**UNIVERSITATEA DIN BACĂU**
**FACULTATEA DE ŞTIINȚE**

**ELENA NECHITA** **CERASELA CRIŞAN**

**MIHAI TALMACIU**

# ALGORITMI PARALELI ŞI DISTRIBUIȚI

**Curs pentru studenții facultăților**

**INGINERIE, specializarea TEHNOLOGIA INFORMAȚIEI**

**şi**

**ŞTIINȚE, specializarea INFORMATICĂ**

**2008**

**It would appear that we have reached the limits**
**of what it is possible to achieve with computer technology,**
**although one should be careful with such statements,**
**as they tend to sound pretty silly in 5 years.**

(John von Neumann, 1949)

**CUPRINS**

# INTRODUCERE

## NECESITATEA ALGORITMILOR PARALELI ŞI A CALCULULUI DISTRIBUIT

In istoria calculatoarelor s-a impus de la început secvențializarea. Rezolvarea unei probleme concrete presupune construirea unui algoritm de calcul care, de regulă, formulează ordinea în care se vor executa diferitele operații.

Structura calculatorului, aşa cum a fost ea elaborată de John von Neumann, stabileşte că operațiile, fie logice, fie aritmetice, se execută în unitatea centrală, în blocul aritmetico-logic. Astfel, programul realizat pentru soluționarea problemei, ca succesiune de instrucțiuni de calculator care urmează algoritmul adecvat, este la rândul său memorat în calculator, iar instrucțiunile sale sunt aduse una câte una în unitatea de comandă, realizându-se pas cu pas transformarea datelor de intrare în rezultatele finale... O vreme părea că paralelismul este atuul de neatins al gândirii umane. Dar încă din anii '60-'70 creşterea vitezei de calcul s-a realizat mai ales prin trucuri, prin diviziunea sarcinilor în cadrul sistemului de calcul, prin introducerea cererilor de întrerupere din partea dispozitivelor de intrare/ieşire, prin accesul direct la memorie.

Apoi au început să apară "supercalculatoarele", dar mai ales sisteme specializate pentru prelucrarea imaginilor numerice, sisteme în care s-a căutat să se compenseze viteza insuficientă de pe atunci printr-o procesare paralelă, alocând pentru fiecare pixel dintr-o linie a imaginii câte o unitate de calcul - un procesor dedicat operațiilor locale din imagine. Astfel au apărut primele configurații de calcul paralel, dar şi primii algoritmi de calcul paralel. In cazul imaginilor numerice, asupra fiecărui element de imagine (pixel) se pot aplica simultan aceleaşi transformări, astfel încât se puteau folosi procesoare identice care nici măcar nu trebuiau să comunice între ele. Era modelul de calcul care s-a numit SIMD (Single Instruction Multiple Data).

In alte aplicații, cu calcule mai complicate, s-a căutat un spor de viteză prin înlănțuirea unor procesoare care trebuiau să execute operațiuni distincte, rezultatele unuia intrând ca date de intrare în cel de al doilea.

Se întrevede deja dificultatea majoră a acestui mod de lucru, interdependența dintre structura hard disponibilă şi algoritmul şi programul de calcul. Pare o revenire la sistemele analogice de calcul, acolo unde pentru fiecare tip de problemă trebuia realizat un anumit montaj, un calculator analogic particular.

Cu toate acestea, paralelismul a constituit unul din mecanismele majore de creştere a performanțelor sistemelor moderne de calcul. Intre altele, înzestrarea controlerelor de intrare/ieşire cu procesoare specializate, degrevarea microprocesorului (unității centrale) de sarcinile de vizualizare pe tubul catodic (la PC-uri), prin creşterea complexității interfeței video. Dar şi completarea microprocesorului cu o memorie tampon, odată cu creşterea performanțelor accesului direct la memorie, transferului din memoria externă în cea internă.

In anii '90, creşterea vitezei interne a microprocesoarelor a redus sensibil interesul pentru structurile de calcul paralel din deceniile precedente. Era mult mai importantă portabilitatea programelor, a aplicațiilor, astfel încât s-a lucrat intens pentru elaborarea unor sisteme de operare performante, căutând ca sistemul de operare să folosească cât mai eficient configurația hard.

In schimb, dezvoltarea rețelelor de calculatoare aduce în scenă noi versiuni de supercalculatore: serverul multiprocesor, pe de o parte, şi "clusterul" de PC-uri

comunicând între ele şi rezolvând feliuțele problemei, pe de alta. Vorbim tot mai mult de calcul distribuit şi, într-un fel, ajungem să folosim Internetul ca un imens super-calculator.

Exemple întâlnite în presă sau în buletine de ştiri, despre *Proiectul Genomului Uman*, despre proiectul *SETI at Home*, sau, mai nou despre *Eistein at Home*, despre studiile seismologice din Japonia, despre implicarea firmei IBM în cercetarea universului sunt tot atâtea ştiri despre sisteme de calcul paralel sau distribuit.

