local} \leftarrow 0$
pentru $k = 0 : p-1$
  $c \leftarrow jr : (j+1)r - 1$
  $C_{local} \leftarrow C_{local} + A_{local}(:,c)B_{local}$
  în paralel
    send(B_{local}, stânga), **recv**(D, dreapta)
  $j \leftarrow (j+1) \bmod p$, $B_{local} \leftarrow D$

Calculele sunt perfect echilibrate între procesoare; fiecare procesor transmite p mesaje de lungime nr fiecare. Timpul total de execuţie este deci:

$$T_{6.2}(n,p) = \frac{2n^3}{p}\alpha + p\sigma + n^2\beta,$$

iar overhead-ul datorat comunicaţiei: $O_c(n,p) = \frac{p\sigma+n^2\beta}{T_{6.2}(n,p)}$. Acest overhead tinde la zero – iar eficienţa la 1 – pe măsură ce n creşte; deci algoritmul este cu atât mai bun cu cât matricele sunt mai mari.

Un algoritm asemănător se poate obţine adoptând o repartizare bloc coloane, şi făcând să circule blocurile matricei A. Întrucât este simplu, îl lăsăm ca problemă. Timpul de execuţie va fi identic cu cel de mai sus.

6.1.3 Algoritmi pe tor

Pornim din nou de la o repartizare uniformă a matricelor; procesorul P_{ij} are în memoria locală blocurile matricelor de intrare A_{ij}, B_{ij}, şi va calcula blocul C_{ij} al rezultatului; formula de calcul este cea din (6.3) şi, presupunând torul de dimensiuni egale, dimensiunea unui bloc este $r = n/\sqrt{p}$ (deci $n_b = \sqrt{p}$, în formula amintită).

Rezultatul C poate rămâne local procesoarelor, neexistând nici o dependenţă între blocuri diferite ale rezultatului; se observă însă că, spre deosebire de cazul topologiei de inel, trebuie să circule acum şi A şi B, blocurile locale ale nici uneia dintre aceste matrice nefiind suficiente pentru calculul din (6.3). Din această formulă se mai observă că blocurile din A de care are nevoie P_{ij} se află pe linia i a torului, iar blocurile necesare din B pe coloana j, adică pe linia şi coloana pe care se află procesorul. De aici rezultă că matricea A va circula pe orizontală (pe liniile torului), iar B pe verticală (pe coloanele torului).

Să presupunem că blocurile matricei B sunt transmise, la fiecare etapă, în sus de către toate procesoarele, până când revin la procesoarele care le-au deţinut iniţial; sunt deci $\sqrt{p}$ etape; în acest fel, fiecare procesor va avea acces la o bloc coloană a matricei B, ceea ce va fi suficient pentru a efectua calculele, din punctul de vedere al lui B.

Cum va circula A ? La un moment dat, cum o linie de procesoare deţine o bloc linie a matricei B, pentru formarea unor produse care să contribuie la formarea unei bloc linii a rezultatului este necesar un singur bloc al matricei A – deţinut de un singur procesor; de exemplu, în poziţia iniţială, bloc linia B_{00}, B_{01}, ..., $B_{0,\sqrt{p}-1}$ trebuie înmulţită cu blocul A_{00}; este vorba despre, în relaţia (6.5), primul termen al sumei, cel pentru $i = 0$. Apoi, după ce blocurile matricei B sunt comunicate vecinilor de sus (practic matricea se translează în sus), pe linia 0 de procesoare se va găsi acum bloc linia B_{10}, B_{11}, ..., $B_{1,\sqrt{p}-1}$; ea va trebui înmulţită cu blocul A_{01}.

Se observă deci că matricea A poate păstra repartizarea iniţială (nu este necesar să circule toată, ca B) dar, la fiecare etapă, câte un bloc al lui A va trebui difuzat pe fiecare linie de procesoare de către deţinătorul său. Modul de comunicaţie este exemplificat în figura 6.1, pentru un tor cu 3×3 procesoare. În fiecare căsuţă se află

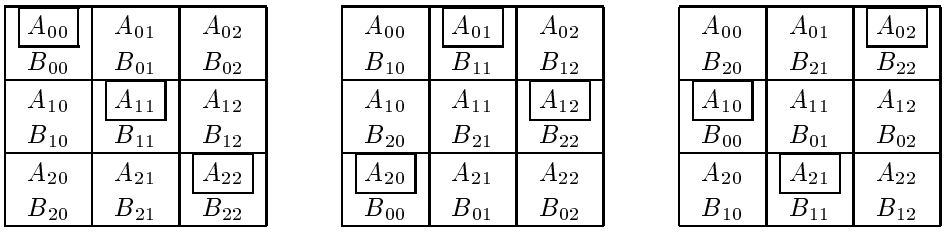


Figura 6.1: Comunicaţia în algoritmul pentru $MM(n)$ pe tor, cu difuzare pe linii a unor blocuri ale matricei A şi cu translare pe verticală a matricei B.

blocurile locale procesorului din poziţia corespondentă, la fiecare etapă a algoritmului, începând cu repartizarea iniţială. Blocul din A aflat într-un mic chenar este cel difuzat pe linia procesorului respectiv în etapa curentă. După $\sqrt{p}$ etape (3, în cazul nostru), se obţine din nou repartizarea iniţială.

Putem descrie adresa procesorului care difuzează la un moment dat, pe linia i; în etapa k, cu $k = 0 : \sqrt{p} - 1$, difuzează procesorul din coloana j pentru care $(i - j + k) \bmod p = 0$; altfel spus, în prima etapă difuzează procesorul de pe diagonală, în a doua cel din dreapta sa, apoi următorul din dreapta, etc. Difuzarea pe o linie a torului este o operaţie banală, o linie de procesoare în tor fiind un inel. Vom nota cu c indicele de coloană al procesorului care face difuzarea: $c = (i + k) \bmod p$. Prezentăm algoritmul doar în notaţie la nivel de bloc; toate variabilele sunt blocuri $r \times r$.

ALGORITM 6.4 ($MM(n)$ pe tor, algoritm cu difuzare pe linii a matricei A, $n \gg p$, pentru procesorul P_{ij})

$c \leftarrow i$, $C_{ij} \leftarrow 0$
pentru $k = 0 : \sqrt{p} - 1$
(1) **difuzare (parţială)**: P_{ic} difuzează A_{ic} pe linia i a torului
$C_{ij} \leftarrow C_{ij} + A_{ic}B_{cj}$
în paralel
send(B_{cj}, sus)
recv($B_{(c+1) \bmod p, j}$, jos)
$c \leftarrow (c + 1) \bmod p$

Pentru claritate, să explicăm mai mult instrucţiunea (1), din punctul de vedere al procesorului P_{ij}; în fiecare etapă k, pe linia i se face o difuzare; dacă $j = c$, atunci P_{ij} este procesorul care iniţiază difuzarea; dacă $j \neq c$, atunci P_{ij} este unul dintre destinatarii difuzării, el recepţionând blocul A_{ic} de la un vecin şi, eventual, transmiţându-l celuilalt, conform algoritmului de difuzare pe inel folosit.

Fiecare procesor are nevoie de cinci variabile de dimensiune $r \times r$: trei pentru C_{ij}, A_{ij} şi blocul curent din B, plus două variabile auxiliare (care pot fi identice, de fapt), una pentru blocul difuzat din A, alta pentru recepţia următorului bloc din B.

A_{00}	A_{01}	A_{02}
B_{00}	B_{11}	B_{22}
A_{11}	A_{12}	A_{10}
B_{10}	B_{21}	B_{02}
A_{22}	A_{20}	A_{21}
B_{20}	B_{01}	B_{12}

A_{01}	A_{02}	A_{00}
B_{10}	B_{21}	B_{02}
A_{12}	A_{10}	A_{11}
B_{20}	B_{01}	B_{12}
A_{20}	A_{21}	A_{22}
B_{00}	B_{11}	B_{22}

A_{02}	A_{00}	A_{01}
B_{20}	B_{01}	B_{12}
A_{10}	A_{11}	A_{12}
B_{00}	B_{11}	B_{22}
A_{21}	A_{22}	A_{20}
B_{10}	B_{21}	B_{02}

Figura 6.2: Comunicaţia în algoritmul de înmulţire de matrice pe tor, cu preskewing.

Ca şi la inel, calculele sunt echilibrat repartizate între procesoare. În ce priveşte timpul necesar comunicaţiei, evaluarea este ceva mai dificilă deoarece difuzarea este o operaţie a cărei eficienţă depinde de algoritmul folosit şi de caracteristicile fizice al calculatorului ţintă. Considerând cel mai bun timp de difuzare pe inel (fără însă a face pipeline), adică $(\sqrt{p}/2)(\sigma + (n^2/p)\beta)$, timpul total va fi

$$T_{6.4} = \frac{2n^3}{p}\alpha + \sqrt{p}\left(\sigma + \frac{n^2}{p}\beta\right) + \sqrt{p}\frac{\sqrt{p}}{2}\left(\sigma + \frac{n^2}{p}\beta\right),$$

ultimul termen reprezentând timpul de difuzare, iar cel din mijloc cel pentru comunicarea matricei B (reamintim că un bloc B_{ij} are n^2/p elemente). După cum se vede, timpul de comunicaţie este $O(n^2)$, comparabil cu cel pentru inel. El poate fi redus prin optimizarea difuzării, dar nu va scădea sub, în cel mai bun caz, $O(2n^2/\sqrt{p})$, presupunând că difuzarea unui bloc durează tot atât cât comunicarea blocului între doi vecini. O ultimă îmbunătăţire este executarea în paralel a difuzării şi a translării matricei B, în fiecare etapă, deoarece ele decurg pe direcţii diferite; vezi problema 6.1.2. Aceasta reduce timpul de comunicaţie la cel necesar difuzării.

Un alt algoritm pe tor provenind din metoda standard pentru rezolvarea $MM(n)$ se obţine impunând o circulaţie asemănătoare pentru matricele A şi B, dar pe direcţii diferite – A spre stânga, B în sus (aceasta din urmă la fel ca în algoritmul 6.4); în acest mod se va reduce cu siguranţă comunicaţia. Rămâne de văzut dacă această idee este valabilă şi, dacă da, de rezolvat problema organizării întâlnirilor corespunzătoare între blocurile din A şi din B, conform relaţiei (6.3), astfel încât blocurile rezultatului să se formeze local; mai exact, să se găsească o dispunere iniţială a matricelor A şi B astfel încât, în cele $\sqrt{p}$ etape de comunicaţie, în P_{ij} să se întâlnească, pe rând, toate blocurile A_{ik}, B_{kj}, cu $k \in 0 : \sqrt{p} - 1$; evident, ordinea întâlnirilor nu are importanţă.

Se vede imediat că repartizarea "obişnuită", cu P_{ij} posedând A_{ij} şi B_{ij}, nu este de folos; termenul $A_{ij}B_{ij}$ nu apare, în general, în sumele din (6.3). Dispunerea iniţială a blocurilor matricelor A şi B este cea prezentată în figura 6.2, în stânga, şi a fost descoperită de Cannon (e cunoscută cu numele de "preskewing" – repartizare "oblică").

Această repartizare poate fi dedusă relativ simplu, o dată stabilit sensul de mişcare

pentru A şi B – stânga, respectiv sus (ideea algoritmului însă, nu e deloc banală !). Se începe prin a completa prima linie de procesoare cu prima bloc linie din A, în modul obişnuit: A_{0j} este local lui P_{0j}. Pe aceeaşi linie, blocurile din B sunt hotărâte de faptul că P_{0j} calculează C_{0j}, deci are nevoie de B_{jj} pentru a-l înmulţi cu A_{0j}. Argumentăm în continuare doar pentru P_{10}, pentru celelalte procesoare procedându-se analog, din aproape în aproape; în a doua etapă, deoarece A se deplasează spre stânga, în P_{00} va ajunge A_{01}; pentru a calcula C_{00}, acest bloc trebuie înmulţit cu B_{10}; deci B_{10} este deţinut iniţial de P_{10}, care l-a trimis în sus în a doua etapă; tot iniţial, în P_{10} se află A_{11}, singurul bloc care se înmulţeşte cu B_{10} pentru a contribui la C_{10}.

O dată completată dispunerea iniţială, algoritmul este uşor de verificat, ceea ce este ilustrat şi în figura 6.2, în care sunt prezentate cele trei etape ale înmulţirii pe un tor 3×3. Într-o concluzie succintă, în repartizarea "oblică" procesorul P_{ij} are iniţial blocurile $A_{i,(j+i) \bmod \sqrt{p}}$ şi $B_{(i+j) \bmod \sqrt{p},j}$.

Dată această repartizare iniţială, algoritmul de rezolvare a problemei $MM(n)$ are o structură foarte simplă. Notând cu indicele *local* cele trei blocuri din A, B, C aflate la un moment dat în posesia unui procesor, şi cu D şi E două variabile auxiliare de aceeaşi dimensiune $(r \times r)$, algoritmul va fi:

ALGORITM 6.5 ($MM(n)$ pe tor, algoritm cu preskewing, $n \gg p$, pentru proc. P_{ij})

$C_{ij} \leftarrow 0$
pentru $k = 0 : \sqrt{p} - 1$
  $C_{local} \leftarrow C_{local} + A_{local} \cdot B_{local}$
  în paralel
    send(A_{local}, stânga)
    recv(D, dreapta)
    send(B_{local}, sus)
    recv(E, jos)
  $A_{local} \leftarrow D$, $B_{local} \leftarrow E$

În această formă, timpul de execuţie este următorul:

$$T_{6.5} = \frac{2n^3}{p}\alpha + \sqrt{p}(\sigma + \frac{n^2}{p}\beta).$$

Se observă dispariţia întregului termen ce reflecta complexitatea difuzării, ceea ce conduce la un timp de comunicaţie de $O(n^2/\sqrt{p})$, mai mic decât cel corespunzător din algoritmul 6.4.

Nu trebuie să uităm însă că, de cele mai multe ori, matricele sunt disponibile în repartizarea prezentată la începutul secţiunii, adică procesorul P_{ij} are blocurile A_{ij}, B_{ij}. La dispunerea din figura 6.2 se poate ajunge uşor din această poziţie prin translări, pe linii pentru A şi pe coloane pentru B. Deoarece la procesorul P_{ij} trebuie să ajungă blocul $A_{i,(j+i) \bmod \sqrt{p}}$, se observă că sunt necesare i translări la stânga ale blocurilor din A, de pe linia i, deoarece coloana $(j + i) \bmod \sqrt{p}$ este cu i poziţii la

dreapta (pe inelul format de linia i a torului) faţă de coloana j. În mod analog, pentru ca $B_{(i+j) \bmod \sqrt{p},j}$ să ajungă în posesia procesorului P_{ij}, sunt necesare j translări în sus ale blocurilor din B, pe coloana j a torului. Un algoritm pentru preskewing este deci următorul (am omis variabilele auxiliare necesare ca tampon la recepţie):

ALGORITM 6.6 (preskewing pe tor, $n \gg p$, pentru procesorul P_{ij})

în paralel
  pentru $k = 0 : i - 1$
    în paralel
      send(A_{local}, stânga)
      recv(A_{local}, dreapta)
  pentru $k = 0 : j - 1$
    în paralel
      send(B_{local}, sus)
      recv(B_{local}, jos)

Timpul de execuţie al algoritmului 6.6 este apropiat de cel necesar unei singure difuzări pe un inel; ori, în algoritmul 6.4 se efectuau $\sqrt{p}$ difuzări pe fiecare linie de procesoare; de unde o complexitate a comunicaţiilor mai bună pentru algoritmii 6.5 şi 6.6, luaţi împreună. Algoritmul 6.6 se poate îmbunătăţi, dacă translarea se face în sensul drumului cel mai scurt; de exemplu, pe ultima linie, în loc de $\sqrt{p} - 1$ translări la stânga, e suficientă una singură la dreapta; astfel se reduce la jumătate timpul de comunicaţie; vezi problema 6.1.3.

Pentru ambii algoritmi prezentaţi, deci indiferent de repartizarea iniţială a matricelor, eficienţa tinde asimptotic la 1 deoarece, pentru n mare, termenul reprezentând complexitatea comunicaţiilor este de $O(n^2)$, iar cel reprezentând calculele de $O(n^3)$.

6.1.4 Un algoritm pentru memorie partajată

Presupunem acum o arhitectură MIMD cu memorie partajată, cu p procesoare, fiecare având o mică memorie locală; cu minime modificări, algoritmul este valabil şi în lipsa memoriei locale. Vom folosi ideea algoritmului 6.2 în ce priveşte modul de desfăşurare a calculelor şi repartizarea matricelor pe procesoare, cu deosebirea că acum matricele se află în memoria comună, şi nu în cele locale, ceea ce simplifică sarcina.

Aşadar, procesorul P_i se ocupă de o bloc linie din produs, pe care o va calcula pe baza relaţiei (6.5), în care $r = n/p$ şi $n_b = p$: $C_i^T = A_i^T \cdot B$. În primul rând, P_i va transfera în memoria sa locală bloc linia A_i^T, care, aşa cum se vede, nu mai trebuie cunoscută de alte procesoare. În schimb matricea B trebuie citită în întregime de toate procesoarele, ceea ce pune două probleme: întâi, memoria locală este mică, şi deci, în general, nu ajunge pentru stocarea matricei B; apoi, dacă toate procesoarele citesc simultan aceleaşi elemente din matricea B, pot exista conflicte la citire care vor întârzia calculele efective. Soluţia este sugerată de algoritmul 6.2: fiecare procesor

să acceseze câte o bloc linie din B la un moment dat, astfel încât, în p etape, un procesor să acceseze în întregime matricea B. Algoritmul care urmează este deci o simplă adaptare a algoritmului 6.2, al cărei aspect este apropiat de varianta 6.3.

ALGORITM 6.7 ($MM(n)$ pe arhitectură cu memorie comună, $n \gg p$, pt. proc. P_i)

$j \leftarrow i$, $c \leftarrow ir : (i+1)r - 1$, $C_{local} \leftarrow 0$
get$(A(c,:), A_{local})$ {citeşte A_i^T}
pentru $k = 0 : p-1$
  get$(B(c,:), B_{local})$ {citeşte B_j^T}
  $C_{local} \leftarrow C_{local} + A_{local}(:,c)B_{local}$ {$C_i^T \leftarrow C_i^T + A_{ij}B_j^T$}
  $j \leftarrow (j+1) \bmod p$, $c \leftarrow jr : (j+1)r - 1$
put$(C_{local}, C(ir : (i+1)r - 1, :))$ {scrie C_i^T}

Variabilele cu indicele *local* sunt matrice $r \times n$ în memoria locală a fiecărui procesor. A_{local} conţine permanent bloc linia A_i^T; în B_{local} se transferă, din memoria comună, bloc linii din B; în C_{local} se calculează bloc linia C_i^T a rezultatului, care este transferată, în final, în memoria comună, în poziţia corespunzătoare; nu se pune problema nici unui conflict de scriere. Deşi am presupus că după fiecare iteraţie dintr-o buclă **pentru** se face sincronizare, aici această ipoteză nu este neapărat necesară; e drept, nesincronizările pot introduce întârzieri, dar nu afectează corectitudinea algoritmului.

Algoritmul este echilibrat din punctul de vedere al încărcării; se apelează de $p+2$ ori funcţia **get** şi o dată funcţia **put**. Timpul de execuţie al algoritmului va fi deci:

$$T_{6.7} = \frac{2n^3}{p}\alpha + (p+2)(\sigma + \frac{n^2}{p}\beta).$$

Şi pentru acest algoritm eficienţa tinde asimptotic la 1.

Algoritmul 6.7 poate avea un singur inconvenient – insuficientă memorie locală pentru stocarea simultană a câte unei bloc coloane din A, B şi C, ceea ce presupune spaţiu pentru $3n^2/p$ elemente în virgulă mobilă. În acest caz se poate adopta, de exemplu, o partiţionare bloc ciclică pe linii a matricelor; să spunem că fiecare procesor se ocupă de q bloc linii, fiecare bloc linie având r linii, şi deci acum $n = rqp$. Fiecare procesor va calcula, pe rând, printr-un algoritm identic cu 6.7, fiecare dintre cele q bloc linii ce-i revin din C. Pentru fiecare bloc linie, trebuie citită întreaga matrice B, deci timpul de execuţie al algoritmului va fi

$$T'_{6.7} = \frac{2n^3}{p}\alpha + q(qp+2)(\sigma + \frac{n^2}{qp}\beta),$$

adică o comunicaţie de aproximativ q ori mai costisitoare decât în cazul precedent.

Probleme

P 6.1.1 Scrieţi un algoritm pentru $MM(n)$ pe inel, pornind de la ipoteza unei repartizări bloc coloane pentru matricele A, B, C.

P 6.1.2 Modificaţi algoritmul 6.4, de înmulţire de matrice pe tor, astfel încât difuzarea unui bloc din A şi comunicarea blocurilor din B vecinilor de sus să decurgă în paralel.

P 6.1.3 Modificaţi algoritmul 6.6 a. î. translările matricelor A şi B să se facă întotdeuna pe drumul cel mai scurt.

P 6.1.4 Scrieţi un algoritm de rezolvare a $MM(n)$ pe arhitecturi cu memorie comună, pornind de la egalitatea $C_{ij} = A_i^T \cdot B_j$. De ce n-a putut fi utilizată această egalitate în cazul inelului ?

P 6.1.5 Scrieţi un algoritm pentru $MM(n)$ pe inel, presupunând A repartizată bloc linii, iar B bloc coloane.

6.2 Sisteme liniare triunghiulare

Ne vom ocupa în această secţiune de problema rezolvării sistemelor liniare de forma $Ax = b$, cu $A \in \mathbf{R}^{n \times n}$, inferior triunghiulară şi $b \in \mathbf{R}^n$; deci, a_{ij}=0, pentru $i < j$; presupunem că problema are soluţie unică, adică toate elementele diagonale a_{ii} sunt nenule. Deşi această problemă, pe care o vom nota $TR(n)$, este banală în tratarea secvenţială şi are o complexitate de numai $O(n^2)$ – faţă de $O(n^3)$, pentru înmulţirea de matrice sau rezolvarea de sisteme dense – se impune studierea ei cu atenţie în cazul implementării pe o arhitectură paralelă, deoarece, aşa cum se va vedea, sunt necesare multe comunicaţii, care, dacă nu sunt bine organizate, pot scădea mult eficienţa; de altfel, dacă pentru alţi algoritmi de calcul numeric eficienţe de 80-90% sunt obişnuite, pentru rezolvarea sistemelor triunghiulare 50% este deja o bună performanţă. De asemenea, rezolvarea de sisteme triunghiulare constituie etapa finală în rezolvarea sistemelor liniare dense, de unde şi un interes practic deosebit.

După ce vom prezenta pe scurt considerente legate de rezolvarea secvenţială a problemei, de complexitatea secvenţială şi de cea paralelă, vor fi discutaţi şi evaluaţi mai mulţi algoritmi paraleli pentru $TR(n)$. Ei vor fi grupaţi în perechi; algoritmii din aceeaşi pereche sunt duali, fiecare fiind inspirat dintr-una din cele două forme secvenţiale clasice. Toţi algoritmii sunt adecvaţi arhitecturilor cu memorie distribuită şi va fi considerat doar cazul practic $n \gg p$. Referinţele bibliografice cele mai folosite în alcătuirea acestei secţiuni au fost Heath & Romine [14], Chamberlain [7] şi Li & Coleman [20, 21].

6.2.1 Algoritmi secvenţiali

Există doi algoritmi consacraţi pentru rezolvarea de sisteme triunghiulare. Ei se deosebesc numai prin ordinea operaţiilor, având aceeaşi idee de bază – substituţia înainte – şi aceeaşi complexitate. Din linia i, cu $i \in 0 : n-1$, a ecuaţiei $Ax = b$ se deduce imediat că:

$$x_i = \left(b_i - \sum_{j=0}^{i-1} a_{ij} x_j \right) / a_{ii}, \tag{6.6}$$

ceea ce arată că elementele din x se pot calcula în ordine crescătoare a indicilor, fiecare x_i depinzând numai de x_j cu $j < i$; astfel, în momentul calculării lui x_i, în relaţia de mai sus, toţi termenii din dreapta sunt cunoscuţi.

Să presupunem că x este iniţializat cu elementele din b. În primul algoritm, după calcularea unui element x_j al soluţiei, se actualizează cu ajutorul acestuia toate elementele x_i, cu $i > j$, adică elementele încă necalculate din x. Practic, se adună vectorul x cu un multiplu al coloanei j a matricei A – de aceea această formă este numită algoritmul cu *sumă vectorială*:

ALGORITM 6.8 ($TR(n)$ secvenţial, prin sumă vectorială)

$x \leftarrow b$
pentru $j = 0 : n-1$
  $x_j \leftarrow x_j / a_{jj}$
  pentru $i = j+1 : n-1$
    $x_i \leftarrow x_i - a_{ij} x_j$

De notat că bucla **pentru** i poate fi înlocuită prin suma vectorială $x(j+1 : n-1) \leftarrow x(j+1 : n-1) - A(j+1 : n-1, j)x(j)$.

În a doua formă, o necunoscută se calculează folosind toate elementele deja calculate ale soluţiei, adică aplicând formula (6.6) ca atare. Pentru a realiza acest calcul se efectuează produsul scalar între linia i a matricei A şi vectorul x, mai exact $A(i, 0 : i-1) \cdot x(0 : i-1)$ (care poate înlocui bucla **pentru** j de mai jos) – de aceea această formă este numită algoritmul cu *produs scalar*:

ALGORITM 6.9 ($TR(n)$ secvenţial, prin produs scalar)

$x \leftarrow b$
pentru $i = 0 : n-1$
  pentru $j = 0 : i-1$ $\{x(i) \leftarrow A(i, 0 : i-1) \cdot x(0 : i-1)\}$
    $x_i \leftarrow x_i - a_{ij} x_j$
  $x_i \leftarrow x_i / a_{ii}$

Numărul de operaţii, în ambii algoritmi, este n^2.

6.2.2 Complexitatea paralelă

Pentru ambii algoritmi, cu sumă vectorială sau cu produs scalar, graful de precedenţă este acelaşi, şi anume cel din figura 6.3, în care s-au folosit notaţiile: T_{ij} pentru operaţia $x_i \leftarrow x_i - a_{ij} x_j$, şi S_i pentru operaţia $x_i \leftarrow x_i / a_{ii}$. Relaţiile de precedenţă sunt $S_i \prec T_{ki}$ şi $T_{ij} \prec S_i$.

În primul algoritm, ordinea de execuţie este pe coloane, adică S_0, T_{10}, T_{20}, T_{30}, S_1, T_{21}, etc., în timp ce în al doilea ordinea este pe linii, adică S_0, T_{10}, S_1, T_{20}, T_{21}, etc.

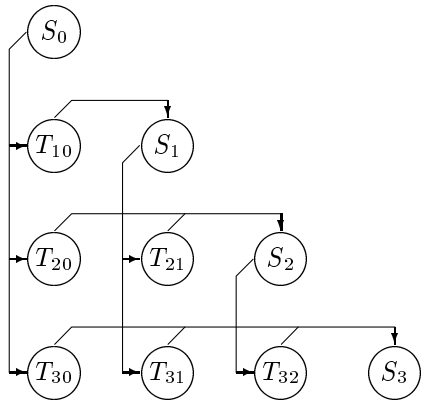


Figura 6.3: Graf de dependenţă pentru rezolvarea unui sistem inferior triunghiular cu $n = 4$.

Se observă că înălţimea grafului este dată de calea S_0, T_{10}, S_1, T_{21}, ..., S_{n-1}, a cărei lungime este de $n + 2(n-1) = 3n - 2$ operaţii. Lăţimea grafului este de $n - 1$. Deci complexitatea paralelă a acestor doi algoritmi este $O(n)$, folosind $O(n)$ procesoare.

Dar complexitatea paralelă a problemei $TR(n)$ este mai bună, după cum se arată, de exemplu, în [4]. Se foloseşte următorul rezultat:

LEMĂ. Dacă A are diagonala unitate ($a_{ii} = 1$, ceea ce nu e o restricţie; dacă elementele diagonale sunt diferite de 1, atunci se împarte cu ele întreaga linie, inclusiv elementul corespunzător din b), şi descompunem $A = I - L$, cu L strict inferior triunghiulară (adică $l_{ij} = 0$, pentru $i \leq j$), atunci:

$$A^{-1} = I + L + L^2 + \ldots + L^{n-1}.$$

Demonstraţie (schiţă). Notând cu B termenul din dreapta al egalităţii de mai sus, se observă că $BA = I - L^n$. Se poate însă observa că L^2 are prima subdiagonală egală cu zero, L^3 are primele două subdiagonale zero, etc.; în concluzie $L^n = 0$. □

Dar, după cum am văzut, o sumă se poate calcula cu $O(\log n)$ operaţii; calculul matricilor L, L^2, ..., L^{n-1} este o problemă de tip prefix sum, şi are complexitatea $O(\log^2 n)$, deoarece o înmulţire de matrici se poate calcula în $O(\log n)$, ceea ce se ştie deja, iar calculul produselor parţiale comportă $O(\log n)$ etape la rândul său (vezi secţiunea dedicată sumei prefixelor); în fine, produsul $A^{-1}b$ se poate calcula în $O(\log n)$.

Soluţia sistemului triunghiular se poate calcula aşadar în timp $O(\log^2 n)$, cu $O(n^4)$ procesoare, adică de $O(n)$ ori (cât e nevoie la suma prefixelor) câte $O(n^3)$ (cât e nevoie la înmulţirea de matrice). Se pot obţine şi alte soluţii cu aceeaşi complexitate, dar folosind $O(n^3)$ procesoare; ele pot fi găsite în [4].

În orice caz, acest algoritm optim din punct de vedere al timpului de execuţie foloseşte un număr execesiv de mare de procesoare (deci este mai puţin eficient) şi, în plus, este mai puţin stabil numeric. El este bun doar pentru determinarea teoretică a complexităţii problemei. Din punct de vedere practic, implementările curente se bazează pe algoritmii de tip substituţie înainte.

6.2.3 Algoritmi cu comunicaţie globală

Pentru proiectarea unui algoritm paralel este importantă precizarea datelor care se folosesc la un moment dat (în bucla interioară a celor doi algoritmi secvenţiali, în cazul nostru), precum şi a datelor de intrare de care depinde un element al soluţiei. În ambii algoritmi, o necunoscută x_i depinde de elementul b_i corespunzător, dar această dependenţă nu implică decât iniţializarea $x \leftarrow b$, deci nu are prea mare importanţă. În bucla interioară a algoritmului cu sumă vectorială, pentru actualizarea elementelor x_i, cu $i > j$, e necesară coloana j a matricei A şi soluţia x_j (proaspăt calculată). În algoritmul cu produs scalar, pentru calculul soluţiei x_i se folosesc linia i a matricei A şi soluţiile x_j, cu $j < i$.

Presupunem o repartizare pe linii sau pe coloane (cei doi algoritmi paraleli de un anume tip, rezultaţi din cei doi algoritmi secvenţiali de bază, vor folosi repartizări diferite), fără a detalia însă, deocamdată, modul de repartizare; considerăm doar că fiecare procesor are $m = n/p$ linii (sau coloane) ale matricei A. Vom nota cu $P(k)$ adresa procesorului care deţine linia (sau coloana, după caz) k. De asemenea, pentru un procesor oarecare, vom denumi mulţimea liniilor (coloanelor) care sunt locale acestuia *linii (coloane) proprii*. Elementele termenului liber b se repartizează la fel, indiferent dacă A se repartizează pe linii sau pe coloane, deoarece este vorba despre un vector; în orice caz, fiecare procesor va avea m elemente din b şi e natural să stabilim că va poseda, în final, elementele corespunzătoare din x; deci, $P(i)$ deţine b_i şi calculează x_i.

Cea mai simplă modalitate de a paraleliza algoritmii secvenţiali prezentaţi este de a distribui procesoarelor calculele din bucla cea mai interioară, care se pot desfăşura în paralel. În algoritmul cu sumă vectorială, pentru un anume j, acestea sunt operaţiile T_{ij}, cu $i = j + 1 : n - 1$, adică cele de pe o "coloană" a grafului de precedenţă din figura 6.3. Ele necesită coloana j a matricei A; pentru a se executa calculele folosind doar date locale procesoarelor, coloana respectivă trebuie împărţită între procesoare; de aici concluzia că repartizarea naturală a matricei A este pe linii. Pe de altă parte, în toate calculele de tip T_{ij} apare x_j, care nu poate fi calculat decât de un singur procesor; pentru a comunica un x_j tuturor procesoarelor e deci necesară o difuzare. Prezentăm mai jos algoritmul, revenind după aceea cu alte explicaţii. Vom presupune, acum şi în tot restul secţiunii dedicate rezolvării paralele a $TR(n)$, că vectorul x este iniţializat cu b.

ALGORITM 6.10 ($TR(n)$ cu difuzare, pe arhitectură cu memorie distribuită, A repartizată pe linii)

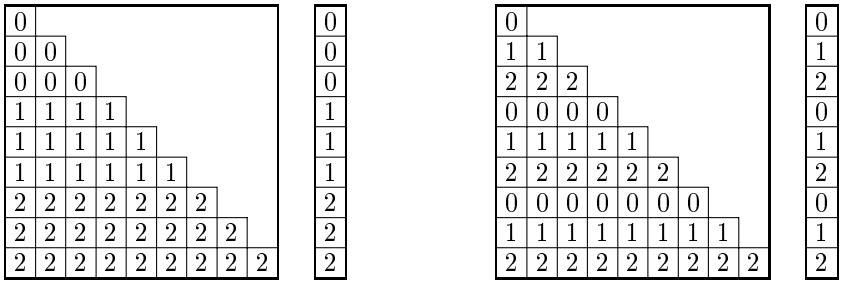


Figura 6.4: Repartizarea matricei inferior triunghiulare A şi a vectorului x, pentru $n = 9$ şi $p = 3$, în cazurile: (a) bloc linii; (b) ciclic pe linii. Fiecare pătrat mic simbolizează un element, iar numărul din interiorul acestuia reprezintă adresa procesorului care deţine elementul.

pentru $j = 0 : n-1$
  dacă id $= P(j)$ **atunci**
(1) $x_j \leftarrow x_j / a_{jj}$
(2) **difuzare**: $P(j)$ trimite x_j celorlalte procesoare
  pentru $i = j+1 : n-1$
    dacă id $= P(i)$ **atunci**
(3) $x_i \leftarrow x_i - a_{ij} x_j$

Prima parte a algoritmului, calculul unei soluţii x_j, este efectuată de procesorul care deţine iniţial b_j, adică $P(j)$, în timp ce celelalte procesoare sunt inactive. Apoi, soluţia x_j este comunicată de către $P(j)$ tuturor celorlalte procesoare, într-o operaţie de difuzare. În fine, în (3), fiecare procesor actualizează elementele proprii x_i, cu $i > j$, deci cele încă necalculate ale soluţiei, folosind valoarea x_j recepţionată la difuzare şi elementele a_{ij} de pe liniile care îi sunt locale.

Revenim acum la problema modului de repartizare pe linii. Dacă presupunem o repartizare bloc linii, cum este cea din figura 6.4a, atunci procesorul P_0, care va deţine liniile de la 0 la $m-1$, îşi va termina complet treaba după primele m iteraţii, când va calcula elementele $x(0 : m-1)$, adică exact cele de care se ocupă. Pentru $j \geq m$, calculele vor fi efectuate doar de $p-1$ procesoare. În mod analog, după încă m iteraţii iese din cursă şi P_1, etc. Este clar că nu se poate spera o eficienţă prea bună; chiar dacă instructiunile (3) sunt executate în paralel, doar o parte dintre procesoare sunt active; în figură, pe o coloană a matricei, numerele reprezintă adresele procesoarelor care execută în paralel operaţii de tip (3), pentru un j fixat. În plus, deşi fiecare procesor deţine m linii, numărul de elemente nenule este diferit pentru fiecare, matricea A fiind triunghiulară; de exemplu, P_0 are $m(m+1)/2$ elemente,

în timp ce P_{p-1} are aproximativ mn, adică mult mai multe; deci, repartizarea, deşi aparent echitabilă, nu este deloc aşa.

Este de dorit, aşadar, ca fiecare procesor să aibă aproximativ acelaşi număr de elemente (nenule) din *fiecare* coloană a matricei A, pentru a contribui în aceeaşi măsură la calculele de tip (3). Aceasta se poate realiza simplu printr-o repartizare ciclică pe linii, cum este cea din figura 6.4b; deci, $P(j) = j \bmod p$. Mai mult, spre deosebire de cazul anterior, acum fiecare procesor are şi aproximativ acelaşi număr de elemente din A.

Pentru a evalua timpul de calcul al algoritmului 6.10, considerăm timpii pentru cele trei faze descrise:

- $n\alpha$ – timpul pentru calculul (secvenţial) al soluţiilor x_j de către fiecare procesor.
- $\approx c_d n(\sigma + \beta)$ – timpul pentru cele $n-1$ operaţii de difuzare (pentru $j = n-1$ nu mai este necesară difuzarea); coeficientul c_d depinde de topologia arhitecturii şi de algoritmul de difuzare ales; de exemplu, pentru hipercub $c_d = \log p$, iar pentru inel $c_d = p/2$.
- pentru actualizări (paralelism complet): $\frac{1}{p}\sum_{j=0}^{n-1} 2(n-1-j)\alpha \approx \frac{1}{p}(n^2-n)\alpha$.

Timpul total este suma celor de mai sus:

$$T_{6.10} = \frac{1}{p}(n^2 + np - n)\alpha + c_d(n-1)(\sigma + \beta).$$

Precizăm că aceasta este o limită superioară a timpului de calcul; practic, pot apare valori mai mici, deoarece în timpul difuzării un procesor nu este permanent ocupat; deci, de îndată ce termină partea sa din operaţia de difuzare, el poate trece imediat la actualizări.

Pentru algoritmul cu produs scalar, executarea în paralel a operaţiilor din bucla **pentru** cea mai interioară, înseamnă cooperarea procesoarelor pentru calculul sumei $\sum_{j=0}^{i-1} a_{ij}x_j$; deci, pentru un i fixat, suma implică elemente de pe linia i a matricei A, şi elementele din x deja calculate. Pentru a repartiza echilibrat calculele, e normal să presupunem linia i împărţită aproximativ egal între procesoare, ceea ce semnifică o repartizare pe coloane a matricei A, şi anume, din motive asemănătoare celor de la algoritmul precedent, ciclic pe coloane; repartizarea lui x nu pune probleme suplimentare, deoarece procesorul care deţine a_{ij}, adică $P(j)$, deţine şi x_j. Deci, fiecare procesor calculează $\sum_{j<i} a_{ij}x_j$, cu j parcurgând coloanele proprii. Pentru a obţine x_i ca în formula 6.6, este necesară adunarea acestor sume parţiale, după cum se procedează în algoritmul următor:

ALGORITM 6.11 ($TR(n)$ cu colectare, pe arhitectură cu memorie distribuită, A repartizată ciclic pe coloane)

pentru $i = 0 : n-1$
  $t \leftarrow 0$
  pentru $j = 0 : i-1$
    dacă id $= P(j)$ **atunci**
(1) $t \leftarrow t + a_{ij}x_j$
(2) $s \leftarrow$ **colectare cu sumare** a valorilor t în $P(i)$
  dacă id $= P(i)$ **atunci**
(3) $x_i \leftarrow (b_i - s)/a_{ii}$

Pentru un i fixat, calculele de tip (1) decurg în perfect paralelism; este vorba de operaţiile T_{ij} de pe "linia" i a grafului de dependenţă din figura 6.3, care sunt independente; datorită repartizării ciclice, pentru orice i, toate procesoarele au aproximativ acelaşi număr de operaţii de efectuat. Toate sumele parţiale trebuie adunate, iar rezultatul comunicat procesorului $P(i)$; aceasta se poate realiza eficient printr-o operaţie de colectare cu sumare, aşa cum am văzut în algoritmul 4.3, pentru hipercub, sau în problema 4.3.7, pentru inel. În sfârşit, în instrucţiunea (3), procesorul $P(i)$ calculează soluţia x_i, timp în care celelalte procesoare sunt inactive (în realitate, celelalte procesoare continuă cu calculele (1) din iteraţia următoare, dar vor fi nevoite să-l aştepte pe $P(i)$ la următoarea colectare; până la urmă, timpul de inactivitate este acelaşi, chiar dacă în alt moment).

Timpul de execuţie al acestui algoritm este aproximativ acelaşi cu cel al precedentului, singura diferenţă putând apare la complexitatea comunicaţiei; dar difuzarea şi colectarea sunt operaţii duale, deci vor consuma un timp identic; faptul că se face sumare în timpul colectării nu este semnificativ.