Se vorbeşte, în general despre calculul de înaltă performanță (HPC - *High Performance Computing*, dar concret despre sisteme distribuite - GRID, de fapt despre calcule în rețea, despre arhitectura deschisă a serviciilor Grid (*Open Grid Services Architecture*), despre clusteri şi super-clusteri.

Unul dintre cele mai ambițioase proiecte, sistemul *TerraGrid*, proiectat pentru a sprijini *National Science Foundation* în proiectele de cercetare de mare anvergură (precum modelarea moleculară, detecția bolilor, descoperirea medicamentelor, sau descoperirea de noi surse de energie) ar urma să folosească peste 3000 de procesoare Intel rulând sub Linux. Părinte al GRID este considerat Ian Foster, autorul manualului electronic DBPP "Proiectarea şi Construcția Prelucrărilor Paralele".

# Capitolul 1

# SISTEME DE CALCUL PARALEL

Conceptul clasic a lui von Neumann despre computerul serial a fost încorporat în primele maşini moderne de calcul. Viteza de calcul a crescut considerabil odată cu înlocuirea tuburilor cu tranzistori şi circuite integrate.

La un moment dat, însă, capabilităţile computerelor sunt inevitabil cu un pas în urma necesităţilor aplicaţiilor ştiinţifice şi tehnologice. In zilele noastre, un computer serial efectuează peste $10^9$ operaţii pe secundă. Din păcate, nu ne putem aştepta ca, în viitor, să fie construite maşini care să lucreze mult mai rapid decât cele existente astăzi. La baza acestei afirmaţii se află raţiuni fizice. Un semnal electric se propagă într-un metru aproximativ într-o nanosecundă ($10^{-9}$ sec) şi cum anumite componente ale calculatorului nu pot avea dimensiuni sub ordinul milimetrilor, limita fizică a numărului de operaţii se află undeva în jurul a $10^{10}$ operaţii într-o secundă.

Următorul pas în căutarea de metode de îmbunătăţire a vitezei de calcul este **paralelismul**. **Un algoritm paralel este un algoritm care permite efectuarea simultană a mai multor operaţii.**

In anii '70 a luat un avânt deosebit proiectarea unor calculatoare a căror circuite erau divizate în subunităţi, fiecare executând diferite operaţii. Paralelismul se realizează la nivelul asamblorului. Un exemplu este iniţierea unei operaţii înaintea terminării operaţiei precedente. Au fost construite maşini (Cray şi Ciber) care cuplează această tehnică, numită "pipelining" (tehnica conductei), cu unităţi hardware independente pentru execuţia unor anumite operaţii, cum ar fi adunarea şi multiplicarea. Termenul de procesor vectorial descrie în mod uzual un asemenea sistem. Procesarea fluxului de date într-o maşină de calcul vectorial se aseamănă cu o bandă de producţie dintr-o fabrică.

Apariţia circuitelor integrate a permis dezvoltarea supercalculatoarelor. Ideea de bază este eliminarea bufferelor de mare viteză şi conectarea directă a procesoarelor la bănci de memorie. Memoria este distribuită între procesoare şi o parte este accesibilă tuturor unităţilor. Unitatea centrală unică este înlocuită cu mai multe procesoare care, deşi individual pot lucra încet, accelerează viteza de procesare operând în paralel. Schemele de interconectare fizică a procesoarelor, utilizate în prezent, sunt de tip hipercub, inel sau latice.

Calculul paralel a dat o dimensiune nouă construcţiei de algoritmi şi programe.

Programarea paralelă nu este o simplă extensie a programării seriale. Experienţa a arătat că modul de judecare a eficienţei algoritmilor bazaţi pe tehnici seriale nu corespunde în cazul paralel. Nu toţi algoritmii secvenţiali pot fi paralelizaţi, aşa cum în natură există o serie de exemple (exemplul lui N. Wilding: trei femei nu pot produce un copil numai în trei luni, lucrând împreună la această problemă). Pe de altă parte, o serie de algoritmi numerici standard seriali dovedesc un grad înalt de paralelism: conţin numeroase calcule care sunt independente unele de altele şi pot fi executate simultan. In proiectarea unor algoritmi de calcul paralel este necesară regândirea sistemelor, limbajelor, problemelor nenumerice şi a metodelor numerice.

Viitorul calculatoarelor paralele depinde în mare măsură de efortul care se face în momentul de faţă pentru stabilirea algoritmilor paraleli cei mai eficienţi şi de proiectarea limbajelor paralele în care aceşti algoritmi pot fi exprimaţi.