În concluzie, pentru ambii algoritmi prezentaţi, 6.10 şi 6.11, se foloseşte comunicaţie globală (de unde şi numele acestei secţiuni). În expresia $T_{6.10}$, partea reflectând calculele aritmetice este $O(n^2/p)$, iar cea descriind complexitatea comunicaţiei este $O(pn)$, în cazul cel mai rău, al inelului; deci, pentru $n \to \infty$, ponderea comunicaţiilor se devine nesemnificativă, şi eficienţa algoritmilor tinde către 1. Totuşi, pentru valori uzuale ale lui n, termenul reflectând comunicaţia este important (să nu uităm că $\sigma \gg \alpha$), ceea ce va conduce la o creştere lentă a eficienţei, pe măsură ce dimensiunea problemei creşte.

Cauzele relativei ineficienţe sunt două. În primul rând, procesoarele participă la operaţii de comunicaţie globală, care implică sincronizare, şi în care pierd timp în aşteptarea mesajelor; în plus, există operaţii efectuate de un singur procesor, în timp ce celelalte sunt inactive (tocmai din cauza necesităţii sincronizării). În al doilea rând, numărul acestor comunicaţii globale este foarte mare, deoarece toate mesajele au dimensiune 1; algoritmii au granularitate fină; aceasta implică o pondere egală a coeficienţilor σ şi β în timpul de comunicaţie.

Vom vedea în continuare cum se poate mări dimensiunea mesajelor, şi deci micşora numărul lor, păstrând structura algoritmilor de mai sus. Altfel zis, vom deduce

algoritmi cu granularitate mare (la nivel de bloc de matrice); pentru a nu lungi prezentarea, ne vom ocupa doar de algoritmul cu difuzare.

Să presupunem că A este partiţionată în blocuri pătrate ca în prima parte din relaţia (6.1); cum A este inferior triunghiulară, atunci $A_{ij} = 0_r$, pentru $i < j$, iar A_{jj} este inferior triunghiulară. De asemenea, ca în secţiunea referitoare la produsul matrice-vector, vom presupune vectorii x şi b împărţiţi fiecare în n_b vectori de dimensiune r, pe care-i vom nota cu X_i, respectiv B_i ($i \in 0 : n_b - 1$). Cu aceste notaţii, echivalentul relaţiei (6.6), dedus acum dintr-o bloc linie a ecuaţiei $Ax = b$, este:

$$A_{ii}X_i = B_i - \sum_{j=0}^{i-1} A_{ij}X_j,$$

şi arată că X_i poate fi calculat prin rezolvarea unui sistem inferior triunghiular "mic", de dimensiune r, după calculul tuturor necunoscutelor din vectorii X_j, cu $j < i$.

Algoritmul secvenţial asemănător cu 6.8 se deduce şi el imediat; forma este aproape identică, numai că operaţiile sunt acum la nivel de bloc, şi nu de element.

ALGORITM 6.12 ($TR(n)$ secvenţial la nivel de bloc, prin sumă vectorială)

$x \leftarrow b$
pentru $j = 0 : n_b - 1$
  rezolvă sistemul inferior triunghiular cu necunoscuta Y: $A_{jj}Y = X_j$
  $X_j \leftarrow Y$
  pentru $i = j + 1 : n_b - 1$
    $X_i \leftarrow X_i - A_{ij}X_j$

Desigur că numărul de operaţii este acelaşi, adică n^2.

Pentru a deduce echivalentul la nivel de bloc al algoritmului cu difuzare 6.10, trebuie să stabilim repartizarea matricei A. Deoarece se paralelizează bucla **pentru** cea mai interioară, adică operaţiile $X_i \leftarrow X_i - A_{ij}X_j$, cu $i > j$ (j este fixat), este natural ca bloc coloana j a matricei A să fie distribuită echilibrat procesoarelor; aceasta înseamnă o repartizare pe linii, *bloc ciclică*; fiecare procesor va avea în memoria locală $m = n_b/p = n/(rp)$ bloc linii, adică n/p linii; notând (doar acum) cu $P(i)$ procesorul care deţine blocul A_{ij}, şi care calculează X_i, se obţine:

ALGORITM 6.13 ($TR(n)$ cu difuzare, la nivel de bloc, pe arhitectură cu memorie distribuită, A repartizată bloc ciclic pe linii)

pentru $j = 0 : n_b - 1$
  dacă id $= P(j)$ **atunci**
(1) rezolvă sistemul inferior triunghiular cu necunoscuta Y: $A_{jj}Y = X_j$
(1') $X_j \leftarrow Y$
(2) **difuzare**: $P(j)$ trimite X_j celorlalte procesoare
  pentru $i = j + 1 : n_b - 1$
    dacă id $= P(i)$ **atunci**
(3) $X_i \leftarrow X_i - A_{ij}X_j$

Din nou forma este foarte asemănătoare cu cea a algoritmului la nivel de element 6.10; numărul de operaţii este, însă, diferit de cel din expresia $T_{6.10}$. Defalcat pe cele trei faze, timpul de execuţie este:

- $n_b r^2 \alpha = nr\alpha$ pentru calculul (secvenţial, deoarece un singur procesor lucrează la (1), restul aşteptând difuzarea) al soluţiilor X_j de către fiecare procesor, pentru că se rezolvă n_b sisteme triunghiulare de dimensiune r.
- $\approx c_d(n_b\sigma + n\beta) = c_d((n/r)\sigma + n\beta)$ – pentru cele n_b difuzări, fiecare a unui mesaj de dimensiune r.
- timpul pentru actualizările (3) rămâne acelaşi ca în $T_{6.10}$.

Rezumând, timpul primei faze este de r ori mai mare ca în $T_{6.10}$, iar coeficientul lui σ din timpul de difuzare se diminuează de r ori. Cum $\sigma \gg \alpha$, aceasta conduce la reducerea timpului total. Totuşi, timpul nu se îmbunătăţeşte considerabil, deoarece am înlăturat doar una din cauzele ineficienţei, cea cu pondere mai mică.

6.2.4 Algoritmi în front de undă

În algoritmii din secţiunea anterioară, la un moment dat, toate procesoarele execută operaţii de acelaşi tip, cooperând fie la actualizarea cu elementul x_j în algoritmul cu difuzare, fie la calculul sumelor parţiale necesare calculului unui element x_i în algoritmul cu colectare. O altă sursă de paralelism, exploatată în cele ce urmează, este lucrul simultan la mai multe elemente ale soluţiei, sau, echivalent, prelucrarea pipeline a unui singur element. Cu alte cuvinte, dacă în algoritmul cu difuzare se executau în paralel operaţiile dintr-o "coloană" a grafului de precedenţă din figura 6.3, iar în algoritmul cu colectare cele dintr-o "linie", putem acum încerca combinarea acestor abordări. Evident, există mai multe posibilităţi, dintre care vom expune câteva în continuare.

Ne vom ocupa întâi de varianta corespunzătoare algoritmului secvenţial cu sumă vectorială. Vom presupune acum o repartizare ciclic pe coloane a matricei A. După ce procesorul $P(j)$ calculează x_j, actualizarea termenilor x_i, cu $i > j$, nu poate fi împărţită cu alte procesoare deoarece $P(j)$ deţine în întregime coloana j din A; dacă toată actualizarea ar fi făcută de $P(j)$, atunci execuţia algoritmului ar fi pur secvenţială (deşi distribuită), pentru că celelalte procesoare nu ar avea nimic de făcut. Pentru a obţine paralelismul, se împarte actualizarea în mai multe etape.

Introducem un vector z, de dimensiune n, iniţializat cu zero, în care se vor acumula actualizările; deci, în final, $z_i = \sum_{j=0}^{i-1} a_{ij}x_j$. De data aceasta nu elementele din x vor circula (ca în algoritmul 6.10, unde se difuzau), ci elementele din z. Acest vector se împarte în mai multe segmente, de dimensiune s; evident, $1 \leq s \leq n$; în fiecare etapă $P(j)$ actualizează un segment, după care trimite acest segment către $P(j+1)$. În acest fel, $P(j+1)$ poate calcula x_{j+1} imediat după recepţionarea primului segment, prelucrând apoi segmentele pe măsură ce le primeşte de la $P(j)$.

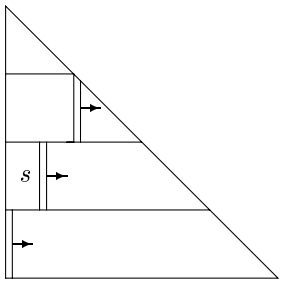


Figura 6.5: Poziţia segmentelor la un moment dat, în algoritmul front de undă pe coloane.

Să detaliem. Procesorul $P(0)$ calculează x_0, apoi începe calculul componentelor $z_i = a_{i0}x_0$, cu $i > 0$, ale vectorului de actualizare. După ce a calculat primele s astfel de elemente, adică primul segment, $P(0)$ le trimite lui $P(1)$, astfel încât acesta să-l poată calcula pe $x_1 \leftarrow (b_1 - z_1)/a_{11}$ şi să-şi înceapă actualizările, adică operaţii de genul $z_i \leftarrow z_i + a_{i1}x_1$, cu indicele i corespunzător liniilor primului segment. În acest timp, $P(0)$ continuă lucrul la următoarele s elemente ale lui z, al doilea segment. Pe scurt, după ce un procesor termină actualizarea unui segment, îl trimite mai departe, în aşa fel încât, la un moment dat, poziţia (spaţială) a segmentelor este cea sugerată în figura 6.5, iar mişcarea acestora este asemănătoare înaintării unui front de undă, de unde şi denumirea algoritmului.

Un exemplu este prezentat în figura 6.6, sub forma unei diagrame Gantt. Am notat cu S_j operaţia $x_j \leftarrow (b_j - z_j)/a_{jj}$ şi cu T_{ij} operaţia $z_i \leftarrow z_i + a_{ij}x_j$, în concordanţă cu graful de precedenţă din figura 6.3; cele două tipuri de operaţii au acelaşi timp de execuţie. Timpii morţi ai procesoarelor sunt figuraţi prin spaţii albe. Pentru a nu complica diagrama, am presupus comunicaţiile instantanee; liniile verticale marchează momentele în care are loc comunicaţia. Alegând $s = 2$, un procesor execută asupra unui segment două, sau una, operaţii de tip T_{ij}.

Deşi exemplul nu o sugerează pregnant, pentru $n \gg p$, în funcţie de dimensiunea segmentului şi de constantele specifice calculatorului gazdă (raportul timp comunicaţie/timp calcule, în special), este posibil ca toate procesoarele să fie active la un moment dat.

Vom prezenta în continuare o formă simplificată a algoritmului şi apoi unele detalii. Variabila locală *segment* este o mulţime conţinând cel mult s elemente ale vectorului de actualizare z; un procesor nu lucrează la un moment decât asupra unui singur segment; *nr_segm* este numărul de segmente care trebuie prelucrate pentru un anume j. Deoarece comunicaţiile au loc întotdeuna de la $P(j)$ la $P(j+1)$ şi repartizarea matricei A este ciclică, topologia minimală cerută de algoritm este inelul, pe care se comunică într-un singur sens.

P_0	S_0 T_{10} T_{20}	T_{30} T_{40}	T_{50} T_{60}	T_{70}		S_4	T_{54} T_{64}	T_{74}	
P_1		S_1 T_{21}	T_{31} T_{41}	T_{51} T_{61}	T_{71}			S_5 T_{65}	T_{75}
P_2			S_2	T_{32} T_{42}	T_{52} T_{62}	T_{72}		S_6	T_{76}
P_3					S_3 T_{43}	T_{53} T_{63}	T_{73}		S_7

Figura 6.6: Diagramă Gantt pentru execuţia algoritmului în front de undă pe coloane, pentru $n = 8$, $p = 4$, $s = 2$.

ALGORITM 6.14 ($TR(n)$ – în front de undă pe coloane, pe inel, A repartizată ciclic pe coloane)

pentru $j = 0 : n-1$
  dacă id $= P(j)$
    pentru $k = 1 : nr\_segm$ $\{nr\_segm = \lceil (n-j)/s \rceil\}$
      dacă id $= 0$ şi $j = 0$ **atunci**
(1) crează $segment$ $\{z((k-1)s+1 : ks) \leftarrow 0\}$
      altfel
(2) **recv**($segment$, stânga) {de la $P(j-1)$}
      dacă $k = 1$
(3) $x_j \leftarrow (b_j - z_j)/a_{jj}$
(4) $segment \leftarrow segment - \{z_j\}$
      pentru $z_i \in segment$
(5) $z_i \leftarrow z_i + a_{ij}x_j$
      dacă $|segment| > 0$ **atunci**
(6) **send**($segment$, dreapta) {către $P(j+1)$}

Toate segmentele sunt create, în (1), de către P_0, la prima iteraţie, şi transmise permanent către dreapta. Numărul de segmente pe care le recepţionează un procesor la un moment dat se calculează cu $nr\_segm = \lceil (n-j)/s \rceil$, deoarece pe coloana j sunt $n-j$ elemente nenule în A. După recepţia primului segment corespunzător unei coloane j, procesorul $P(j)$ poate calcula x_j, în (3) (care e echivalentul formulei (6.6)), după care z_j nu mai este necesar, deci poate fi eliminat din segment. Se observă deci că primul segment are cu câte un element mai puţin la fiecare necunoscută calculată şi dispare după primele s etape, moment în care procesul se repetă cu al doilea segment (devenit acum primul). Celelalte valori din primul segment şi celelalte segmente sunt actualizate în (5), după care segmentele sunt trimise către $P(j+1)$, dacă sunt nevide. Se poate modifica algoritmul astfel încât instrucţiunile (6), dintr-o iteraţie, şi (2), din următoarea, să se execute în paralel.

În acest fel segmentele parcurg inelul, fiecare până la dispariţie; ultimul segment, de exemplu, este transmis de $n-1$ ori; se poate aprecia, în general, că volumul comunicaţiei este mare; totuşi, nu trebuie uitat că multe din transmisii pot avea loc în paralel. Dacă s, dimensiunea unui segment, este mic, creşte gradul de paralelism, dar şi timpul de comunicaţie, deoarece numărul segmentelor este mare; dacă s este mare, numărul de mesaje se reduce, dar scade şi paralelismul. Faptul că volumul comunicaţiei scade pe măsură ce calculele avansează compensează într-un fel trecerea de mai multe ori a unui segment pe la acelaşi procesor. De asemenea, se poate aprecia că în cazul comunicaţiilor asincrone timpul de execuţie poate fi redus, chiar semnificativ, faţă de cazul sincron.

Timpul de execuţie al acestui algoritm este următorul (pentru detalii vezi [14]):

$$T_{6.14} = \frac{1}{p}(n^2 + np + \frac{s(s-1)}{2}p^2)(\alpha + \frac{\sigma}{s} + \beta),$$

şi se minimizează în funcţie de s; optimul se obţine dintr-o ecuaţie de gradul 3 în s, ai cărei coeficienţi depind de caracteristicile calculatorului gazdă. Deci, pentru fiecare calculator şi pentru fiecare n, trebuie iniţial calculată valoarea optimă a lui s. Calculul nu trebuie făcut la fiecare execuţie, ci se poate crea, o dată, un tabel cu valorile optime, din care se extrage la execuţie valoarea adecvată.

Varianta în front de undă a algoritmului cu produs scalar presupune o repartizare pe linii a matricei A. Procesorul $P(i)$ trebuie să calculeze singur produsul scalar ce implică linia i a matricei; dacă ar executa acest calcul în întregime, atunci calculele s-ar desfăşura pur secvenţial. Se foloseşte din nou ideea împărţirii în segmente, de data aceasta a calculului produsului scalar.

Având în vedere asemănarea pronunţată cu algoritmul în formă de undă orientat pe coloane, ne vom mulţumi cu o scurtă expunere comparativă. Dacă în acel algoritm circulau valorile lui z, care se actualizau pe baza valorilor locale ale lui x, acum lucrurile se petrec pe dos; pentru calculul $z_i = \sum_{j=0}^{i-1} a_{ij}x_j$, procesorul $P(i)$ are nevoie doar de valorile x_j, linia i din A fiindu-i proprie; deci ideea este de a face să circule segmente de dimensiune s, conţinând elemente din x, în timp ce elementele din z rămân locale. Pentru a calcula x_i, procesorul $P(i)$ primeşte astfel de segmente, actualizând z_i, până când este obţinută suma de mai sus; apoi poate efectiv determina x_i, pe care-l ataşează ultimului segment; toate segmentele sunt trimise spre $P(i+1)$, pe rând, astfel încât un procesor lucrează cu un singur segment la un moment dat. Poziţia segmentelor la un moment dat este cea sugerată în figura 6.7.

În algoritmul de mai jos, *segment* este o variabilă locală care conţine cel mult s elemente din x; suma z_i poate fi găzduită de o variabilă scalară; am păstrat notaţia numai pentru a nu strica asemănarea cu algoritmul pe coloane.

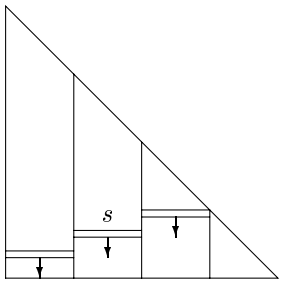


Figura 6.7: Poziţia segmentelor la un moment dat, în algoritmul front de undă pe linii.

ALGORITM 6.15 ($TR(n)$ – în front de undă pe linii, pe inel, A repartizată ciclic pe linii)

pentru $i = 0 : n-1$
  $z_i \leftarrow 0$
  dacă id $= P(i)$ **atunci**
    pentru $k = 1 : nr\_segm - 1$ $\{nr\_segm = \lceil i/s \rceil\}$
(1) **recv**($segment$, stânga)
     în paralel
(2) **send**($segment$, dreapta) {către $P(i+1)$}
     pentru $x_j \in segment$
(3) $z_i \leftarrow z_i + a_{ij}x_j$
(4) **recv**($segment$, stânga) {dacă $i \bmod s = 0$}
     pentru $x_j \in segment$
(5) $z_i \leftarrow z_i + a_{ij}x_j$
(6) $x_i \leftarrow (b_i - z_i)/a_{ii}$
(7) $segment \leftarrow segment \cup \{x_i\}$ {sau crează un nou segment}
(8) **send**($segment$, dreapta)

Trebuie remarcate câteva detalii de implementare. Pentru un i fixat, tratarea segmentelor e despărţită în două etape, ultimul segment fiind prelucrat separat. Pentru primele, transmisia (2) are loc, în mod ideal, imediat după recepţia (1), deoarece conţinutul segmentului nu se modifică, deci e bine ca $P(i+1)$ să-l primească cât mai devreme; pe de altă parte, pentru a nu se aştepta efectuarea acestei comunicaţii, actualizările (3) au loc în paralel (sau concurent, dacă arhitectura procesorului nu permite altfel). Pentru ultimul segment se execută recepţia, în (4), dar numai dacă este cazul, adică dacă nu trebuie creat un nou segment; acum actualizările (5) preced transmisia (8), deoarece trebuie calculat x_i, în (6), şi adăugat segmentului, în (8). Se observă că numărul de segmente este acum în continuă creştere, până când se ajunge la numărul maxim de segmente.

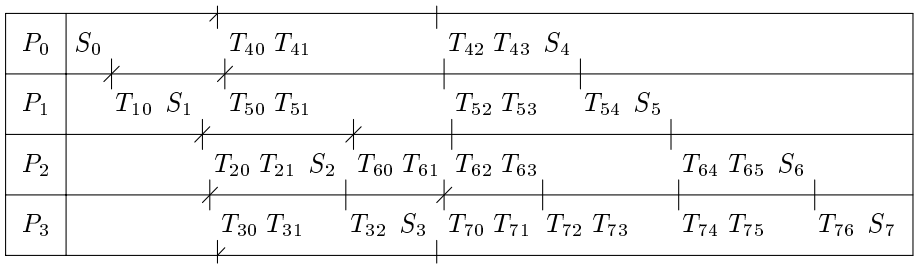


Figura 6.8: Diagramă Gantt pentru execuţia algoritmului în front de undă pe linii, pentru $n = 8$, $p = 4$, $s = 2$.

Diagrama Gantt a unui exemplu este prezentată în figura 6.8, pentru aceiaşi parametri ca în cazul algoritmului în front de undă pe coloane. De data aceasta comunicaţia este simbolizată printr-o linie între zonele alocate unor procesoare vecine; am presupus că durata comunicaţiilor este nenulă, dar foarte mică, pentru a sugera mai clar ordinea transmisiilor. O mică linie oblică marchează transmisiile primului segment. Comunicaţia a fost presupusă sincronă; pentru a nu complica diagrama, am presupus că transmisia (2) şi actualizările (3) se pot efectua în orice ordine (cea mai avantajoasă), dar secvenţial.

Timpul de execuţie pentru acest algoritm are o expresie complicată, însă cu o alură asemănătoare celui pentru algoritmul în front de undă orientat pe coloane, adică cu un minim pentru $1 \leq s \leq n$.

6.2.5 Algoritmi ciclici

Algoritmii în front de undă au dezavantajul unui volum de comunicaţie relativ mare. Cei care vor fi prezentaţi în continuare vor folosi minimum de comunicaţie, atât ca număr de mesaje, cât şi ca număr de elemente de matrice transmise.

Vom începe cu varianta paralelă a algoritmului cu sumă vectorială. Se presupune o repartizare ciclic pe coloane a matricei A. Există o oarecare asemănare cu algoritmul în front de undă corespondent, în sensul că actualizările se acumulează într-un vector z, de dimensiune n, iniţializat cu zero şi care, în final, conţine valorile $z_i = \sum_{j=0}^{i-1} a_{ij}x_j$, pe baza cărora se calculează elementele din x. De data aceasta, însă, vectorul z nu se mai împarte în mai multe segmente; există un singur pachet (segment), de dimensiune $p - 1$, care circulă de-a lungul unui inel, conţinând o parte din elementele lui z, sub forma unor sume parţiale.

Vom prezenta întâi algoritmul, şi apoi unele explicaţii; în ce priveşte notaţiile, SUM este pachetul de $p - 1$ elemente care circulă, şi pentru care fiecare procesor trebuie să rezerve spaţiu; de asemenea, fiecare procesor are o variabilă locală z, de dimensiune n (sunt necesare doar $n - p + 1$ valori, dar nu ne complicăm), în care

calculează partea sa din suma de mai sus; pentru toate procesoarele, vectorii SUM şi z se iniţializează cu 0. Algoritmul este următorul:

ALGORITM 6.16 ($TR(n)$ – ciclic pe coloane, pe inel, A repartizată ciclic pe coloane)

pentru $j = 0 : n-1$
  dacă id $= P(j)$
(1) **recv**(SUM, stânga) {de la $P(j-1)$}
(2) $z_j \leftarrow z_j + SUM(0)$
(3) $x_j \leftarrow (b_j - z_j)/a_{jj}$
(4) $SUM(0 : p-3) \leftarrow SUM(1 : p-2)$ {elimină $SUM(0)$ din pachet}
  pentru $i = 1 : p-2$
(5) $SUM(i-1) \leftarrow SUM(i-1) + z_{j+i} + a_{j+i,j}x_j$
(6) $SUM(p-1) \leftarrow z_{j+p-1} + a_{j+p-1,j}x_j$
(7) **send**(SUM, dreapta) {către $P(j+1)$}
  pentru $i = j+p : n-1$
(8) $z_i \leftarrow z_i + a_{ij}x_j$

Deci, pachetul SUM circulă de-a lungul inelului de procesoare, fiecare procesor primindu-l din stânga în (1) şi retransmiţându-l spre dreapta în (7), după ce a făcut toate modificările necesare. Pachetul conţine actualizări pentru următoarele $p-1$ elemente din x de calculat; deoarece o linie a matricei A este repartizată între toate procesoarele, fiecare dintre acestea contribuie la calculul sumei $z_i = \sum_{j=0}^{i-1} a_{ij}x_j$ cu produsele $a_{ij}x_j$ pe care le poate efectua cu date din memoria locală; în loc de a transmite tot vectorul z, se poate folosi doar SUM, care la plecarea dintr-un procesor conţine informaţia necesară celorlalte $p-1$ procesoare, care e suficientă până la revenirea pachetului la procesorul curent. Instrucţiunea (2) reprezintă obţinerea valorii finale a lui z_j, după ce în $SUM(0)$ s-au acumulat actualizările efectuate de celelalte procesoare. Deci, în (3), $P(j)$ calculează necunoscuta x_j folosind chiar formula (6.6). Apoi, în (4), elimină prima valoare din SUM (în algoritm, printr-o deplasare la stânga a vectorului SUM), care nu mai este necesară mai departe; actualizează, în (5), celelalte valori din SUM cu necunoscuta proaspăt calculată şi adaugă, în (6), la pachetul SUM suma sa parţială (actualizată şi cu necunoscuta curentă) destinată calculului variabilei x_{j+p-1} – care va fi calculată de $P(j+p-1)$, vecinul din stânga al lui $P(j)$, dar cel mai depărtat în sensul de circulaţie al pachetului SUM. În acest moment pachetul SUM este complet prelucrat şi poate fi expediat vecinului din dreapta. În fine, în (8), $P(j)$ actualizează cu necunoscuta proaspăt calculată sumele parţiale z pentru celelalte necunoscute (de la $j+p$ încolo), operaţie care ocupă, în general, mult mai mult timp decât toate cele precedente; doar pentru j suficient de mare, adică apropiat de n, aceste actualizări sunt de scurtă durată.

Se poate demonstra că, pentru acest algoritm, comunicaţia este minimală, atât ca număr de pachete cât şi ca număr de elemente transmise. Aceasta nu înseamnă că algoritmul e optim ca timp total de execuţie, pentru că apare altă problemă; procesorul $P(j)$ poate termina actualizările (8) ale vectorului z înainte de a reprimi

un nou pachet SUM; în acest caz, el nu are nimic de făcut şi va trebui să aştepte; timpul mort astfel acumulat poate avea o pondere însemnată – cu atât mai mare cu cât raportul n/p este mai mic; acest lucru este absolut normal: dacă p este mare, un procesor va lucra mai mult la actualizarea pachetului SUM, care va circula astfel mai lent, în timp ce actualizarea vectorului z va dura relativ puţin.

Dacă $n > p(t_p + p)$ (unde t_p este costul, măsurat în flop, al transmisiei a $p-1$ elemente în virgulă mobilă către un vecin[1]) – ceea ce înseamnă o valoare destul de mare, atunci timpul de execuţie al algoritmului va avea aproximativ expresia:

$$T_{6.16} = \left[\frac{1}{p}(n^2 + np) + p(t_p + p)^2\right]\alpha.$$

Eficienţa algoritmului tinde asimptotic spre 1, dar se apropie de această valoare numai pentru valori mari ale lui n. Se arată în [20] că o eficienţă teoretică de aproximativ 50% se obţine pentru $n = p(t_p + p)$.

Un algoritm asemănător se poate stabili şi pentru o repartizare pe linii a matricei A. Ca şi la algoritmul în front de undă pe linii, va circula x; ca şi la algoritmul ciclic pe coloane, va exista un singur pachet, numit acum $XSUB$ (notaţia e consacrată) şi conţinând ultimele $p-1$ necunoscute calculate. Pentru a păstra stilul de prezentare, am folosit şi variabila z, cu semnificaţia de până acum; trebuie observat că z şi x pot fi găzduite de aceeaşi variabilă locală şi că z este repartizat ciclic între procesoare ($P(i)$ are elementul z_i). Algoritmul este următorul:

ALGORITM 6.17 ($TR(n)$ – ciclic pe linii, pe inel, A repartizată ciclic pe linii)

pentru $i = 0 : n-1$
  dacă id $= P(i)$
(1) **recv**($XSUB(1:p-1)$, stânga)
  pentru $j = 1 : p-1$
(2) $z_i \leftarrow z_i + a_{i,i-p+j} \cdot XSUB(j)$
(3) $x_i \leftarrow (b_i - z_i)/a_{ii}$
(4) $XSUB(0:p-2) \leftarrow XSUB(1:p-1)$
(5) $XSUB(p-1) \leftarrow x_i \quad \{XSUB \leftarrow XSUB - \{x_{i-p+1}\} \cup \{x_i\}\ \}$
(6) **send**($XSUB(1:p-1)$, dreapta)
  pentru $k = i+1 : n-1$
    dacă id $= P(k)$ **atunci**
      pentru $j = 1 : p$
(7) $z_k \leftarrow z_k + a_{k,i-p+j} \cdot XSUB(j-1)$

Am folosit un mic artificiu tehnic în utilizarea pachetului $XSUB$; cele $p-1$ elemente transmise sunt cele din poziţiile $1 : p-1$; vom explica puţin mai târziu la ce foloseşte poziţia 0. Un procesor $P(i)$, după ce recepţionează în (1) pachetul $XSUB$, completează suma parţială z_i în (2), pe baza soluţiilor aflate în $XSUB$ (calculate în

[1] Cu alte cuvinte $t_p = (\sigma + (p-1)\beta)/\alpha$.

ultima tură de celelalte $p-1$ procesoare), după care calculează x_i folosind clasica formulă (6.6). Apoi, în (4) şi (5), şterge din pachetul $XSUB$ cea mai veche componentă a soluţiei şi adaugă la acesta necunoscuta proaspăt calculată x_i; după care, în (6), trimite $XSUB$ mai departe. Apoi actualizează elementele locale (de pe liniile care îi aparţin) ale vectorului z, folosind valorile din $XSUB$; în algoritm am ţinut seama de deplasarea spre stânga din (4) a elementelor din $XSUB$; de aceea, în actualizările (8), apare şi $XSUB(0)$, care nu trebuia trimis spre $P(i+1)$, dar e încă necesar pentru actualizările locale.

Acest algoritm are proprietăţi similare cu cele ale algoritmului ciclic pe coloane.

6.2.6 Comparaţii ale algoritmilor precedenţi

Am prezentat până acum trei perechi de algoritmi paraleli de rezolvare a sistemelor liniare triunghiulare. Dintre aceştia, algoritmii cu comunicaţie globală sunt, teoretic, cei mai puţin performanţi. Cel mai bun algoritm depinde de raportul n/p; dacă acesta este mic, algoritmii în front de undă sunt superiori; dacă este mare (număr mic de procesoare faţă de dimensiunea problemei), atunci algoritmii ciclici sunt mai buni.

De exemplu, în [14] se raportează că, pe un hipercub de tip Ncube, pentru $n = 500$, pentru $p = 4$, 8, 16 algoritmii ciclici au avut cel mai bun timp de execuţie; în schimb, pentru $p = 32$, 64 algoritmii în front de undă au fost superiori.

Se poate deci concluziona că, din punct de vedere practic, pentru rezolvarea paralelă a unui sistem liniar triunghiular se va alege unul dintre algoritmii în front de undă sau ciclic, cu repartizare a datelor pe linii sau pe coloane după cum e mai convenabil în context. Alegerea se face în funcţie de valoarea n/p şi de caracteristicile calculatorului pe care se face implementarea (în special raportul timp de comunicaţie/timp de calcul). Repartizarea pe linii sau pe coloane se alege în funcţie de contextul în care se rezolvă $TR(n)$.

Nu trebuie neglijaţi, totuşi, nici algoritmii cu comunicaţie globală, în special variantele la nivel de bloc (de genul algoritmului 6.13), datorită simplităţii lor şi a folosirii unor operaţii de comunicaţie care sunt deseori gata implementate cu mare eficienţă.

6.2.7 Un algoritm pe tor

Vom prezenta în continuare o modalitate de rezolvare a problemei $TR(n)$ pe un tor de dimensiuni egale. Primul lucru care trebuie stabilit este repartizarea matricei A; de data aceasta, datorită "asemănării" între un tor şi o matrice, există o soluţie simplă şi unică; profitând şi de experienţa anterioară, vom considera o repartizare ciclică atât pe linii, cât şi pe coloane; altfel spus, elementul a_{ij} va fi deţinut de procesorul de pe linia $i \bmod \sqrt{p}$ şi coloana $j \bmod \sqrt{p}$ ale torului. Un exemplu este prezentat în figura 6.9; se observă că procesoarele au aproximativ acelaşi număr de elemente din A. În ce priveşte vectorii b şi x, din motive ce vor deveni clare mai târziu, îi presupunem repartizaţi ciclic doar procesoarelor de pe diagonala torului; deci b_i aparţine procesorului P_{tt}, cu $t = i \bmod \sqrt{p}$; acelaşi procesor va deţine în final x_i.

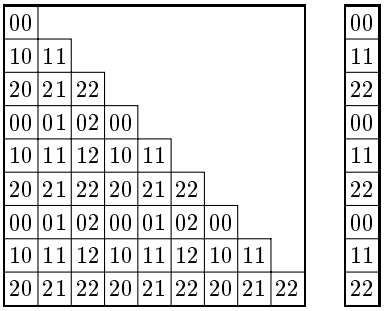


Figura 6.9: Repartizare a matricei inferior triunghiulare A şi a vectorului x pe un tor 3×3, pentru $n = 9$.

Să încercăm să deducem un algoritm care să combine ideile algoritmilor cu difuzare 6.10 şi cu colectare 6.11. E natural să dorim aceasta, din moment ce repartizarea pe tor îmbină repartizările din cei doi algoritmi amintiţi, pe linii, respectiv pe coloane. Grupând procesoarele torului pe linii, algoritmul cu difuzare s-ar putea generaliza astfel: după calculul unui element x_j, grupul de procesoare $P(j)$ îl difuzează celorlalte grupuri, care efectuează actualizări. Pe de altă parte, grupând procesoarele pe coloane, se poate generaliza algoritmul cu colectare: la iteraţia i, fiecare grup calculează partea din suma din (6.6) care îi revine, după care se obţine întreaga sumă printr-o colectare cu sumare cu destinaţia $P(i)$, grup care poate calcula acum x_i.

Deci, doar din argumente de bun simţ, se observă că se vor face difuzări pe coloanele torului şi colectări pe linii. De asemenea, deoarece calculul final al unui element din x este efectuat de un singur procesor, şi trebuie să alegem câte un procesor reprezentativ pentru fiecare linie şi fiecare coloană, e normal ca procesoarele de pe diagonala torului să fie cele "avantajate"; de aici, cerinţa repartizării vectorului b numai între aceste procesoare.

Să trecem acum la o abordare mai concretă. Introducem matricea $W \in \mathbf{R}^{n \times \sqrt{p}}$; fiecare coloană de procesoare din tor posedă o coloană din W, repartizarea pe linii fiind ciclică, la fel ca pentru A. Procesorul P_{st}, care deţine elementul w_{it} (deci $s = i \bmod \sqrt{p}$), va avea de calculat suma (prin care definim W):

$$w_{it} = \sum_{j<i,\ P(j)=t} a_{ij} x_j, \quad (\text{unde } P(j) = j \bmod \sqrt{p}).$$

Se observă că această sumă necesită doar elemente a_{ij} locale procesorului. De asemenea, folosind consacratul vector z, deducem imediat că

$$z_i (= \sum_{j=0}^{i-1} a_{ij} x_j) = \sum_{t=0}^{\sqrt{p}-1} w_{it}.$$

Această relaţie sugerează că, o dată calculate toate valorile w_{it} de pe linia $s = P(i)$ a torului, $x_i = (b_i - z_i)/a_{ii}$ se poate calcula după o colectare cu sumare pe linia respectivă, cu destinaţia P_{ss} (care deţine atât b_i, cât şi a_{ii}); este, de fapt, ideea algoritmului cu colectare 6.11.

Pe de altă parte, pentru a calcula w_{it}, un procesor are nevoie de toate valorile x_j, pentru $P(j) = t$; ori acestea sunt calculate de procesorul P_{tt}. Deci, de fiecare dată când un procesor de pe diagonala torului calculează o valoare din x, el trebuie să o difuzeze tuturor procesoarelor de pe coloana sa, acestea urmând a actualiza valorile w_{it} care le sunt locale. Deoarece operaţia de actualizare va apare în mai multe ocazii, să descriem procedura pe care o urmează procesorul P_{st}:

procedură *actualizare*(k, x_j)
  pentru $i = k + 1 : n - 1$
    dacă $i \bmod \sqrt{p} = s$ **atunci** {pentru liniile proprii}
      $w_{it} \leftarrow w_{it} + a_{ij}x_j$

Introducerea parametrului suplimentar k se va dovedi necesară mai târziu; în mod normal $k = j$, deci un procesor actualizează toate elementele care îi aparţin, pentru $i > j$.

S-a conturat până acum următorul algoritm de principiu, schiţat pentru ansamblul procesoarelor:

ALGORITM 6.18 ($TR(n)$ pe tor, schiţă generală)
  pentru $j = 0 : n - 1$
    $t \leftarrow j \bmod \sqrt{p}$
    colectare cu sumare (pe linia t a torului) în P_{tt}: $z_j \leftarrow \sum_{l=0}^{\sqrt{p}-1} w_{jl}$
    P_{tt} calculează $x_j \leftarrow (b_j - z_j)/a_{jj}$
    P_{tt} difuzează x_j pe coloana t a torului
    P_{st}, cu $s \in 0 : \sqrt{p} - 1$ (coloana t a torului), efectuează *actualizare*(j, x_j)

Acest algoritm arată plăcut, dar ascunde defecte majore; deci, înainte de a detalia, să le evidenţiem şi să încercăm să le eliminăm. Un exemplu simplu ne va fi de folos; după ce P_{00} calculează x_0, îl difuzează pe coloana 0 a torului; printre alţii, îl recepţionează şi P_{10}, care trece imediat la actualizări; în acest timp, P_{11}, care aşteaptă de la P_{10} valoarea w_{10}, necesară pentru calculul lui x_1, nu are nimic de făcut; de-abia după ce P_{10} îşi termină actualizările, este disponibil pentru comunicarea cu P_{11} (participarea la colectare). Se arată astfel că algoritmul nu are un grad înalt de paralelism; dacă toate procesoarele de pe o coloană a torului sunt simultan ocupate (mai puţin pe durata calculului lui x_j, care e nesemnificativă), în schimb coloanele lucrează aproximativ secvenţial; timpul de execuţie nu va fi cu mult mai bun decât cel al unui inel de numai $\sqrt{p}$ procesoare, ceea ce e absolut neconvenabil.

Soluţia e de a amâna o parte a actualizărilor până când se rezolvă toate problemele de comunicaţie ale etapei curente; de exemplu, P_{10} va calcula întâi w_{10}, îl va trimite către P_{11}, iar apoi va actualiza $w_{\sqrt{p}+1,0}$, $w_{2\sqrt{p}+1,0}$, etc. În acest fel P_{11} nu va aştepta

decât un timp minim, iar P_{10} va avea doar de schimbat ordinea unor operaţii. Ideea este deci de a face comunicaţiile cât mai devreme cu putinţă, imediat ce datele de transmis sunt gata.

Această tactică ne obligă la spargerea în etape a operaţiilor de difuzare şi colectare, care vor avea loc într-un timp mai îndelungat (nu vor însemna practic o sincronizare a procesoarelor), dar care se vor suprapune cu alte operaţii. Pentru a obţine o bună eşalonare în timp a comunicaţiilor între vecini, vom efectua difuzarea prin transmitere în jos, pe inelul format de fiecare coloană a torului, iar colectarea prin transmitere la dreapta, pe fiecare linie a torului.

Vom prezenta acum algoritmul pentru procesorul P_{st}, într-o formă mai amănunţită. Pentru a clarifica oarecum lucrurile, să precizăm comunicaţiile la care participă un procesor nediagonal; într-o etapă, P_{st} participă la o difuzare pe coloana t a torului şi la o colectare pe linia s; deci, are de recepţionat câte un mesaj de sus şi din stânga, şi de transmis jos şi spre dreapta.

ALGORITM 6.19 ($TR(n)$ pe tor, detaliat, A repartizată ciclic pe linii şi coloane, pentru procesorul P_{st})

iniţializează toţi w_{it} locali cu 0
pentru $j = 0 : n-1$
  dacă $t = j \bmod \sqrt{p}$ **atunci**
    dacă $s = t$ **atunci** {procesor diagonal}
(1) **recv**(z_j, stânga) {dacă $j \neq 0$}
(2) $z_j \leftarrow z_j + w_{jt}$
(3) $x_j \leftarrow (b_j - z_j)/a_{jj}$
(4) **send**(x_j, jos) {dacă $j \neq n-1$}
(5) $actualizare(j, x_j)$
    altfel {$s \neq t$, procesor nediagonal}
(6) **recv**(x_j, sus)
(7) **send**(x_j, jos) {dacă $s \neq (t-1) \bmod \sqrt{p}$}
(8) $i \leftarrow j + (s-t) \bmod \sqrt{p}$
(9) $w_{it} \leftarrow w_{it} + a_{ij}x_j$
(10) **recv**(z_i, stânga) {dacă $t \neq (s+1) \bmod \sqrt{p}$}
(11) $z_i \leftarrow z_i + w_{it}$
(12) **send**(z_i, dreapta)
(13) $actualizare(j + \sqrt{p}, x_j)$

Deci, după ce un procesor diagonal primeşte în (1) rezultatul colectării de pe linia sa, la care adaugă în (2) şi contribuţia sa, e pregătit să calculeze un element din x în (3), aplicând formula (6.6); apoi, în (4), iniţiază difuzarea elementului proaspăt calculat şi actualizează în funcţie de acesta elementele din W proprii, în (5).