In cadrul unui calculator paralel, nu este necesară încorporarea unor procesoare cu performanţe deosebite. Astfel, costul unui calculator paralel cu un număr mare de

procesoare poate fi relativ ieftin față de un calculator serial sau un supercomputer vectorial cu performanțe de procesare comparabile. Totuşi, la momentul actual, calculatoarele vectoriale prezintă procentul cel mai ridicat de achiziționări pe piața de supercalculatoare. Motivul este tehnologia software relativ primitivă existentă pentru calculatoarele paralele. Prin transferarea pe calculatorul paralel a codurilor seriale elaborate de-a lungul anilor nu se poate obține implicit eficiența maximă. Din păcate, tehnologia comunicațiilor este mult în urma tehnologiilor de calcul şi, pe calculatoarele existente pe piață, multe aplicații suferă de o anumită limită a comunicațiilor: raportul dintre timpul de comunicare şi timpul de calcul efectiv, în majoritatea aplicațiilor, este extrem de ridicat.

## CE SUNT CALCULATOARELE PARALELE? CE ESTE PROGRAMAREA PARALELĂ?

Un **calculator paralel** este o colecție de procesoare, de obicei de acelaşi tip, interconectate într-o anumită rețea care permite coordonarea activităților lor şi schimbul de date. Se presupune că procesoarele se află la distanțe mici unele de altele şi pot colabora la rezolvarea unei probleme.

Spre deosebire de un calculator paralel, un **sistem distribuit** este o mulțime de procesoare, de obicei de tip diferit, distribuite pe o arie geografică mare, construit în scopul utilizării resurselor disponibile şi colectarea şi transmiterea informațiilor printr-o rețea de conectare a procesoarelor. Programele paralele utilizează concurența pentru a rula mai rapid. Un sistem distribuit utilizează procese concurente datorită distribuirii fizice a maşinilor din care este compus (un exemplu este poşta electronică: presupune procese diferite pe diferite stații de lucru, scopul nefiind acela de a comunica mai rapid decât prin utilizarea unei singure stații).

Scopul procesării paralele este executarea unor calcule mai rapid decît ar fi posibil cu un singur procesor, prin utilizarea concurentă a mai multe procesoare. Este destinat aplicațiilor ce necesită soluții rapide sau rezolvarea unor probleme de dimensiuni mari (de exemplu, dinamica fluidelor, vremea probabilă, modelarea şi simularea sistemelor mari, procesarea şi extragerea informației, procesarea imaginilor, inteligență artificială, manufacturare automată).

Există trei motivații pentru utilizarea procesorului paralel:

1. pentru a atinge performanța cerută relativă la timpul de execuție;
2. pentru că este o arhitectură disponibilă;
3. pentru că problema care se pune se pretează la calculul paralel.

Cei trei factori principali care au favorizat introducerea pe scară largă a procesării paralele sunt:

• costul relativ scăzut al unui sistem cu mai multe procesoare;

• tehnologia circuitelor integrate a avansat în asemenea măsură întrucât pe un singur cip pot fi înglobate milioane de tranzitoare;

• ciclul de timp al procesorului serial se apropie de limitele fizice sub care nu este posibilă nici o îmbunătățire.

În dezvoltarea conceptului de paralelism s-au conturat două direcții de cercetare:

1. în problema hardului, respectiv care este arhitectura calculatorului care avantajează anumiți algoritmi de rezolvare a unor probleme diverse;

2. calcul paralel orientat pe problemă, respectiv cât de mult îmbunătățesc algoritmii paraleli viteza de calcul pentru o problemă dată.

**Programarea paralelă** este arta de a programa o colecție de calculatoare pentru a executa eficient o singură aplicație. Dacă aplicația este numerică sau de calcul simbolic, eficiența înseamnă atingerea unei viteze mari de execuție (invers proporțională cu numărul de procesoare). În cazul unei aplicații în timp real sau a unui sistem de operare, eficiența constă în satisfacerea în timp real a cerințelor impuse de unități sau utilizatori.

Programarea paralelă caută căile de divizare a aplicațiilor în unități (procese) care pot fi executate concurent pe mai multe procesoare. Presupune:

1. specificarea problemei;

2. identificarea unităților fundamentale şi interacțiunile dintre acestea;

3. transpunerea acestor unități fundamentele în procese cu interacțiunile specificate prin primitive de comunicare.

Programarea paralelă este parte componentă a **programării concurente**. Termenul de programare concurentă este asociat atât cu sistemele multiprocesor şi rețelele, cât şi cu sistemele de operare şi sistemele în timp real.

Un algoritm secvențial specifică o execuție secvențială a unui set de instrucțiuni. Execuția sa este numită proces. Un algoritm paralel specifică doi sau mai mulți algoritmi secvențiali care pot fi executați simultan ca procese paralele (concurente). Astfel, un proces este o colecție de instrucțiuni de control secvențiale care accesează date locale sau globale şi care poate fi executat în paralel cu alte unități de program.

Procesoarele pe care se execută un program paralel pot fi grupate într-o unitate (multiprocesor sau calculator paralel) sau pot fi separate ca maşini autonome conectate printr-o rețea.

Programarea paralelă necesită un limbaj de calcul şi un limbaj de coordonare.

Limbajul de coordonare este cheia ce permite utilizatorului unificarea într-un program a mai multor activități separate, fiecare specificate utilizând limbajul de calcul. Limbajul de calcul permite calcularea unor valori şi manipularea obiectelor-date locale. Limbajul de coordonare permite crearea activităților (procese) simultane şi comunicarea între acestea. De obicei, funcțiile standard ale limbajului de calcul şi ale limbajului de coordonare sunt unite într-un super-limbaj.

Un limbaj concurent este un limbaj de programare de nivel înalt pentru programarea concurentă, adică care oferă următoarele facilități:

- descrierea proceselor paralele;
- mijloace de comunicare Între procese;
- posibilități de sincronizare a proceselor.

Limbajele concurente sunt de obicei orientate funcție de o anumită arhitectură: sistem monoprocesor, multiprocesor sau sistem distribuit.

## ANALOGIE CU VIAȚA COTIDIANĂ

Modelul de dezvoltare a calculatoarelor se aseamănă cu modelul dezvoltării umane. Supraviețuirea speciei umane depinde de avansul a două fronturi. Primul este performanța individului, fizică şi intelectuală, iar al doilea concurența socială: rezultatele sunt mult mai eficiente dacă se lucrează în grup. Cu cât grupul de indivizi este mai mare) cu atât progresul este mai mare.

Un număr de oameni de ştiință şi cercetători ai inteligenței artificiale au argumentat că inteligența umană se bazează pe interacțiunea unui număr mare de unități simple de procesare. Această idee prezintă interes în modelarea ființei umane şi a

inteligenței ei prin **procesare distribuită paralel** (prescurtat PDP). Un studiu al mecanismului minții relevă faptul că creierul uman este organizat dintr-un număr mare de elemente interconectate care își transmit reciproc semnale sub formă de excitatori și inhibitori. Un sistem PDP "învață" din exemple. Spre deosebire de inteligența artificială, în PDP nu există reguli de bază. Elementele unui PDP sunt:

1. o mulțime de unități de procesare;
2. starea activității sistemului la fiecare moment;
3. o anumită conectare între unități;
4. o regulă de propagare a unei directive de activitate;
5. o regulă de activare;
6. o regulă de învățare prin care structura de conectare este modificată prin experiență.

Pentru o cât mai bună înțelegere a modului de construcție a algoritmilor paraleli, se consideră următoarea problemă din viața cotidiană (Williams, 1990).

**Problema Familială**. O familie, compusă din Tata, Mama şi copii Ioan, Toma şi Simona, a terminat prânzul. Rămâne de curățat masa, spălat vasele, șters şi pus la loc în dulap.

**Soluții**. Există un număr de soluții care depinde de numărul de persoane (sau procesoare!) care sunt disponibile. Se remarcă următoarele.

• Modul secvențial cu utilizarea unei persoane (procesor serial). Mama curăță masa, spală vasele şi le pune în dulap, în timp ce Tata duce copii în parc.

• Modul secvențial uti1izând patru persoane (procesoare pipe-line):

1. Tata curăță masa când s-a terminat prânzul;
2. Ioan spală vasele, când tata a terminat de curățat;
3. Toma şterge vasele, când Ioan a terminat;
4. Simona le pune în dulap, când Toma a terminat;
5. Mama pleacă la cumpărături!

• Modul banda rulanta (pipe-line) utilizând patru persoane (procesor vectorial):

1. Tata ia un obiect de pe masă, i-l dă lui Ioan şi pleacă să aducă altul;
2. Ioan spală obiectul primit, i-l dă lui Toma şi aşteaptă să primească alt obiect de la Tata;
3. Toma şterge obiectul primit, i-l dă lui Simona şi aşteaptă să primească un alt obiect de la Ioan;
4. Simona şterge obiectul primit şi aşteaptă să primească altul de la Toma.
5. Mama îi priveşte cu admirație!

Din păcate, pot să apară probleme dacă Ioan spală încet vasele. Datorită sincronizării, ceilalți trebuie să aştepte după el!

• Modul de utilizare a mai multor persoane (procesor matriceal). Se presupune că familia este suficient de numeroasă ca fiecare să fie responsabil de un singur obiect de pe masă. Şeful familiei direcționează orice mişcare determinând fiecare persoană să facă simultan acelaşi lucru. Dacă şeful spune "curățați", fiecare va lua un obiect de pe masă. Când şeful spune "spălați", fiecare va spăla obiectul său ş.a.m.d. Şeful poate decide ca anumiți membrii ai familiei să nu lucreze. De exemplu, poate cere să fie spălate numai 5 pahare, astfel încât o serie de membrii rămân momentan fără activitate.