Un procesor nediagonal aşteaptă întâi ultima necunoscută x_j calculată de procesorul diagonal de pe coloana sa, pe care o trimite imediat mai departe; (6) şi (7) sunt contribuţia procesorului la difuzarea pe coloana sa. Apoi actualizează, în (9), doar primul element local w_{it}; indicele i, calculat anterior în (8), este cel mai mic indice al

unei linii proprii, mai mare decât j; într-adevăr, dacă $s > t$ atunci $i = j + (s - t)$, iar dacă $s < t$ atunci $i = j + (s - t + \sqrt{p})$. Cu acest w_{it}, procesorul participă la colectarea cu sumare de pe linia sa, în instrucţiunile (10), (11) şi (12); în sfârşit, poate continua actualizările, în (13). Ca detaliu, instrucţiunile (7), (9) şi (10) pot decurge în paralel; toate valorile z_i locale unui procesor pot fi găzduite de o singură variabilă scalară.

O formulă exactă pentru timpul de execuţie al acestui algoritm este greu de dedus. Să ne mulţumim cu aprecierea că operaţiile aritmetice sunt echilibrat împărţite între procesoare, iar volumul comunicaţiilor este aproximativ $4(n/\sqrt{p})(\sigma + \beta)$ (un procesor efectuează 4 comunicaţii pe etapă, şi sunt $n/\sqrt{p}$ etape). Desigur, pot exista timpi morţi în activitatea procesoarelor, dar ei nu pot fi prea mari, deoarece actualizările ocupă mult mai mult timp decât celelalte operaţii.

Probleme

P 6.2.1 Scrieţi analogul algoritmului cu difuzare 6.10 pentru rezolvarea sistemului *superior* triunghiular $Ax = b$ ($a_{ij} = 0$ pentru $i > j$).

P 6.2.2 Scrieţi un algoritm pentru rezolvarea $TR(n)$ pentru o arhitectură cu memorie comună, pornind de la varianta secvenţială cu sumă vectorială.

P 6.2.3 Desenaţi diagrame Gantt pentru evoluţia algoritmilor cu difuzare 6.10 şi cu colectare 6.11, în cazul $n = 6$, $p = 3$.

P 6.2.4 La fel, pentru evoluţia algoritmilor ciclici de rezolvare a problemei $TR(n)$, pe coloane 6.16 şi pe linii 6.17.

P 6.2.5 Ce instrucţiuni pot fi executate în paralel (sau concurent) de către un procesor, în algoritmul ciclic pe coloane 6.16 de rezolvare a problemei $TR(n)$? Dar în algoritmul 6.17 ?

P 6.2.6 Cum se poate ajunge de la o repartizare a unui vector pe tor ca aceea a lui x din figura **??** (pentru înmulţirea matrice-vector), la cea din algoritmul pentru $TR(n)$ pe tor ?

Răspunsuri la probleme

P1.1.1 Numărul de tacte la care se produce un rezultat este dictat de timpul cel mai mare al unui "post"; răspunsul este deci 2. Se poate obţine un rezultat la fiecare tact dacă se dublează postul B, ca în figura 7.1; comutatoarele dinaintea şi de după cele două blocuri B îşi schimbă poziţia la fiecare tact şi sunt în fază (blocul C citeşte rezultatul furnizat de un bloc B şi, în acelaşi moment blocul B respectiv trebuie să citească un nou operand). În primul caz 100 de rezultate finale erau produse în 203 tacte, în al doilea, în 103.

P1.2.1 Se procedează prin inducţie după p. Pentru $p = 1$ este evident. Presupunem că orice arbore de ordin p are $p - 1$ arce. Fie un arbore de ordin $p + 1$; presupunem că nu există nici un nod cu gradul 1; deci, fiecare nod are cel puţin două arce. Vom construi un ciclu astfel: plecăm dintr-un nod oarecare pe unul din arce; în nodul următor, continuăm pe al doilea arc al său, primul fiind deja utilizat; şi tot aşa, până când se ajunge într-un nod deja parcurs; arcele sunt toate distincte, deci am obţinut un ciclu, ceea ce e imposibil într-un arbore. Deci există un nod de grad 1; eliminând acest nod şi arcul său se obţine un arbore de ordin p, cu $p - 1$ arce; deci arborele iniţial are p arce.

P1.2.2 $\Delta p/2$. Din fiecare nod pleacă Δ arce; fiecare arc uneşte două noduri.

P1.2.3 Un drum între cele două noduri se construieşte din arce aflate pe dimensiunile egale cu poziţiile biţilor diferiţi. De exemplu, de la 100 la 001 se poate ajunge în doi paşi, prin 000 sau 101 (pe dimensiunile 2,0, respectiv 0,2).

P1.2.4 Nodurile la distanţă x de P_0 au reprezentări binare conţinând x biţi de 1. Deci, în total, sunt $\binom{d}{x}$ noduri. Reamintim în treacăt o identitate binecunoscută: $\binom{d}{0} + \binom{d}{1} + \ldots + \binom{d}{d} = 2^d$, care arată aici numărul total de noduri în hipercub, acestea fiind grupate după distanţele faţă de P_0.

P1.2.5 Sunt şase cazuri posibile; ne referim la figura 1.11a; deci, vecinii lui P_{ij} sunt:
– $P_{i+1,j}$, $P_{i,j+1}$, dacă $i = 0$, $j = 0$ sau $j = 1$, i impar şi $\neq n - 1$.

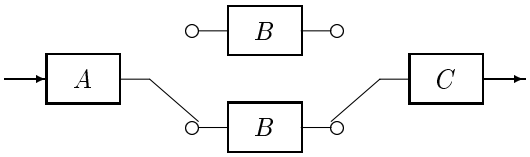


Figura 7.1: Schema pipeline pentru problema 1.1.1.

– $P_{i-1,j}$, $P_{i,j+1}$, dacă $i = n-1$, $j = 0$ sau $j = 1$, i par şi $\neq 0$.
– $P_{i,j-1}$, $P_{i+1,j}$, dacă i par, $j = n-1$.
– $P_{i,j-1}$, $P_{i-1,j}$, dacă i impar, $j = n-1$.
– $P_{i-1,j}$, $P_{i+1,j}$, dacă $j = 0$, $0 < i < n-1$.
– $P_{i,j-1}$, $P_{i,j+1}$, altfel.

Iată deci că o figură simplă ascunde complicaţii destul de mari.

P1.2.6 Dacă i este numărul binar şi g codul Gray, conversia binar–Gray se face cu $g_l = i_l \oplus i_{l+1}$, cu $0 \leq l \leq d-1$ (cu convenţia că $i_d = 0$). Invers, conversia Gray–binar se face cu $i_l = g_l \oplus g_{l+1} \oplus \ldots \oplus g_{d-1}$, sau $i_l = g_l \oplus i_{l+1}$, cu $l = d-1 : -1 : 0$.

În ciclul hamiltonian construit după codul Gray reflectat, care este un inel, procesorul g are vecinii g' şi g'', codurile Gray ale numerelor binare $i' = i-1$ şi $i'' = i+1$, unde i are codul Gray g. Deci, pentru P_4, parcurgem etapele: $g = 4 = 0100$ este codul Gray pentru $i = 0111$ (P_4 este pe poziţia 7 în inel); $i' = 0110$, $i'' = 1000$, deci $g' = 0101 = 5$ şi $g'' = 1100 = 12$.

P2.2.1 Pe hipercub, algoritmul este următorul:

1. **dacă** id $= k$
    1. **pentru** $i = 0 : d-1$, **în paralel**
        1. **send**(M, i)
2. **altfel dacă** $dist_H(\text{id}, k) = 1$ {vecinii sunt la distanţă Hamming 1 }
    1. $i \leftarrow$ numărul de ordine al bitului de 1 din id $\oplus\, k$
    2. **recv**(M, i)

P2.2.2 În mod non-blocant, fiecare procesor transmite şi recepţionează pe toate canalele, în paralel. În mod blocant sincron, un procesor transmite întâi pe o dimensiune i dacă e.g. bitul i din adresa sa este 1, şi recepţionează întâi dacă acest bit este 0.

P2.4.1 Da. Fiecare procesor va avea alocată o parte din memoria comună, care va simula memoria locală. Pentru fiecare canal de comunicaţie între două procesoare va fi alocată câte o zonă, împărţită în două, fiecare parte pentru un sens de comunicaţie.

P2.4.2 Suma se calculează în u_0. Considerăm id $\equiv i$.

1. $u_{2i} \leftarrow u_{2i} + u_{2i+1}$
2. **dacă** $i = 0$
    1. $u_0 \leftarrow u_0 + u_2$

P3.3.1 $j(\sigma+\beta) + \frac{m-\Delta}{\Delta}\beta$. Dacă graful este neregulat, Δ este gradul nodului sursă.

P3.3.2 Sunt $p-1$ căi; una are lungime 1, iar celelalte lungime 2 şi trec prin cele $p-2$ noduri diferite de sursă şi destinaţie.

P3.3.3 Când toate căile au aceeaşi lungime, deci atunci când $j = \sqrt{p}$; poate că merită să faceţi două mici desene, cu $\sqrt{p}$ par, respectiv impar.

P3.3.4 Un exemplu este cel din figura 7.2, în care sunt reprezentate câte două căi într-un desen; primele două au lungime 1, respectiv 7, iar celelalte lungime 3, respectiv 5.

P3.3.5 $b = a \oplus s$ are j biţi de 1. Fie o permutare $(0, 1, \ldots, d-1) \xrightarrow{i} (i_0, i_1, \ldots, i_{d-1})$, astfel încât $b_{i_0} = \ldots = b_{i_{j-1}} = 1$ şi $b_{i_j} = \ldots = b_{i_{d-1}} = 0$; deci numărul binar $b_{i_{d-1}} \ldots b_{i_1} b_{i_0}$ este egal cu $2^j - 1$. Fie acum o cale între nodurile 0 şi $2^j - 1$ şi fie c un nod de pe acea cale; transformăm această cale într-una între P_s şi P_a, transformând numărul c după cum urmează. Se aplică permutarea inversă lui i biţilor numărului c, deci $c_0 \leftarrow c_{i_0}$, etc. (în

Figura 7.2: Patru căi arc-disjuncte între doi vecini, pe grilă.

acest fel se construieşte o cale între 0 şi b); apoi se face operaţia sau exclusiv $c \leftarrow c \oplus s$, acesta fiind rezultatul. Pentru sursă şi destinaţie, transformările decurg astfel: $0 \stackrel{i^{-1}}{\rightarrow} 0 \stackrel{\oplus s}{\rightarrow} s$ şi $2^j - 1 \stackrel{i^{-1}}{\rightarrow} b \stackrel{\oplus s}{\rightarrow} a$. Transformarea nu e unică, ci depinde de modul de alegere a permutării i. De exemplu, să vedem care este calea de lungime 4 dintre $s = 110$ şi $a = 011$, într-un hipercub de dimensiune 3; $b = 101$ şi putem alege i astfel: $(0,1,2) \stackrel{i}{\rightarrow} (0,2,1)$; în figura 3.10 se observă că între 0 şi $2^2 - 1 = 011$, calea de lungime 4 este 000, 100, 101, 111, 011; aplicând inversa permutării i, aceasta se transformă în 000, 010, 011, 111, 101; în fine, după un sau exclusiv cu $s = 110$ se obţine 110, 100, 101, 001, 011. Dacă se alege $(0,1,2) \stackrel{i}{\rightarrow} (1,2,0)$, atunci aplicând inversa lui i se obţine 000, 010, 110, 111, 101; în final calea este 110, 100, 000, 001, 011.

Observaţie. Este poate mai simplu să se construiască direct căile între P_s şi P_a. Se consideră mulţimea $\mathcal{D}$ a poziţiilor în care s şi a diferă şi se construiesc j căi de lungime j prin modificări ale biţilor din acest grup (rotaţii, în $\mathcal{D}$, ale căii principale prin $\mathcal{D}$). Cele $d - j$ căi de lungime $j + 2$ se construiesc negând câte un bit care nu e în $\mathcal{D}$, mergând apoi pe calea principală prin $\mathcal{D}$, şi negând la loc bitul iniţial.

P3.3.6 a) Fiecare cale are un singur nod în C_k, obţinut prin rotaţia lui $u_k = 0 \ldots 0 \underbrace{1 \ldots 1}_{k}$; aceste d noduri sunt evident diferite. b) Cele j căi obţinute prin rotirea căii principale nu au noduri comune. Nodurile de pe calea l, cu $l \geq j$, au bitul l egal cu 1, şi diferă prin aceasta de toate celelalte noduri.

P3.3.7 În C_k sunt $\binom{d}{k}$ noduri; pentru fiecare nod, k arce merg către C_{k-1} şi $d - k$ arce către C_{k+1}. Deci sunt $\binom{d}{k}(d-k) = \binom{d-1}{k} d$ arce între C_k şi C_{k+1}.

P3.3.8 Difuzare, full duplex: primele Δ emise de P_s ajung la cel mai depărtat procesor cel puţin după timpul $D(\sigma + \beta)$, iar restul de $m - \Delta$, după cel puţin încă $\frac{m-\Delta}{\Delta}\beta$; se obţine aceeaşi margine ca şi în cazul comunicaţiei între două procesoare aflate la distanţă egală cu diametrul grafului.

Difuzare, half duplex: graful are cel mult $(p\Delta)/2$ arce folosite unidirecţional la un moment dat; numărul minim de valori care trebuie transmise global este $(p-1)m$, mesajul trebuind să ajungă la toate celelalte procesoare; presupunând toate legăturile folosite permanent, se obţine marginea inferioară cerută.

Difuzare generală: marginile sunt obţinute din condiţii de ocupare a arcelor grafului. Numărul minim total de valori care trebuie transmise este $p(p-1)m$ (fiecare procesor trimite m valori pentru celelalte $p - 1$ procesoare; pentru a ajunge la câte un procesor, un mesaj trebuie să străbată cel puţin câte un arc), iar numărul maxim de mesaje care pot fi transmise simultan este de $p\Delta$ în cazul full duplex şi $(p\Delta)/2$ în modelul half duplex.

Distribuţie: procesorul care distribuie trebuie să transmită $(p-1)m$ valori pe cele Δ

canale ale sale.

Schimb complet, inel: la distanţă i de un procesor se află două procesoare. Numărul minim de arce parcurse de mesajele emise de procesor este $\sum_{i=1}^{p/2} 2i = \frac{p}{2}(\frac{p}{2}+1)$. Pentru toate procesoarele, numărul total minim de treceri pe canale este deci $\frac{p^2}{2}(\frac{p}{2}+1)$. Dacă toate cele $2p$ canale, în regim full duplex, respectiv p în half duplex, sunt folosite simultan, rezultă marginile din textul problemei.

Schimb complet, hipercub: la distanţă k de un procesor se află $\binom{d}{k}$ procesoare; cele $(p-1)$ mesaje emise de către un procesor, spre fiecare dintre celelalte, au de parcurs cel puţin $\sum_{k=1}^{d} k\binom{d}{k} = \frac{d}{2}p - 1$ canale (pentru calculul sumei, vezi problema 3.3.24), deci în total se parcurg cel puţin $(dp^2)/2$ canale. Presupunând canalele folosite în paralelism complet, se obţin marginile din textul problemei.

P3.3.9 Deoarece din rădăcină pornesc Δ arce, se pot construi cel mult Δ arbori de acoperire. Graful are $p\Delta$ arce (orientate); cei Δ arbori ar avea în total $\Delta(p-1)$ arce, deci nici o contradicţie.

P3.3.10 P_0 difuzează mesajul M pe inel, după arborele de acoperire din figura 3.11:

1. **dacă** id $= 0$ **atunci**
    1. **în paralel** **send**(M, dreapta), **send**(M, stânga)
2. **altfel dacă**id $\leq \lfloor p/2 \rfloor$ **atunci**
    1. **recv**(M, stânga)
    2. **dacă** id $\neq \lfloor p/2 \rfloor$ **atunci** **send**(M, dreapta)
3. **altfel**
    1. **recv**(M, dreapta)
    2. **dacă**id $\neq \lfloor p/2 \rfloor + 1$ **atunci** **send**(M, stânga)

P3.3.11 Toate expresiile e cu care este comparat id se modifică în $(s+e) \bmod p$. Evident că se pot face simplificări; de exemplu, în instrucţiunea 2.1 din algoritmul 3.3, $(s+p-1) \bmod p \equiv (s-1) \bmod p$.

P3.3.12 Algoritmul pentru procesorul P_{xy} este:

1. **dacă** $x = 0$ şi $y = 0$ **atunci**
    1. **în paralel** **send**(M, dreapta), **send**(M, stânga), **send**(M, sus), **send**(M, jos)
2. **altfel dacă** $y = 0$ **atunci**
    1. **dacă** $x \leq \lfloor \sqrt{p}/2 \rfloor$ **atunci**
        1. **recv**(M, stânga)
        2. **în paralel** **send**(M, jos), **send**(M, sus),
            1. **dacă** $x \neq \lfloor \sqrt{p}/2 \rfloor$ **atunci** **send**(M, dreapta)
    2. **altfel**
        1. **recv**(M, dreapta)
        2. **în paralel** **send**(M, jos), **send**(M, sus),
            1. **dacă** $x \neq \lfloor \sqrt{p}/2 \rfloor + 1$ **atunci** **send**(M, stânga)
3. **altfel** $\{y \neq 0\}$
    1. **dacă** $y \leq \lfloor \sqrt{p}/2 \rfloor$ **atunci**
        1. **recv**(M, jos), **dacă** $y \neq \lfloor \sqrt{p}/2 \rfloor$ **atunci** **send**(M, sus)
    2. **altfel**
        1. **recv**(M, sus), **dacă** $y \neq \lfloor \sqrt{p}/2 \rfloor + 1$ **atunci** **send**(M, jos)

P3.3.13 Adresele nodurilor adiacente arcului diferă în bitul k; după rotaţia cu i biţi la stânga (prin care se obţine $\mathcal{A}_i(0)$), ele vor diferi în bitul $(k+i) \bmod d$.

P3.3.14 În etapa l, unde l este poziţia celui mai semnificativ bit de 1 din j. În algoritmul de difuzare rotativ, mesajul M^i va fi primit în etapa l_i, unde l_i este poziţia celui mai semnificativ bit de 1 din $\mathrm{Ro}(j, -i)$; explicaţie: prin rotaţia cu i biţi la dreapta se transformă nodurile din $\mathcal{A}_i(0)$ în noduri din $\mathcal{A}_0(0)$; ori, pentru acest arbore, ştim deja răspunsul: l_i.

P3.3.15 Algoritmul de difuzare rotativ are forma următoare:

1. **pentru** $i = 0 : d-1$
   1. $l_i \leftarrow$ poziţia celui mai semnificativ bit de 1 din $\mathrm{Ro}(\mathrm{id}, -i)$ (-1 când $\mathrm{id} = 0$)
2. **pentru** $k = 0 : d-1$
   1. **în paralel**
      1. **pentru** $i = 0 : d-1$, **în paralel**
         1. **dacă** $l_i = k$ **atunci recv**(M^i, $(l_i + i) \bmod d$)
      2. **pentru** $i = 0 : d-1$, **în paralel**
         1. **dacă** $l_i < k$ **atunci send**(M^i, $(k+i) \bmod d$)

Se impun unele explicaţii, dată fiind forma condensată a algoritmului; k este etapa de comunicaţie; în fiecare etapă, fiecare procesor poate efectua în paralel, atât recepţii, în 2.1.1, cât şi transmisii, în 2.1.2. Condiţia din 2.1.1.1 este justificată în rezolvarea problemei precedente: mesajul M^i soseşte în etapa l_i. Canalul pe care soseşte M^0 este l_0 (folosind arborele $\mathcal{A}_0(0)$); canalul pe care vine M^i se obţine prin rotaţie, deci este în dimensiunea $(l_i + i) \bmod d$ (se foloseşte $\mathcal{A}_i(0)$; vezi şi problema 3.3.13). După ce a primit un mesaj, un procesor îl emite în toate etapele, până la sfârşit, ceea ce se vede în condiţia din 2.1.2.1, pe canalele din dimensiunile succesive celei pe care a fost primit (adică $(l_i + i + k - l_i) \bmod d = (k+i) \bmod d$).