• Modul selectiv de transmitere a mesajelor (procesare paralela cu transmitere de mesaje). Obiectele care se curăță sunt "mesaje". Spre deosebire de modelul anterior, în care mesajele sunt transmise unidirecțional, mesajele pot fi transmise în mai multe direcții:

1. Dacă un obiect este curat, Tata îl transmite direct lui Simona;
2. Dacă Toma primeşte un obiect murdar, îl returnează lui Ioan.

Transmiterea obiectelor se poate face printr-un spațiu rezervat comunicării (masă, zonă tampon, buffer) care admite o anumită încărcare .

• Modul masa comuna (procesare paralela cu memorie comună). Obiectele murdare, ude sau curate stau pe aceeaşi masă. Fiecare persoană dispune de un spațiu mic de depozitare. Astfel, anumite obiecte, ca cele din apa de spălare, nu sunt accesibile tuturor persoanelor, numai cele de pe masă. Este necesară o atenție sporită pentru a nu re-spăla sau re-şterge lucrurile curate, respectiv uscate.

**Soluția grupului de procesoare.** Modele similare se pot construi înlocuind oamenii cu procesoare. Modelele depind de numărul şi tipul elementelor de procesare.

Există cinci modele hardware:
1. secvențial sau serial (model convențional),
2. procesor pipe-line şi procesor vectorial (procesoare care transmit datele într-o direcție bine definită),
3. procesor matriceal (procesoare "proaste" care ascultă de un controlor),
4. transmitere de mesaje (mai multe procesoare care lucrează împreună şi transmit date conform unui protocol stabilit);
5. memorie comună (mai multe procesoare care lucrează asupra aceluiaşi lot de date şi asupra unor date locale).

Calculatorul serial este constituit din două părți:
1. unitatea centrală de procesare (CPU) de dimensiune mică;
2. memoria care are o dimensiune mare.

CPU lucrează tot timpul, pe când memoria aşteaptă: din când în când CPU pune sau extrage informații în sau din memorie. De aceea, la un moment dat, doar o parte mică din hardware participă la calcul.

Un exemplu similar, din viața cotidiană, a fost propus de T. Jebeleanu: o companie în care şeful lucrează zi şi noapte, pe când cei 1000 de angajați nu lucrează numai când şeful vine în biroul lor. Modalitățile de creştere a eficienței activității din companie sunt:

1. angajarea mai multor persoane (mai multă memorie) - activitatea este îmbunătățită, dar eficiența descreşte;

2. angajarea unui nou şef mai competent (un CPU mai rapid) - efectul este similar cu cel din cazul anterior;

3. şeful este suplinit de mai mulți directori (procesoare) cu sarcini precise: unii aduc informațiile de la angajați, alții le prelucrează, alții decid unde să fie trimise rezultatele, alții transmit rezultatele - aceşti directori lucrează în paralel!

4. se angajează şefi de echipă (memorii cache care introduc un anumit paralelism în utilizarea hardware-ului, de partea memoriei);

5. se unesc mai multe companii (paralelism în sisteme cu granulație mare) - intervin probleme de comunicare între şefi, iar organizarea întregului ansamblu este mai dificilă decât problema inițială a organizării companiei mici.

## NIVELURI DE PARALELISM

Paralelismul este utilizat în scopul reducerii timpului de calcul. Nivelurile de aplicabilitate ale ideii de paralelism corespund anumitor perioade de timp:

1. paralelism la nivel de job:
   (a) Între joburi;
   (b) Între faze ale joburilor;
2. paralelism la nivel de program:
   (a) Între părți ale programului;
   (b) În anumite cicluri;
3. paralelism la nivel de instrucțiune: Între diferite faze de execuție ale unei instrucțiuni;
4. paralelism la nivel aritmetic şi la nivel de bit:
   (a) Între elemente ale unei operații vectoriale;
   (b) Între circuitele logicii aritmetice.

La nivelul cel mai înalt, se urmăreşte maximizarea vitezei de execuție a joburilor. Execuția unui job poate fi împărțită în mai multe faze secvențiale, fiecare având nevoie de anumite programe şi anumite resurse hard ale sistemului (faze tipice: compilare, editarea legăturilor, execuția, tipărirea rezultatelor). Deoarece operațiile de intrare/ieşire sunt mai lente decât unitatea centrală, se introduc mai multe canale de intrare/ieşire sau procesoare de periferice, care pot opera în paralel cu execuția programului.

În interiorul unui program pot exista părți de cod care sunt independente una față de alta şi pot fi executate în paralel de mai multe procesoare. Un alt exemplu sunt instrucțiunile repetitive a cărui execuție secvențială la anumit ciclu nu depinde de datele din ciclul precedent, astfel încât se pot executa în paralel atâtea cicluri câte procesoare sunt disponibile.

Execuția anumitor instrucțiuni poate fi împărțită în mai multe sub-operații şi poate fi aplicat principiul trecerii datelor printr-o "conductă" constituită din mai multe procesoare (procesoare pipeline).

La nivelul cel mai de jos se poate interveni în logica aritmetică realizând operații pe toți biții unui număr în paralel.

La construirea unui sistem paralel apar o serie de probleme:

1. Câte procesoare să fie utilizate?
2. Cât de mare să fie viteza fiecărui procesor?
3. Cum trebuie procesoarele interconectate?
4. Cât de mare să fie memoria?
5. Să se utilizeze o memorie comună sau memorie locală fiecărui procesor?

Eficiența unui program paralel depinde de calculatorul pe care se implementează şi astfel de răspunsurile la aceste întrebări.

## CLASIFICAREA SISTEMELOR PARALELE

In ultimii douăzeci de ani, au fost studiate şi construite sute de arhitecturi diferite, de la maşini cu procesoare foarte rapide total interconecate, până la maşini cu mii de procesoare lente.

Diferențele dintre arhitecturile paralele existente la momentul actual pot fi cuantificate prin parametrii următori:

1. numărul de procesoare şi puterea procesorului individual;

2. complexitatea rețelei de conectare şi flexibilitatea sistemului (sistemul poate fi utilizat la rezolvarea unei clase mari de probleme?);

3. distribuția controlului sistemului, adică dacă masivul de procesoare este condus de un procesor sau dacă fiecare procesor are propriul său controler;

4. mecanismul de control a sistemului;

5. organizarea memoriei.

O primă clasificare a sistemelor paralele se face în funcție de numărul de procesoare. Termenul consacrat este cel de granulație. Sistemele cu un număr mare de elemente de procesare, fiecare tratând un volum mic de date, au o "granulație fină" (de obicei de ordinul a 1000 de procesoare). Sistemele cu număr mic de procesoare care fiecare tratează un volum mare de date, sunt granulate grosier (de obicei, aproximativ 16 procesoare). Granulația medie se consideră ca fiind cea de 64 de procesoare.

Dintre sistemele cu granulație grosieră se remarcă sistemele pipeline, sistemele cu structuri de interconectare a procesoarelor tip arbore sau de tip inel, sistemele cu interconectarea vecinilor apropiați, sistemele cu interconectare prin magistrală (bus).

Dintre sistemele cu granulație fină se remarcă sistemele tip hipercub, sistemele organizate pe biți, circuitele reconfigurabile prin comutatoare sau sistemele sistolice.

Funcție de tipul procesoarelor din sistemul paralel, sistemele paralele sunt:

1. omogene, dacă conțin procesoare de acelaşi tip,
2. heterogene, dacă conțin procesoare de tipuri diferite.

Din punct de vedere al mecanismului de control se disting mai multe categorii:

1. grup de procesoare per instrucțiune unică: la un timp dat numai un set mic de instrucțiuni este într-o anumită fază de execuție. Două categorii se deosebesc:

• procesoare matriceale care efectuează calcul paralel sincronizat: instrucțiuni individuale operează asupra unui număr mare de date (asemenea procesoare necesită o memorie modulară),

• calculatoare care efectuează mai multe instrucțiuni concomitent,

2. grup de procesoare per instrucțiuni multiple.

Din punct de vedere al organizării memoriei se disting două categorii de sisteme:

1. multiprocesoare cu memorie comună;
2. multiprocesoare cu memorie distribuită.

După modul de comunicare a datelor între procesoare din cadrul unei rețele se disting:

1. sistem multiprocesor strâns cuplat, în care procesoarele cooperează intens pentru soluționarea unei probleme;
2. sistem multiprocesor slab cuplat, în care un număr de procesoare independente şi nu necesar identice comunică între ele printr-o rețea de comunicare.

Clasificarea cea mai des utilizată este următoarea (**clasificarea Flynn**). Conform acestei clasificări calculatoarele paralele aparțin unuia din următoarele patru sisteme:

• SISD: sistem cu un singur set de instrucțiuni şi un singur set de date;

• SIMD: sistem cu un singur set de instrucțiuni şi mai multe seturi de date;

• MISD: sistem cu mai multe seturi de instrucțiuni şi un singur set de date;

• MIMD: cu mai multe seturi de instrucțiuni şi mai multe seturi de date.

Sistemele paralele actuale fac parte din categoria SIMD sau MIMD. In primul caz procesarea paralelă are loc în paşi sincronizați (sistemul este condus printr-un unic controlor - respectă acelaşi tact dat de un ceas comun), iar în cazul al doilea, în paşi independenți (fiecare procesor are propriul controlorul - ceas propriu).

Procesarea concurentă îmbracă două forme:

1. pipelining
2. paralelism.

Un procesor pipeline conține un număr de procesoare aranjate astfel încât datele de ieşire ale unuia să constituie datele de intrare ale altuia. Procesoarele unui calculator paralel sunt aranjate într-o structură oarecare care permite operarea simultană a mai multor secvențe a unui set de date. Diferența fundamentală constă în faptul că operațiile pe un procesor pipeline se realizează prin executarea în secvență a mai multor sarcini distincte, asigurându-se astfel doar un grad limitat de paralelism (exploatează operațiile complexe).