P3.3.16 Se obţine algoritmul 3.2 de comunicare după d căi disjuncte (vezi fig. 3.10).

P3.3.17 Timpul de difuzare este chiar D. Iniţiatorul difuzării îşi informează vecinii, apoi aceştia vecinii lor, etc. La momentul k, toate nodurile la distanţă $\leq k$ au primit mesajul.

P3.3.18 În prima etapă, iniţiatorul difuzării comunică mesajul unui vecin oarecare. Apoi, ambele noduri informează alte două noduri; astfel numărul nodurilor care au primit mesajul se dublează la fiecare etapă. Timpul necesar este deci $\lceil \log p \rceil$.

Aceasta este şi limita inferioară pentru un graf oarecare, care are mai puţine arce decât graful complet. De asemenea, este şi timpul necesar pe hipercub, în algoritmul 3.4. Deci, în modelul timp constant, 1-port, acest algoritm este optim (o bună dovadă a calităţilor hipercubului, care are mult mai puţine arce decât graful complet).

P3.3.19 În modelul multiport, full duplex, timpul este 1 (fiecare comunică cu fiecare în ambele sensuri). În half duplex, timpul este 2; de exemplu, un procesor colectează simultan toate mesajele, apoi le difuzează; dacă nu vă place ideea de a difuza un mesaj atât de lung (deşi în timp constant nu contează lungimea), se poate şi altfel: în prima etapă toate perechile P_i, P_j comunică în acest sens dacă $i < j$; în a doua etapă se comunică în sens invers.

În 1-port, full duplex, timpul este $\lceil \log p \rceil$, ca pentru hipercub (algoritmul sugerat în figura 3.21). În half duplex, un timp de două ori mai mare este obţinut imediat. Se poate însă şi mai bine, dar cu complicaţii; până acum se ştie că timpul se încadrează între limitele $1.44\lceil \log p \rceil$ şi $1.88\lceil \log p \rceil$; prima valoare se poate demonstra ca limită teoretică (dar pe vreo două pagini); a doua este timpul de difuzare generală pe hipercub, după un algoritm complicat.

P3.3.20 Se transmite pe o singură direcţie, spre dreapta, să zicem.

1. **pentru** $k = 0 : p-2$
   1. **în paralel** **send**($M_{(\mathrm{id}-k) \bmod p}$, dreapta), **recv**($M_{(\mathrm{id}-k-1) \bmod p}$, stânga)

P3.3.21 Pe un graf oarecare, P_s trimite toate mesajele tuturor vecinilor; aceştia reţin mesajul lor şi transmit restul mai departe; după k paşi, toate procesoarele aflată la distanţă k de P_s şi-au primit mesajul. Timpul este deci D, diametrul grafului (mai exact, excentricitatea nodului P_s).

În modelul 1-port, pe inel, se foloseşte arborele de acoperire din figura 3.11. Se transmit întâi toate mesajele spre dreapta, apoi toate spre stânga, după care fiecare procesor retransmite mai departe. Când p este par, ramurile arborelui sunt inegale, deci timpul va fi $p/2$. Când p este impar, timpul va fi $(p+1)/2$.

Pe hipercub, algoritmul este cel deja prezentat în figura 3.23.

P3.3.22 Mesajul pentru procesorul P_{xy} este trimis întâi pe orizontală, dacă $x+y$ este par, şi întâi pe verticală dacă $x+y$ este impar. De exemplu, mesajul pentru procesorul $(2,2)$ este trimis lui $(2,0)$, împreună cu toate mesajele pentru care $x=2$ şi $x+y$ este par; apoi procesorul $(2,0)$ distribuie pe verticala sa. În schimb, mesajul pentru $(2,1)$ este trimis lui $(0,1)$, împreună cu toate mesajele pentru care $y=1$ şi $x+y$ e impar; procesorul $(0,1)$ distribuie apoi pe orizontala sa.

P3.3.23 E suficient să demonstrăm aceasta pentru distribuţie. Dar distribuţia decurge pe arborele din figura 3.13, construit pe prinicipiul căii celei mai scurte (e intuitiv: se merge întâi pe orizontală pe drumul cel mai scurt, apoi la fel pe verticală).

P3.3.24 Considerăm suma termenilor egali distanţaţi de limitele de sumare: $i\binom{d}{i} + (d-i)\binom{d}{d-i} = i\binom{d}{i} + (d-i)\binom{d}{i} = d\binom{d}{i}$. Deci $\sum_{i=0}^{d} i\binom{d}{i} = (d/2)\sum_{i=0}^{d}\binom{d}{i} = d2^{d-1}$.

P3.3.25 a), b) Transpunerea se face printr-un schimb complet, indiferent de topologie, deoarece procesorul P_i trebuie să trimită blocul A_{ij} procesorului P_j, pentru orice i şi j.

c) Se poate proceda astfel: fiecare procesor "diagonal" P_{kk} colectează mesajele de la procesoarele de pe linia sa, P_{kj}, cu $0 \le j < \sqrt{p}$. Apoi, P_{kk} distribuie pe coloana sa, procesoarelor P_{ik}, cu $0 \le i < \sqrt{p}$. Cum colectarea şi distribuţia au acelaşi timp de execuţie şi se desfăşoară pe un inel, timpul total va fi $\sqrt{p}(\sigma + m\beta)$ (m fiind numărul de elemente dintr-un bloc).

Acum se poate aplica îmbunătăţirea clasică; fiecare mesaj se împarte în două pachete, trimise unul pe orizontală, altul pe verticală (P_{ij} trimite primul pachet lui P_{ii} şi pe al doilea lui P_{jj}). Procesoarele diagonale distribuie pe verticală primul pachet şi pe orizontală pe cel de-al doilea. Se înjumătăţeşte timpul de propagare de mai sus.

S-ar părea că mai bine nu se poate. Dar, să observăm că procesorul P_{ij} schimbă un mesaj cu P_{ji}, deci cu exact un alt procesor; transpunerea pe tor este deci o operaţie de transmisie permutată. Ea va dura cât două schimburi complete cu mesaje de lungime m/p, adică, dacă luăm cel mai eficient algoritm de schimb complet, $2\sqrt{p}\,\sigma + (\sqrt{p}/4)\beta$.

P3.3.26 În modelul multiport, timpul este 1; fiecare procesor trimite pe fiecare canal mesajul potrivit, recepţionând în acelaşi timp. În modelul 1-port, transmite (şi recepţionează) pe rând pe câte un canal, deci timpul este $p-1$.

P3.5.1 Un algoritm simplu poate fi schiţat după cum urmează, procesorul destinaţie fiind P_{xy}: **dacă** $\mathrm{id}_x \neq x$ şi $\mathrm{id}_y \neq y$ **atunci** co va fi canalul opus lui ci; **dacă** $\mathrm{id}_x = x$ sau $\mathrm{id}_y = y$ **atunci** se alege co, canalul (unic) care apropie de destinaţie. Prima regulă poate să nu fie aplicabilă în cazul unui procesor de la periferie; pentru acestea, se alege orice canal

diferit de ci, care apropie mai mult de destinaţie (pot fi două astfel de canale). Conform acestor reguli se obţin, în general, două căi distincte, sau trei în cazul procesoarelor aflate pe aceeaşi linie sau coloană.

P3.5.2 Desigur. Căile între 000 şi 110 sunt: prima 000, 001, 101, 111, 110, a doua 000, 010, 110, a treia 000, 100, 110.

P3.5.3 Indiferent de canalul de intrare, mesajul pentru P_a este trimis spre P_a; astfel, se obţin o cale de lungime 1 şi $p-2$ căi de lungime 2.

P4.3.1 Produsul scalar $x^T y = \sum_{i=0}^{n-1} x_i y_i$, este în fond tot o sumă, după efectuarea produselor $x_i y_i$.

P4.3.2 $p = O(n/\log n)$. Fiecare procesor efectuează întâi suma a câte $O(n/p) = O(\log n)$ elemente, în timp $O(\log n)$; rămân astfel de sumat câte un element pentru fiecare procesor; aceasta se poate face în timp $O(\log(n/\log n)) = O(\log n - \log\log n) = O(\log n)$. Deci, timpul total de execuţie este $O(\log n)$, iar costul, evident $O(n)$.

P4.3.3 $\sum_{l=0}^{n/2^k-1} x_{i+l2^k}$.

P4.3.4 Funcţia de mai jos calculează suma unor valori întregi, dar generalizarea este banală. Suma este calculată doar de P_0, celelalte procesoare returnând o valoare nesemnificativă. Variabila `k2` este egală cu 2^{k+1} (atenţie: nu se verifică egalitatea lui p cu o putere a lui 2).

```
int suma_hip(int val)
{
  int my_id, p, d, vecin, x;
  int k, k2;
  MPI_Status status;

  MPI_Comm_rank (MPI_COMM_WORLD, &my_id);
  MPI_Comm_size (MPI_COMM_WORLD, &p);
  d = log(p)/log(2) + 0.5;

  for ( k = d-1, k2 = p ; k >= 0 ; k--, k2 >>= 1 )
  {
    vecin = my_id ^ ( 1 << k );
    if ( my_id < (k2>>1) )
    {
      MPI_Recv(&x, 1, MPI_INT, vecin, 0, MPI_COMM_WORLD, &status);
      val += x;
    }
    else if ( my_id < k2 )
      MPI_Send(&val, 1, MPI_INT, vecin, 0, MPI_COMM_WORLD);
  }

  return val;
}
```

P4.3.5 Trebuie modificate condiţia din instrucţiunea **dacă**, din $i < n/2^k$ în $i \bmod 2^k = 0$, şi adaptaţi corespunzător indicii din atribuirea 1.1.1. Algoritmul pentru P_i este:

1. **pentru** $k = 1 : \log n$
    1. **dacă** $i \bmod 2^k = 0$ **atunci**
        1. $x_i \leftarrow x_i + x_{i+2^{k-1}}$

Astfel, în etapa k, rămân în cursă doar procesoarele cu adrese de forma $j2^k$, comunicând cu procesoare $j2^k + 2^{k-1}$, deci vecine în hipercub în dimensiunea $k-1$. Atenţie, algoritmul este scris pentru $p = n$ procesoare, situaţia "normală" în cazul memoriei distribuite (spre deosebire de $p = n/2$ pe PRAM); procesoarele cu adresă impară nu efectuează nici o operaţie. Se va comunica în ordine inversă a dimensiunilor, faţă de algoritmul 4.3, dar tot după un arbore de acoperire binomial, la fiecare pas după o direcţie.

Nu mai este necesară comutativitatea operaţiei +; la pasul k, variabila locală procesorului P_i (cu adresa astfel încât $i \bmod 2^k = 0$) are valoarea $\sum_{l=i}^{i+2^k-1} x_l$, adică suma unor valori x_l cu indici consecutivi.

P4.3.6 Fiecare procesor îşi împarte elementele proprii în d grupuri şi calculează suma pentru fiecare grup. Procesorul P_i are acum d valori: $x_0^i, \ldots, x_{d-1}^i$. Apoi se calculează simultan sumele $s_k = \sum_{l=0}^{p-1} x_k^l$, fiecare comunicând pe un alt arbore din familia de arbori binomiali rotativi. Comunicaţiile decurg simultan, deoarece pentru fiecare s_k se foloseşte la un moment dat o singură dimensiune. Din păcate, P_0 comunică la fiecare etapă pe toate direcţiile şi face d adunări, câte una pentru fiecare s_k; deci timpul consumat de P_0 dictează timpul total. Acesta este: $d[n/(pd) - 1]\alpha$ pentru sumele locale, $d^2\alpha$ pentru sumele efectuate de P_0 pe parcurs (d etape, câte d adunări) şi $d(\sigma + \beta)$ pentru comunicaţie. Totalul este mai mare decât cel pentru algoritmul 4.3, deci ideea arborilor rotativi nu e eficient aplicată astfel. Dacă însă trebuie calculate mai multe sume, cel mult d, şi pentru fiecare se foloseşte un arbore din familie, atunci timpul de comunicaţie va fi identic cu cel din cazul calculului unei singure sume.

P4.3.7 Fiecare procesor calculează suma elementelor locale, apoi o trimite spre P_0, pe drumul cel mai scurt (arborele de acoperire din figura 3.11). P_0, de îndată ce recepţionează două sume, le adună la suma sa locală, continuând să recepţioneze, în paralel; dacă $2\alpha \leq \sigma + \beta$, calculele nu întârzie comunicaţia ! (Desigur, aceasta este valabil în cazul modelului de comunicaţie asincron; totuşi, chiar în modelul sincron, execuţia concurentă a calculelor şi comunicaţiei pe P_0 va reduce posibilele întârzieri intervenite în comunicaţie.) Timpul total va fi $(n/p-1)\alpha$ pentru sumele locale, $(p/2)(\sigma+\beta)$ pentru comunicaţie, şi 2α pentru adunarea ultimelor două sume primite.

Dacă procedam ca la hipercub, efectuând sumele pe parcurs (fiecare procesor primeşte de la un vecin o valoare, o adună la valoarea proprie şi transmite această sumă celuilalt vecin), s-ar fi făcut $(p/2)\alpha$ adunări, nesuprapuse cu comunicaţia.

Ideea colectării în P_0 şi a efectuării calculelor pe măsură ce vin sumele locale, în paralel cu comunicaţia este bună la inel pentru că la orice distanţă de P_0 se află doar 2 procesoare. Pe un hipercub, colectând după un arbore de acoperire echilibrat, ar trebui efectuate mult mai multe calcule în timpul $\sigma + \beta$; dacă comunicaţiile sunt foarte lente, merită încercat.

P4.3.8 Fiecare procesor calculează suma elementelor locale. Apoi, se comunică în $p/2$ etape, ca la difuzarea generală; în prima etapă, un procesor transmite în ambele direcţii suma sa; apoi, pe măsură ce primeşte sumele altor procesoare, le adună la suma sa şi le transmite mai departe, cu excepţia ultimei etape. Practic, aceasta este simetrizarea algoritmului din problema 4.3.7; timpul de calcul este acelaşi; şi acum retransmisia a două sume se poate face în paralel cu adunarea lor la suma locală.

Dacă sunt de calculat m sume, singurul truc aplicabil este de a grupa mesajele aparţinând unor calcule diferite. Astfel timpul de comunicaţie va fi $(p/2)\sigma + (pm/2)\beta$. Coeficientul lui σ este acelaşi ca în cazul unei singure sume.

P4.3.9 Da, bucla **pentru** poate merge doar până la $(i+1)n/p-2$, pentru că $s_{(i+1)n/p-1}$ este chiar z_i.

P4.3.10 Fie $j = i + 2^l$; notăm gi şi gj codurile Gray ale lui i, respectiv j. Reamintim regula de conversie binar–Gray: $gi_k = i_k \oplus i_{k+1}$ (vezi **P1.2.6**). Sunt două cazuri posibile.

a) $j_l = 1$, deci $i_l = 0$ şi atunci $j_k = i_k$ pentru $k \neq l$. Atunci $gj_l = \overline{gi_l}$ şi $gj_{l-1} = \overline{gi_{l-1}}$, iar în rest adresele coincid. Deci distanţa este 2, mai puţin în cazul $l = 0$ (când nodurile sunt vecine în ciclul hamiltonian).

b) $j_l = 0$, deci $i_k = \overline{j_k}$, pentru $k \geq l$; pentru $k < l$, adresele coincid. Atunci $gj_{d-1} = \overline{gi_{d-1}}$, dar $gj_{d-2} = gi_{d-2}$, $\ldots$, $gj_l = gi_l$ (deoarece $a \oplus b = \bar{a} \oplus \bar{b}$); apoi, $gj_{l-1} = \overline{gi_{l-1}}$, $gj_{l-2} = gi_{l-2}$, $\ldots$, $gj_0 = gi_0$; deci, dimensiunile în care diferă gi şi gj sunt $d-1$ şi $l-1$, iar distanţa între noduri este 2, făcând din nou excepţie cazul $l = 0$.

P4.3.11 Instrucţiunile 2.2 şi 2.3 se modifică în:

2.2. **dacă** $\mathrm{id}_k = 1$ **atunci** $s \leftarrow v + s$, $z \leftarrow v + z$
2.3. **altfel** $z \leftarrow z + v$

Explicaţia e următoarea: pentru un procesor dintr-un hipercub "superior" (la un moment dat), suma globală v recepţionată provine dintr-un hipercub "inferior", deci ea trebuie să fie termen stâng în operaţia notată cu $+$, iar variabila locală termen drept. Pentru procesoarele dintr-un hipercub "inferior", lucrurile se petrec pe dos.

P4.3.12 Da, dacă două procesoare calculează în paralel coeficienţii a_i, respectiv b_i. Numărul de operaţii va fi $2 + 2\log n$, cu n procesoare, dar pe un CREW PRAM.

P4.3.13 Păstrăm ipoteza simplificatoare $n = 2^r$; presupunem în plus că $p = 2^s$, deci $n/p = 2^{r-s}$. Pe EREW PRAM, algoritmul pentru procesorul P_i este următorul:

1. **pentru** $k = 1 : r - s - 1$
    1. **pentru** $l = in/p : 2^k : (i+1)n/p - 1$
        1. $a_l \leftarrow a_{l+2^{k-1}} a_l$, $b_l \leftarrow a_{l+2^{k-1}} b_l + b_{l+2^{k-1}}$
2. **pentru** $m = 1 : s + 1$
    1. **dacă** $i < 2^{s+1-m}$ **atunci**
        1. $l \leftarrow in/p$, $k \leftarrow r - s - 1 + m$
        2. $a_l \leftarrow a_{l+2^{k-1}} a_l$, $b_l \leftarrow a_{l+2^{k-1}} b_l + b_{l+2^{k-1}}$
3. **dacă** $i = 0$ **atunci** $x_n \leftarrow a_0^r x_0 + b_0^r$

În prima buclă **pentru**, se reduce numărul elementelor din şir la $2p$; a doua buclă **pentru** este transcrierea algoritmului 4.10.

P4.3.14 Substituţia expresiei termenului x_{i+1} în relaţia de calcul a termenului x_{i+2} conduce la:

$$x_{i+2} = \frac{(a_{i+1}a_i + b_{i+1}c_i)x_i + (a_{i+1}b_i + b_{i+1}d_i)}{(c_{i+1}a_i + d_{i+1}c_i)x_i + (c_{i+1}b_i + d_{i+1}d_i)}$$

Deci dependenţa dintre x_{i+2} şi x_i are aceeaşi formă ca relaţia de recurenţă iniţială. Atunci se poate aplica algoritmul de reducere ciclică 4.10; bineînţeles, se înlocuieşte calculul noilor coeficienţi ai recurenţei cu expresiile ce rezultă din relaţia de mai sus.

P4.3.15 Dată fiind forma sistemului, ecuaţia i are forma $a_{i,i-1}x_{i-1} + a_{ii}x_i = b_i$, pentru $i \geq 2$; deci, $x_1 = b_1/a_{11}$, şi $x_i = (b_i - a_{i,i-1}x_{i-1})/a_{ii}$, pentru $i \geq 2$. Ori, aceste relaţii definesc un şir recurent de ordinul I; practic, este al mod de scriere a relaţiilor de recurenţă, sub formă matriceală.

P4.3.16 Algoritmul pentru procesorul P_i, pe EREW PRAM, n putere a lui 2, $p = n/2$:

1. **pentru** $k = 1 : \log n$
    1. **dacă** $i < n/2^k$ **atunci**
        1. $a_i^k \leftarrow a_{2i+1}^{k-1} a_{2i}^{k-1}$, $b_i^k \leftarrow a_{2i+1}^{k-1} b_{2i}^{k-1} + b_{2i+1}^{k-1}$
2. **dacă** $i = 0$ **atunci** $w_0^{\log n} \leftarrow a_0^{\log n} x_0 + b_0^{\log n}$
3. **pentru** $k = \log n : -1 : 1$
    1. **dacă** $i < n/2^k$ **atunci**
        1. $w_{2i+1}^{k-1} \leftarrow w_i^k$, $w_{2i}^{k-1} \leftarrow a_{2i}^{k-1} w_{i-1}^k + b_{2i}^{k-1}$

Am notat cu w elementele şirului, indicele inferior fiind poziţia în şir, iar cel superior nivelul de recurenţă; convenim că $w_{-1}^k = 0$; rezultatul este $w_i^0 \equiv x_{i-1}$. În prima buclă sunt calculaţi doar coeficienţi, iar în a doua, doar elemente ale şirului; această a doua buclă trebuie amorsată prin calculul ultimului element $x_n \equiv w_0^{\log n}$.

P4.3.17 Presupunem că un procesor P_i are o singură pereche de coeficienţi, notaţi a, b; în aceste variabile, procesorul va calcula coeficienţii pentru care $x_{i+1} = ax_0 + b$, aplicând de mai multe ori formule de tipul (4.5). În plus, fiecare procesor mai are o pereche de coeficienţi, notaţi c şi d, pentru care, în final, $x_n = cx_0 + d$. Deci, ca şi în algoritmul 4.9, un procesor calculează coeficienţii necesari doar lui (echivalentul sumelor prefixelor), dar şi coeficienţii necesari ultimului termen (echivalentul sumei totale). Algoritmul este următorul:

1. **pentru** $k = 0 : d - 1$
    1. **în paralel** **recv**$((a', b'), k)$, **send**$((c, d), k)$
    2. **dacă** $\mathrm{id}_k = 1$ **atunci**
        1. $b \leftarrow ab' + b$, $a \leftarrow aa'$
    3. $d \leftarrow cb' + d$, $c \leftarrow ca'$

Ca şi în cazul algoritmului 4.9, e de aşteptat o eficienţă relativă sub 1/2, deoarece toate operaţiile sunt dublate.

P4.3.18 La fiecare pas, procesorul P_i recepţionează două valori din x şi actualizează z_i ca în instrucţiunea 2.4 din algoritmul următor (corect doar pentru $p = n$ impar, dar uşor adaptabil şi pentru p par):

1. $z_i \leftarrow y_i + a_{ii}x_i$, $j \leftarrow i$, $l \leftarrow i$
2. **pentru** $k = 0 : \lfloor p/2 \rfloor - 1$
    1. **în paralel**
        1. **send**(z_j, stânga)
        2. **recv**($z_{(j+1) \bmod n}$, dreapta)
        3. **send**(z_l, dreapta)
        4. **recv**($z_{(l-1) \bmod n}$, stânga)
    2. $j \leftarrow (j + 1) \bmod n$
    3. $l \leftarrow (l - 1) \bmod n$
    4. $z_i \leftarrow z_i + a_{ij}x_j + a_{il}x_l$

P4.3.19 Procesorul P_i are în memorie linia i din A, memorată sub forma unui vector a, şi elementele x_i, y_i, memorate în variabilele scalare x, y. Rezultatul z_i este memorat în variabila scalară z.

1. $z \leftarrow y$, $j \leftarrow$ id
2. **pentru** $k = 0 : p-1$
   1. $z \leftarrow z + a_j x$
   2. **dacă** $k < p-1$ **atunci**, **în paralel**
      1. **send**(x, stânga)
      2. **recv**(u, dreapta)
   3. $x \leftarrow u$
   4. $j \leftarrow (j+1) \bmod p$

P4.3.20 În notaţie MATLAB, liniile pe care le posedă procesorul P_i, presupunând o repartizare bloc-linii, sunt $l = ip : (i+1)p-1$; datele locale lui P_i sunt $A(l,:)$, $x(l)$, $y(l)$. La fiecare pas se comunică n/p elemente din x şi se actualizează n/p elemente din z; vom nota cu w variabila locală care găzduieşte elementele din x (un vector de dimensiune n/p) şi cu c indicii corespunzători lui w în x. Cu aceste notaţii, algoritmul pentru P_i este următorul:

1. $z(l) \leftarrow y(l)$, $j \leftarrow i$, $c \leftarrow l$, $w \leftarrow x(c)$
2. **pentru** $k = 0 : p-1$
   1. $z(l) \leftarrow z(l) + A(l,c)w$
   2. **dacă** $i < p-1$ **atunci**, **în paralel**
      1. **send**(w, stânga)
      2. **recv**(w', dreapta)
   3. $j \leftarrow (j+1) \bmod n$
   4. $c \leftarrow jp : (j+1)p-1$ {indicii elementelor din x}
   5. $w \leftarrow w'$ {acum $w = x(c)$}

În cazul în care n nu se divide cu p, ultimul procesor are repartizate mai puţine linii. Pentru a descrie aceasta, e suficient a se înlocui, în expresia lui l, $(i+1)p-1$ cu $\min((i+1)p-1, n)$. De asemenea, pentru ca înmulţirile să fie efectuate corect, în expresia lui c se înlocuieşte $(j+1)p-1$ cu $\min((j+1)p-1, n)$. (O altă variantă, aproape echivalentă, e de a completa cu zerouri matricea A şi vectorii x, y, până la cel mai apropiat multiplu de p.)

P4.3.21 Indiferent de relaţia între m şi n, numărul de operaţii efectuate de un procesor este acelaşi, în algoritmii 4.16 şi 4.18, repartizarea calculelor între procesoare fiind echilibrată. În ce priveşte comunicaţia, lucrurile nu mai stau aşa. În algoritmul 4.16 se comunică elementele lui x, în fiecare dintre cele $p-1$ etape de comunicaţie câte n/p (x are dimensiune n), ceea ce conduce la o complexitate a comunicaţiei de aproximativ $p\sigma + n\beta$. În algoritmul 4.18 se comunică elementele lui z, în fiecare dintre cele $p-1$ etape câte m/p (z are dimensiune m), deci, în total, $p\sigma + m\beta$. Acum devine clar care dintre algoritmi e mai bun: dacă $m > n$, atunci algoritmul 4.18 e mai rapid, dacă $m < n$, atunci 4.16.

P5.1.1 Algoritmul secvenţial nu face nici o interschimbare de elemente, folosind o singură locaţie suplimentară de memorie (cea pentru minim); un algoritm paralel fără interschimbări ar avea nevoie de p locaţii suplimentare (de ce ?).

P5.1.2 Bucla în i a algoritmului 5.1 va fi "spartă" în mai multe bucle de dimensiune p:

1. **pentru** $i = 1 : \log n$
   1. $j \leftarrow k$
   2. **cât timp** $j < n/2^i$
      1. $exch(2^i j, 2^{i-1}(2j+1)), \quad j \leftarrow j + p$

Presupunând $p = 2^r$ se obţine:

$$T(n) = \frac{n}{2p} + \frac{n}{4p} + \ldots + \frac{n}{2^{\log n-(r-1)}} + \underbrace{1 + \ldots + 1}_{\log p} = \frac{n}{p} + \log p - 1$$

şi deci $\varepsilon(n,p) = \dfrac{n-1}{n-1+p\log p}$. Dacă $p \log p \ll n$ atunci $\varepsilon(n,p) \approx 1$.

O variantă mai simplă este aceea în care fiecare procesor calculează minimul între n/p elemente, apoi, pentru cele p elemente rămase, se aplică algoritmul 5.1.

P5.1.3 În algoritmul 5.1 minimul rămâne între elementele cu indice divizibil cu 2^i, după pasul i. Pentru a rămâne între primele 2^i elemente:

1. **pentru** $i = \log n - 1 : -1 : 0$ {se inversează ordinea aici}
   1. **dacă** $k < n/2^{\log n - i}$ **atunci** $exch(k, k + 2^i)$

P5.1.4 De exemplu: $c_{(i+j) \bmod n} \leftarrow 1$ sau $c_{i \oplus j} \leftarrow 1$; ultimul mod e corect doar pentru p putere a lui 2.

P5.1.5 a) Da b) Da; dacă sunt mai multe elemente cu valoare minimă, în cazul testului $a_i > a_j$ va fi selectat cel cu indicele cel mai mare, iar în cazul testului $a_i \geq a_j$ cel cu indicele cel mai mic.

P5.1.8 Cu cel din problema **P5.1.3** (se elimină procesoarele cu numere mari). Pentru a semăna cu celălalt algoritm se inversează ordinea în buclă: **pentru** $i = 0 : d - 1$.

P5.2.1 Citirea valorilor c_k şi c_{k-1} se face în doi paşi în loc de unul. În plus, trebuie presupus că fiecare procesor cunoaşte valoarea v, altfel e necesară difuzarea acesteia, în $\log n$ etape. Numărul de comparaţii rămâne însă tot 1.

P5.2.2 Se face difuzarea valorii v, dacă nu e deja cunoscută de toate procesoarele. Fiecare procesor face o căutare binară locală cu $\log m$ comparaţii. Un procesor P_k poate întâlni trei situaţii: $v > A_k$, $v < A_k$ sau v între două elemente din A_k; să codificăm aceste situaţii cu 0, 1, 2. Fiecare procesor trimite codul său vecinului din dreapta (P_k lui P_{k+1}). Un singur procesor P_j va difuza valoarea rangului; fie unicul care posedă codul 2 şi atunci rangul e $jm+r$, r fiind rangul local; fie cel care deţine codul 1 şi primeşte din stânga codul 0 şi atunci rangul e jm (completaţi algoritmul cu acţiunile suplimentare ale procesoarelor P_0 şi P_{p-1}).

P5.2.3 Algoritmul este identic cu 5.5, cu diferenţa că evaluarea rangului local se face parcurgând secvenţa A_k de la un capăt la celălalt, deci cu n/p comparaţii în loc de $\log(n/p)$.

P5.2.4 Desigur că s-ar putea copia elementele mai mici decât v într-o zonă contiguă de memorie, la fel cele mai mari, apoi se va recopia totul în A. Se poate face însă totul pe loc în A, după cum urmează (presupunem că pivotul este a_s):

procedură *partiţionare*(s, d, v, i_v)
1. $l \leftarrow s + 1$, $u \leftarrow d$, $v \leftarrow a_s$
2. **cât timp** $l < u$
   1. **cât timp** $a_l < v, \quad l \leftarrow l + 1$
   2. **cât timp** $a_u \geq v, \quad u \leftarrow u - 1$

3. **dacă** $l < u$ **atunci** $a_l \leftrightarrow a_u$, $l \leftarrow l+1$, $u \leftarrow u-1$

3. $i_v \leftarrow u$, $a_s \leftrightarrow a_u$

Cei doi indici demarchează zona în care elementele nu au fost încă grupate; la stânga lui l sunt elemente mai mici decât v, la dreapta lui u, elemente mai mari; l este crescut până la găsirea primului element mai mare decât v; u este micşorat până la găsirea unui element mai mic decât v; apoi aceste două elemente sunt interschimbate. În final, u indică poziţia celui mai din dreapta element mai mic decât pivotul; în această poziţie este adus pivotul, prin interschimbare cu a_s. Numărul de comparaţii este $d-s$.

Acest algoritm este foarte secvenţial. Algoritmii paraleli de partiţionare (pentru arhitecturi cu memorie comună) folosesc copierea în altă zonă.

P5.2.5 Să considerăm un mic exemplu, în care două procesoare deţin secvenţele $\langle 0,1,1,2\rangle$, respectiv $\langle 0,0,1,3\rangle$. Rangul lui 0 este 0, rangul lui 1 este 3. Ce se întâmplă dacă dorim aflarea elementului cu rangul 2 şi aplicăm algoritmul 5.9 ? Iteraţia 1: se propun candidaţii 1 şi 0, este ales 0, care are rang $0 < 2$; spaţiile de căutare se reduc la $\langle 1,1,2\rangle$ şi $\langle 1,3\rangle$. Iteraţia 2: ambii candidaţi sunt 1, rangul este $3 > 2$; spaţiile de căutare se reduc la mulţimea vidă şi algoritmul nu a găsit soluţia. Explicaţia este simplă: valoarea cu rangul 2 nu există, dacă definim rangul unei valori ca fiind egal cu numărul de elemente strict mai mici decât acea valoare. Ar trebui să rămânem la definiţia rangului ca poziţie în secvenţa sortată, dar aceasta nu ne este de mare ajutor din punct de vedere al calculului; P_1 din exemplul de mai sus n-are de unde şti ce rang local să atribuie valorii 0: 0 sau 1 ?

Poate că am divagat un pic. Iată o soluţie. Să definim rang *minim* r al unei valori v, numărul de elemente din A strict mai mici decât v (ceea ce am numit simplu rang până acum) şi rang *maxim* R numărul de elemente din A mai mici sau egale cu v (rangul primului element strict mai mare decât v în secvenţa A sortată). Să presupunem că, în algoritmul 5.9, se calculează şi r şi R. Atunci condiţia de terminare este $r \le j \le R-1$, ceea ce se explică foarte simplu: în secvenţa sortată A, v apare prima dată în poziţia r şi ultima dată în poziţia $R-1$, între aceste poziţii fiind numai elemente cu valoarea v.

Calculul celor două ranguri în algoritmul 5.5 nu dublează decât partea de comunicaţie şi calculul sumei globale. Găsirea rangurilor locale este simplă; după stabilirea rangului minim prin căutare binară, cel maxim se determină prin căutarea primului element mai mare decât v la dreapta rangului minim (inclusiv acesta).

În fine, o concluzie mai filosofică: găsirea rangului şi selecţia nu sunt una inversa celeilalte decât dacă elementele din A sunt distincte.

P5.2.6 Alegerea candidatului local, la paşii (1) şi (6), se face luând mediana unui mic eşantion sau un element oarecare. Alegerea globală a candidatului decurge la fel (paşii (2) şi (3)). P_0 difuzează candidatul global. Fiecare procesor aplică algoritmul de partiţionare din **5.2.4**, calculând şi rangul local al lui v; abia acum se continuă cu calculul rangului global (pasul (4), mai puţin difuzarea lui v, vezi algoritmul 5.5). La pasul (5), reducerea spaţiului de căutare se face prin eliminarea uneia din cele două părţi ale secvenţei locale, mai mică sau mai mare decât candidatul după cum e cazul.

Complexitatea comunicaţiei rămâne aceeaşi, adică $O(p \log n)$, iar numărul de comparaţii va fi $O(n/p)$ (se raţionează ca la algoritmul 5.8).

În nici un caz nu se face sortarea locală, urmată de aplicarea algoritmului 5.9 ca atare. Aceasta înseamnă $O(n/p \log(n/p))$ operaţii (sortarea este cea mai costisitoare).

P5.3.1 Notăm $D = A \perp B$; notăm i indicele în A, j indicele în B şi l indicele în D.

1. $i \leftarrow 0$, $j \leftarrow 0$, $l \leftarrow 0$

2. **cât timp** $i \leq n$ şi $j \leq m$
   1. **dacă** $a_i < b_j$ **atunci** $d_l \leftarrow a_i$, $i \leftarrow i+1$
   2. **altfel** $d_l \leftarrow b_j$, $j \leftarrow j+1$
   3. $l \leftarrow l+1$
3. **cât timp** $l \leq n+m$ {adaugă un sfârşit de secvenţă}
   1. **dacă** $i \leq n$ **atunci** $d_l \leftarrow a_i$, $i \leftarrow i+1$
   2. **altfel** $d_l \leftarrow b_j$, $j \leftarrow j+1$
   3. $l \leftarrow l+1$

P5.3.2 La primul pas de recursie am văzut că sunt suficiente $\sqrt{nm}$ procesoare. La al doilea pas, problema de interclasare este spartă în $q = \sqrt{n}$ probleme, fiecare de dimensiune n_i, m_i, unde $n_i = \lfloor\sqrt{n}\rfloor$ sau $n_i = \lceil\sqrt{n}\rceil$; despre fiecare m_i în parte nu ştim nimic; în schimb $\sum_{i=1}^{q} n_i = n$, $\sum_{i=1}^{q} m_i = m$. Rezolvarea fiecărei subprobleme necesită $\sqrt{n_i}\sqrt{m_i}$ procesoare. Trebuie să demonstrăm că

$$\sum_{i=1}^{q} \sqrt{n_i}\sqrt{m_i} \leq \sqrt{n}\sqrt{m}$$

Dar aceasta este chiar inegalitatea Cauchy-Buniakowski-Schwarz pentru numerele $\sqrt{n_i}$, $\sqrt{m_i}$.

P5.3.4 Un procesor care termină de interclasat secvenţele sale îl poate ajuta pe un altul, de exemplu pe cel care are cea mai lungă secvenţă B'_i şi nu este încă ajutat; el începe să interclaseze începând de la coadă (şi cu asta am sugerat răspunsul întrebării puse la începutul secţiunii). Evident că aşa ceva nu-i deloc uşor de programat; deci nu e eficient, probabil.

P5.3.5 Sunt $rang(a_i^*)$ elemente mai mici decât a_i^* în secvenţa $A^* \perp B^*$ şi sunt i elemente mai mici decât a_i^* în A^*. Deci, din cele $rang(a_i^*)$ mai mici în $A^* \perp B^*$, $rang(a_i^*) - i$ aparţin lui B^*. Cum $lp(a_i^*)$ este indicele celui mai mare element din B^* mai mic decât a_i^*, atunci $lp(a_i^*) = rang(a_i^*) - i - 1$ (al j-lea element din B^* este b_{j-1}^*), de unde (5.2).

P5.3.6 Sunt n valori mai mari sau egale cu a_x ($n-1-x$ în A şi $n-y = x+1$ în B) (a_x este cea mai mare valoare în $A_0 \perp B_0$, pentru că $b_{y-1} \leq a_x$). Este de subliniat că *valoarea* a_x este mediana, şi nu *elementul* a_x este cel median (dacă $a_{x_1} = a_{x_2}$, doar unul va fi în poziţie mediană în $A \perp B$).

P5.3.7 Procesorul P_k compară a_k cu b_{n-1-k} şi scrie în c_k 0 sau 1 după cum a_k este mai mic sau mai mare; C este un vector logic, de n elemente; în el se vor afla j valori de 0 consecutive, urmate de valori de 1, până la sfârşit. Mediana este a_{j-1} sau b_{n-j} şi se va găsi în m.

1. **dacă** $a_k < b_{n-1-k}$ **atunci** $c_k \leftarrow 0$
2. **altfel** $c_k \leftarrow 1$
3. **dacă** $c_k \neq c_{k-1}$ **atunci** $m \leftarrow \min(a_{k-1}, b_{n-k})$

Al doilea **dacă** este executat de toate procesoarele, dar unul singur va face atribuirea ce urmează. Algoritmul se adaptează imediat pe un EREW PRAM.

P5.3.8 Da. Dacă, de exemplu, $a_{x_0} < b_{y_0}$ şi $a_{x_1} = b_{x_1}$, atunci, după instrucţiunea (4), $d = 1$, $c_0 = 0$, $c_1 = 1$. În acest caz şi P_0, în instrucţiunea (6), şi P_1, în instrucţiunea (5), modifică la, ua, lb, ub. Şi totuşi algoritmul funcţionează corect în continuare. Dacă vrem neapărat (sau pentru eleganţă...) să evităm acest conflict procedăm astfel: mutăm instrucţiunea (5) imediat după (3), la acelaşi nivel; tot acolo adăugăm o instrucţiune **stop**, pentru că, în fond, căutarea s-a terminat.

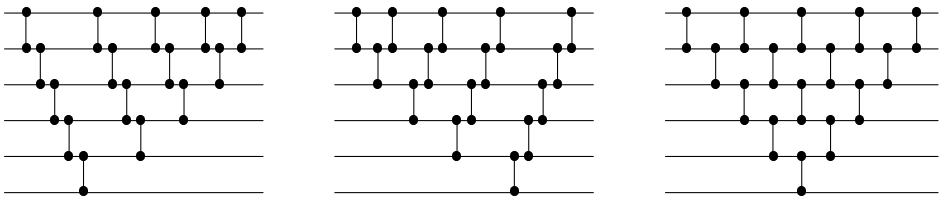


Figura 7.3: Reţele de sortare inspirate de (a) bubblesort, (b) sortarea prin inserţie. (c) O redesenare arată că reţelele sunt identice.

P5.3.9 Fie $Z = X \perp Y = \langle z_0, \ldots, z_{m+n-1} \rangle$. Demonstrăm prin inducţie după indicele în Z. Argumentăm numai pentru X; prin simetrie totul este valabil şi pentru Y.

Dacă $z_0 = x_0$, atunci $rp(x_0) = 0$, deci $0 + rp(x_0) = 0$.

Dacă $z_k = x_l$ atunci fie $z_{k+1} = x_{l+1}$, şi atunci $rp(x_{l+1}) = rp(x_l)$, fie $z_{k+1} = y_r$, şi atunci $rp(x_l) = r$, $rp(y_r) = l + 1$.

P5.3.10 Demonstrăm numai primul caz. O k-separare constă de fapt în găsirea celor mai mici k elemente din $A \perp B$; acestea se află toate în $A_{[k]} \perp B_{[k]}$, pentru că a_i, cu $i \geq k$ are deja k elemente din A mai mici decât el, iar b_i, cu $i \geq k$ are k elemente din B mai mici. În plus, $A_{[k]} \perp B_{[k]}$ are $2k$ elemente.

P5.3.11 Se poate proceda în două feluri, teoretic echivalente. Primul procesor primeşte elementul celuilalt, îl compară cu al său, păstrează minimul şi trimite maximul celuilalt. Sau, procesoarele îşi transmit, pe rând, elementele şi apoi amândouă fac comparaţia. Pentru operaţia cu secvenţe, $exch(A_i, A_j)$, prima variantă este mai slabă.

P5.3.12 Da, dar cu două procesoare. Fiecare face o comparaţie şi o atribuire, în paralel.

P5.4.1 Reţelele sunt prezentate în figura 7.3. Surpriza este că este vorba de aceeaşi reţea, aşa cum se vede după redesenare. Se observă că, într-o reţea, un comparator poate glisa la stânga şi la dreapta atâta vreme cât nu se "ciocneşte" de un altul. Timpul de execuţie este $2n - 3$, cu grad de paralelism cel mult $n/2$ şi folosind $\binom{n}{2}$ comparatoare, adică minimul necesar.

P5.4.2 Cazul $p = \lfloor n/2 \rfloor$, pentru procesorul P_k:

1. **pentru** $i = 0 : n - 1$
    1. **dacă** k este par **atunci** $exch(2k, 2k + 1)$
    2. **altfel dacă** $k \neq \lceil n/2 \rceil$ **atunci** $exch(2k + 1, 2k + 2)$

Pentru cazul $p \ll n$ se pot scrie mai multe variante; utilizaţi principiul lui Brent pentru algoritmul de mai sus sau prin sortări locale urmate de $exch()$; comparaţi complexităţile.

P5.4.3 Desenele nu sunt identice; dacă ne gândim însă că o permutare de linii nu modifică proprietatea unei reţele de a sorta, reţelele sunt identice (a doua provine din prima prin inversarea liniilor—prin oglindire faţa de orizontală, dacă preferaţi).

P5.4.4 Acum este vorba despre oglindire faţă de verticală şi aici nu e evident. Cum reţeaua are minimul de comparatoare, rezultă că pentru $Z = \langle n, \ldots, 1 \rangle$ se fac interschimbări la fiecare comparaţie, rezultatul fiind $Z' = \langle 1, \ldots, n \rangle$. Dacă se parcurge reţeaua în sens

invers, pornind de la Z' şi schimbând sensul comparaţiilor se obţine Z, deci reţeaua inversată sortează descrescător, iar dacă se reface sensul comparaţiilor, sortează crescător.

P5.4.5 Modificările sunt minore. Este vorba numai despre difuzări: ale parametrilor s, d, q, q', a pivotului, imediat după alegerea lui, a poziţiei pivotului i_v, calculată de procedura *partiţionare*. Deci $O(\log q)$ operaţii (neimplicând comparaţii), la fiecare iteraţie, ceea ce este nesemnificativ. În *partiţionare*, suma prefixelor este deja pentru EREW PRAM.

P5.4.6 La iteraţia l ($l = d-1 : -1 : 0$), se alege $t = p/2^{d-l}$, ceea ce asigură că procesorul respectiv rămâne cu cel puţin $p/2^{d-l-1}$ elemente, din care va alege pivotul la următoarea iteraţie; în final poate rămâne fără nici un element, dar scopul – găsirea unui pivot – a fost atins. De menţionat că celelalte cel puţin $p/2^{d-l-1}$ elemente sunt transmise unui procesor care va alege şi el pivoţi, începând din iteraţia următoare; nici acesta nu va rămâne fără elemente, până după ultima iteraţie. De obicei t se ţine constant; aici am vrut doar să arătăm o limită inferioară; deci $t \geq p/2$.

P5.4.7 Nu. Cheile egale vor fi în final trimise aceluiaşi procesor. Evident că am făcut ipoteza că algoritmul 5.9 lucrează corect (vezi problema 5.2.5).

P5.4.8 Presupunem secvenţele $X_0, \ldots, X_{p-1}$ concatenate într-o zonă contiguă de memorie X; notăm $i_0, \ldots, i_{p-1}$ indicii de la care încep secvenţele ($i_0 = 0$, $i_1 = n_0$, etc.).

1. **pentru** $j = 1 : \log p$
    1. **pentru** $l = 0 : p/2^j - 1$
        1. $s \leftarrow 2^j l$, $d \leftarrow 2^{j-1}(2l+1)$

interclasează secvenţele începând la i_s şi i_d, de lungime n_s, respectiv n_d
$i_s \leftarrow i_s + i_d$, $n_s \leftarrow n_s + n_d$

La fiecare iteraţie în j se pot face interclasările într-o zonă Y, de lungime n, care se copiază apoi în X (în fiecare interclasare în parte scriindu-se în Y începând de la valoarea i_s curentă). Oricum, e nevoie de n locaţii de memorie suplimentare.

P5.4.10 Fiecare procesor alege elementele din poziţiile $\lfloor im/p \rfloor$, $i = 1, 2$; deci P_0: 16 şi 27, P_1: 13 şi 23, P_2: 10 şi 22. Se ordonează aceste elemente: $Y = \langle 10, 13, 16, 22, 23, 27 \rangle$; din Y se aleg elementele din poziţiile $i(p-1)$, $i = 1, 2$, adică 16 şi 23; acestea sunt cheile de partiţionare. În final P_0 va avea 16 elemente, P_1 7 şi P_2 13.

P5.4.11 Algoritmul pentru procesorul P_k este ($p = 2^d$):

1. formează secvenţa locală X
2. **pentru** $i = 0 : d-1$
    1. **dacă** $k_i = 1$ **atunci**
        1. **send**(X, i), **stop**
    2. **altfel recv**(X', i), $X \leftarrow X \perp X'$
3. **difuzare**: P_0 difuzează cheile de partiţionare

După pasul i, rămân active numai procesoarele cu număr de ordine având ultimii $i+1$ biţi egali cu 0. La fiecare pas mesajele transmise îşi dublează dimensiunea (presupunând că erau iniţial egale). Numărul de operaţii aritmetice este $\sum_{i=0}^{d-1} 2^{d+1}(p-1) \approx 2p^2$; comunicaţia se face în timp $O(p^2)$, plus $O(p \log p)$ pentru difuzare. Faţă de algoritmul 5.27, timpul de sortare este redus de $d/2$ ori, pentru că acum toate procesoarele participă la această operaţiune.

P5.4.13 Se foloseşte un algoritm de căutare binară; secvenţă unimodală este separată în două; se compară ultimul element din prima parte cu primul din a doua parte; se continuă căutarea numai în subsecvenţa care conţine cel mai mic dintre aceste două elemente (o subsecvenţă a unei secvenţe unimodale este şi ea unimodală). De exemplu:

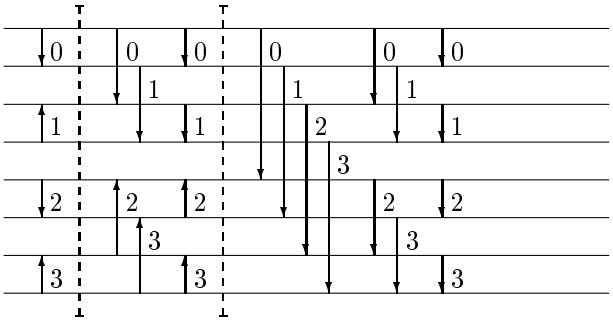


Figura 7.4: O planificare a procesoarelor pentru sortarea bitonică pe un PRAM.

1. $i \leftarrow 0$, $f \leftarrow m-1$
2. **cât timp** $i \neq f$
    1. $j = \lfloor (i+f)/2 \rfloor$
    2. **dacă** $a_j < a_{j+1}$ **atunci** $f \leftarrow j$
    3. **altfel** $i \leftarrow j+1$

Minimul se va găsi în a_j. În cazul unei secvenţe bitonice, acest algoritm nu se poate aplica. Se foloseşte algoritmul secvenţial clasic de căutare a minimului, eventual cu o condiţie de terminare mai bună (care să profite de faptul că există un singur minim local). Oricum, în cel mai rău caz e nevoie de $n-1$ comparaţii.

P5.4.14 Detaliem doar partea referitoare la interclasare; rezultatul va fi secvenţa sortată $\tilde{A}$; număr de comparaţii: $m-1$.

1. găseşte j, indicele elementului minim
2. $\tilde{a}_0 \leftarrow a_j$, $l \leftarrow 1$, $i \leftarrow (j-1) \bmod m$, $f \leftarrow (j+1) \bmod m$
3. **cât timp** $i \neq f$
    1. $\tilde{a}_l \leftarrow \min(a_i, a_f)$, $l \leftarrow l+1$, $\tilde{f} \leftarrow \mathrm{indmin}(a_i, a_f)$
    2. **dacă** $\tilde{f} = i$ **atunci** $i \leftarrow (i-1) \bmod m$
    3. **altfel** $f \leftarrow (f+1) \bmod m$

P5.4.15 Să mai desenăm o dată figura 5.9, amintindu-ne că fiecare săgeată reprezintă o comparaţie şi deci este efectuată de un procesor (modelul este EREW PRAM); dacă asociem fiecărei săgeţi numărul procesorului care efectuează comparaţia respectivă, rezultă figura 7.4 (evident că aceasta nu este singura planificare posibilă).

Ne vom inspira din algoritmul 5.28 în ce priveşte organizarea buclelor. Vă rămâne acum doar sarcina de a verifica buna funcţionare a algoritmului ce urmează; k este numărul de ordine al procesorului, între 0 şi $n/2$.

1. **pentru** $l = 0 : d-1$
    1. **pentru** $i = l : -1 : 0$
        1. $j_1 \leftarrow k \bmod 2^i + 2^{i+1} \lfloor k/2^i \rfloor$
        2. $j_2 \leftarrow j_1 + 2^i$

3. **dacă** $k_l = 0$ **atunci** $exch(j_1, j_2)$ {ordonează crescător}
4. **altfel** $exch(j_2, j_1)$

P5.4.16 Se împarte secvenţa A în $2p$ secvenţe de $m = n/(2p)$ elemente; fiecare procesor sortează două astfel de secvenţe. Apoi cele p procesoare aplică sortarea bitonică celor $2p$ secvenţe, cu amendamentul descris pe larg pentru cazul memorie distribuită (ideea din algoritmul 5.30). Diferenţa majoră faţă de acest caz este că un singur procesor calculează $\min(A_i, A_j)$ şi $\max(A_i, A_j)$, cu m comparaţii (în cazul distribuit două procesoare făceau fiecare câte m comparaţii). Numărul total de comparaţii este

$$2\frac{n}{2p}\log\frac{n}{2p} + \sum_{l=0}^{d-1}\left(\frac{n}{p} + \sum_{i=0}^{l}\frac{n}{2p}\right) \approx \frac{n}{p}\log n + \frac{n}{p}(\log p - 1) + \frac{n}{4p}\log^2 p = O((n/p)\log n).$$

Comparând cu relaţia (5.7) se constată că aici coeficientul termenului în $(n/p)\log p$ (al doilea ca importanţă) este de două ori mai mic.

Dacă aplicăm ideea algoritmului 5.31, de a obţine subsecvenţele minim şi maxim prin interclasare, atunci numărul de comparaţii va fi

$$2\frac{n}{2p}\log\frac{n}{2p} + \sum_{l=0}^{d-1}\sum_{i=0}^{l}2\frac{n}{2p} \approx \frac{n}{p}\log n - \frac{n}{p} + \frac{n}{2p}\log^2 p,$$

deci mai mare decât în cazul anterior; explicaţia e simplă: interclasarea necesită de două ori mai multe operaţii decât simpla extragere a minimului şi maximului prin comparaţii între câte două elemente. De altfel, comparând cu (5.8), se constată aproape aceeaşi expresie, deci pe PRAM nu se reduce numărul de operaţii.

P5.4.17 Algoritmul este corect dacă demonstrăm că metoda de interclasare este corectă; o vom face în cazul general $|A| = n$, $|B| = m$. După sortare, secvenţa A va conţine k zerouri, urmate de $n - k$ valori de unu; B va conţine l zerouri urmate de $m - l$ de unu. Atunci $X = \langle x_0, x_1, \ldots\rangle$ va începe cu $\lceil k/2\rceil + \lceil l/2\rceil$ zerouri, iar $Y = \langle y_0, y_1, \ldots\rangle$ va începe cu $\lfloor k/2\rfloor + \lfloor l/2\rfloor$ zerouri. Diferenţa $(\lceil k/2\rceil + \lceil l/2\rceil) - (\lfloor k/2\rfloor + \lfloor l/2\rfloor)$ poate avea valorile 0, 1 sau 2 ($\lceil k/2\rceil - \lfloor k/2\rfloor$ poate fi 0 sau 1). Dacă este 0 sau 1, atunci secvenţa Z este deja sortată. Dacă este 2, atunci una din operaţiile $exch(y_i, x_{i+1})$ va ordona Z. Cele trei cazuri sunt ilustrate de următorul exemplu:

$A = \langle 0, 1, 1, 1, \ldots\rangle$	$A = \langle 0, 0, 1, 1, \ldots\rangle$	$A = \langle 0, 0, 0, 1, \ldots\rangle$
$B = \langle 0, 1, 1, 1, \ldots\rangle$	$B = \langle 0, 1, 1, 1, \ldots\rangle$	$B = \langle 0, 1, 1, 1, \ldots\rangle$

P5.4.18 Reţeaua de sortare este cea din figura 7.5 (secretul e de a desena dinspre dreapta spre stânga). Timpul de calcul este acelaşi cu cel al sortării bitonice, adică $\frac{1}{2}\log^2 n$; se folosesc mai puţine procesoare decât la sortarea bitonică, dar gradul de paralelism este tot $n/2$; s-a demonstrat că numărul total de comparaţii folosite este asimptotic tot $O(n\log^2 n)$; este una dintre cele mai economicoase reţele de sortare. La implementarea pe o arhitectură MIMD cu memorie distribuită sunt importante gradul de paralelism (care e bun) şi topologia de comunicaţie; dacă, la sortarea bitonică, pe un hipercub comunicau doar vecinii, acum, chiar dacă unele procesoare au timpi morţi, există comunicaţie între procesoare aflate la orice distanţă (şi tocmai acestea sunt procesoarele care nu au timpi morţi).

P5.4.19 Se presupune că A este o secvenţă conţinând, în ordine, k de unu, l de zero şi $n - k - l$ de unu (forma generală a unei secvenţe unimodale cu minim formată din unu şi

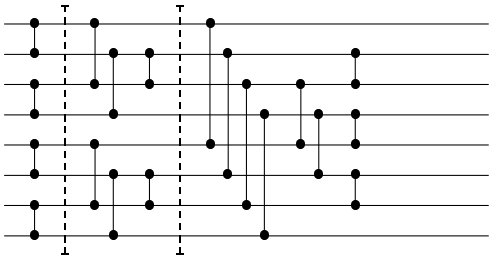


Figura 7.5: Alg. Batcher de sortare cu interclasare par-impară: reţea de sortare.

zero). Se pot întâlni patru cazuri, după parităţile lui k şi l; secvenţele "pară" şi "impară" rezultate sunt exemplificate mai jos:

$\langle 1,1,0,1,1,\ldots\rangle$	$\langle 1,1,0,0,1,\ldots\rangle$	$\langle 1,1,0,0,1,\ldots\rangle$	$\langle 1,1,0,0,1,\ldots\rangle$
$\langle 1,0,0,1,1,\ldots\rangle$	$\langle 1,0,0,1,1,\ldots\rangle$	$\langle 1,1,0,1,1,\ldots\rangle$	$\langle 1,1,0,0,1,\ldots\rangle$

După sortare, numărul de zerouri iniţiale din secvenţele "pară" şi "impară" diferă prin cel mult 1, deci una dintre operaţiile $exch()$ va ordona secvenţa A.

Desenul reţelei de sortare se face, ca şi în cazul sortării bitonice, înapoi (spre stânga) pe etape şi înainte în cadrul unei etape (etapele sunt separate de linii punctate, în figura 7.4).

P5.4.20 Presupunând $n = 2^m$, suma va fi egală cu $\log m!$. Dar $m^{m/2} < m! < m^m$, şi deci $\log m! = O(m \log m)$. Cum $m = \log n$, se obţine rezultatul cerut.

Bibliografie

[1] B. Abali, F. Özgüner, and A. Bataineh. Balanced Parallel Sort on Hypercube Multiprocessors. *IEEE Trans. Par. Distrib. Syst.*, 4(5):572–581, 1993.

[2] K.E. Batcher. Sorting networks and their application. In *Proc. AFIPS 1968 SICC*, pages 307–314, 1968.

[3] D.P. Bertsekas, C. Özveren, G.D. Stamoulis, P. Tseng, and J.N. Tsitsiklis. Optimal Communication Algorithms for Hypercubes. *J.Par.Distrib.Comput.*, 11:263–275, 1991.

[4] D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and distributed computation.* Prentice-Hall, 1989.

[5] P. Bjørstad, F. Manne, T. Sørevik, and M. Vajteršic. Efficient Matrix Multiplication on SIMD Computers. *SIAM J. Matrix Anal. Appl.*, 13(1):386–401, 1992.

[6] T. Brown and R. Xiong. A Parallel quicksort Algorithm. *J. Par. Distrib. Comput.*, 19:83–89, 1993.

[7] R.M. Chamberlain. An Algorithm for LU Factorization with partial pivoting on a hypercube. Technical Report CCS 86/11, Dept. of Science and Technology, Chr. Michelson Institute, Bergen, Norway, June 1986.

[8] E.G. Coffman and P.J. Denning. *Operating Systems Theory.* Prentice-Hall, 1973.

[9] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *19 th. Annual ACM Symp. Theory Comp.*, pages 1–6, 1987.

[10] J.J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A Set of Level-3 Basic Linear Algebra Subprograms. *ACM Trans.Math.Software*, 16:1–17,18–28, 1990.

[11] B. Dumitrescu, J-L. Roch, and D. Trystram. Fast matrix multiplication algorithms on mimd architectures. *Parallel Algorithms and Applications*, 4(2):53–70, 1994.

[12] G. Fox et al. *Solving problems on concurrent processors.* Prentice-Hall, 1988.

[13] G.H. Golub and C.F. Van Loan. *Matrix Computations. Second edition.* The John Hopkins University Press, 1989.

[14] M.T. Heath and C.H. Romine. A Parallel Triangular Solver for a Distributed-memory Multiprocessor. *SIAM J. Sci. Stat. Comput.*, 9(3):558–587, May 1988.

[15] R.W. Hockney and C.R. Jesshope. *Parallel Computers 2.* Adam Hilger Ltd., 1988.

[16] R.M. Karp and V. Ramachandran. Parallel Algorithms for Shared-Memory Machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science.* Elsevier Science Publishers, 1990.

[17] D.E. Knuth. *The Art of Computer Programming. Vol.3, Sorting and Searching.* Addison-Wesley, 1973.

[18] C.P. Kruskal. Searching, merging, and sorting in parallel computation. *IEEE Trans.Comput.*, 32(10):942–946, 1983.

[19] S. Lennart Johnsson and C.T. Ho. Optimum Broadcasting and Personalized Communication in Hypercubes. *IEEE Trans.Comput.*, 38(9):1249–1268, 1989.

[20] G. Li and T.F. Coleman. A Parallel Triangular Solver for a Distributed-memory Multiprocessor. *SIAM J. Sci. Stat. Comput.*, 9(3):485–502, May 1988.

[21] G. Li and T.F. Coleman. A New Method for Solving Triangular Systems on Distributed-memory Message-passing Multiprocessor. *SIAM J. Sci. Stat. Comput.*, 10(2):382–396, March 1989.

[22] B. Monien and H. Sudborough. Embedding one Interconnection Network in Another. *Computing Suppl.*, 7:257–282, 1990.

[23] Y. Saad and M.H. Schultz. Data Communication in Parallel Architectures. *Parallel Computing*, 11:131–150, 1989.

[24] H. Shi and J. Schaeffer. Parallel Sorting by Regular Sampling. *J. Parallel Distrib. Comput.*, 14:361–372, 1992.

[25] Q.F. Stout and B. Wagar. Intensive Hypercube Communications - Prearranged Communication in Link-Bound Machines. *J.Par.Distrib.Comput.*, 10:167–181, 1990.

[26] B. Wagar. Hyperquicksort. In *Hypercube Multiprocessors 1987, Greece*, pages 292–299. SIAM, 1987.

[27] R. Xiong and T. Brown. Parallel Median Splitting and k-splitting with Application to Merging and Sorting. *IEEE Trans.Par.Distrib.Syst.*, 4(5):559–565, 1993.