**SISTEME SISD**

Un sistem SISD reprezintă maşina serială clasică, care execută o singură instrucțiune la un moment dat.

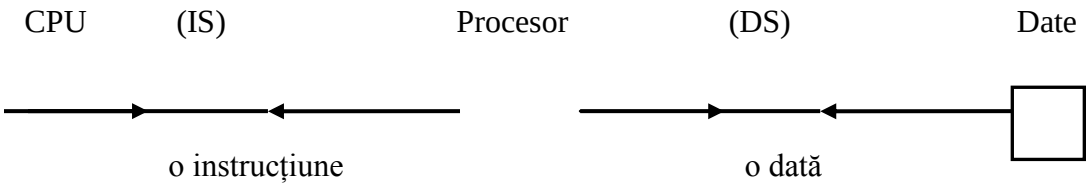
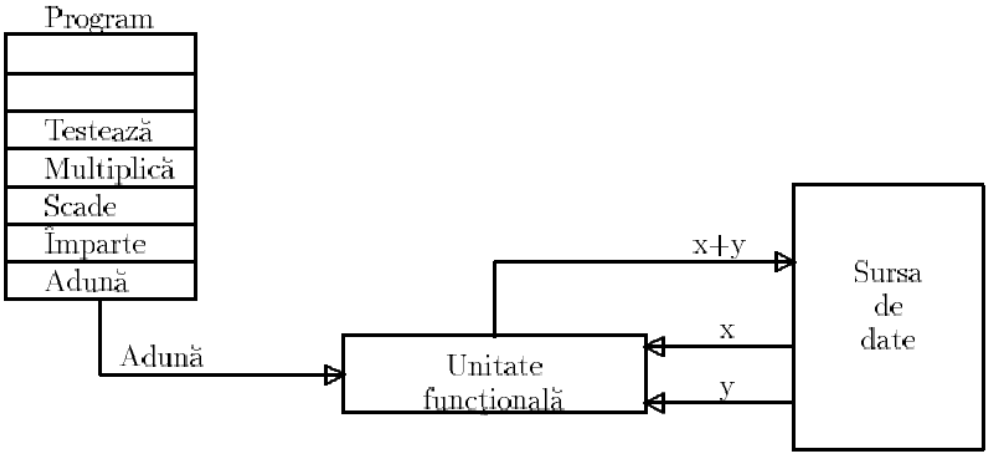



Figura de mai sus prezintă schema iar figura următoare prezintă funcționarea unui SISD:




**SISTEME SIMD**

Un sistem SIMD este compus dintr-o unitate de control (MCU - master control unit) şi un număr de procesoare identice.

Unitatea de control transmite aceeaşi instrucțiune la fiecare procesor în parte.

Instrucțiunile sunt executate în acelaşi timp, adică sunt sincrone. Fiecare procesor are o memorie privată şi anumite sisteme permit accesul la o memorie globală. Fiecare procesor operează asupra unui cuvânt (32 sau 64 biți) sau asupra unui operand format dintr-un singur bit, în fiecare ciclu de memorie.

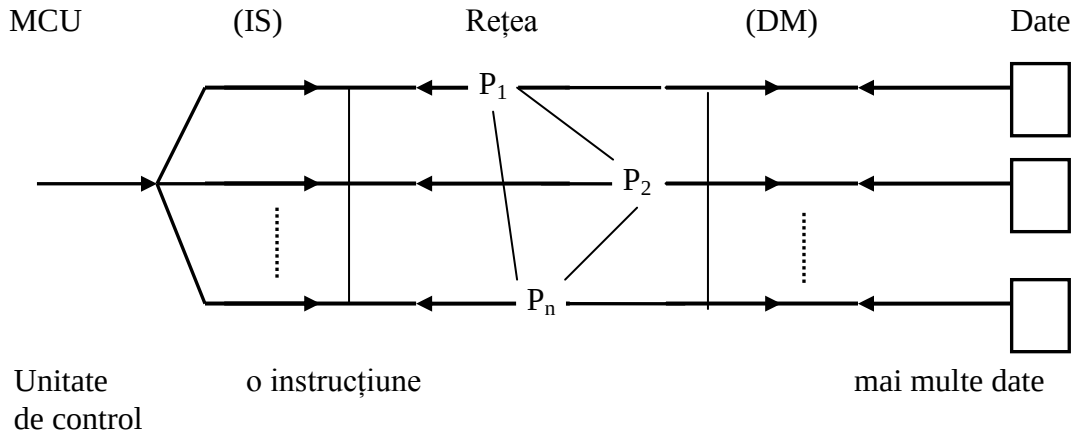
Sistemele SIMD sunt indicate pentru rezolvarea problemelor care pot fi descompuse în subprobleme ce presupun un efort de calcul similar (descompunere regulată). Asemenea probleme sunt:

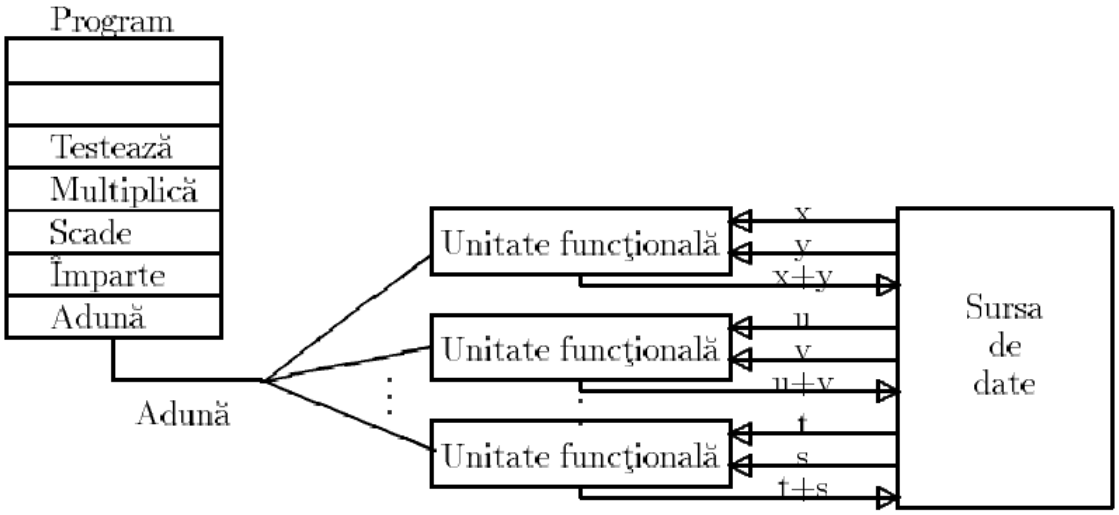
• procesarea de imagini: grupuri contigue de pixeli sunt asignate unui procesor, grupuri învecinate fiind asociate cu procesoare învecinate;

• dinamica fluidelor: traiectoriile moleculelor unui gaz sunt urmărite relativ la o grilă, fiecare procesor fiind responsabil pentru o secțiune din grilă;

• automate celulare, care sunt sisteme elementare a căror comportare colectivă simulează fenomene complexe din natură. Un automat celular formează în spațiu o grilă sau o latice. Fiecărei celule i se asociază o variabilă de stare care se poate schimba după anumite intervale, conform unor reguli.

Figurile următoare prezintă schema, respectiv funcționarea unui SIMD:




Se disting două tipuri de sisteme SIMD:

1. organizate pe cuvânt,
2. organizate pe bit.

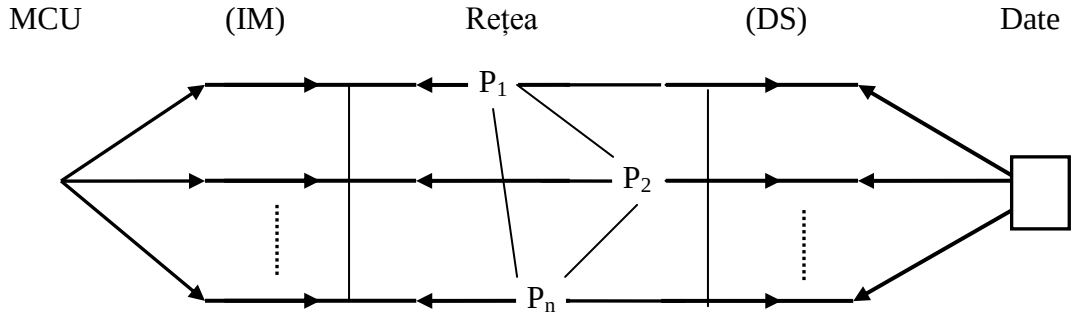
Operațiile aritmetice într-un sistem organizat pe cuvânt sunt similare cu cele efectuate de maşinile seriale.

Algoritmii pentru maşinile organizate pe bit depind de lungimea în biți a datelor şi nu de cardinalul mulțimii de date. Mai multe instrucțiuni sunt necesare pentru a efectua o operație aritmetică simplă. Timpul aritmetic este mai lung decât cel dintr-un sistem organizat pe cuvânt. Creşterea vitezei nu se obține prin creşterea vitezei fiecărui procesor în parte, ci prin utilizarea a cât mai multe procesoare simultan. In implementarea funcțiilor numerice, ca de exemplu, rădăcina pătrată, se poate profita de avantajele operației pe biți şi produce algoritmi aproximativi egali în efort cu o operație de multiplicare.

Exemplele clasice de sisteme SIMD sunt
1. procesoarele vectoriale,
2. procesoarele matriceale şi anumite matrici sistolice.

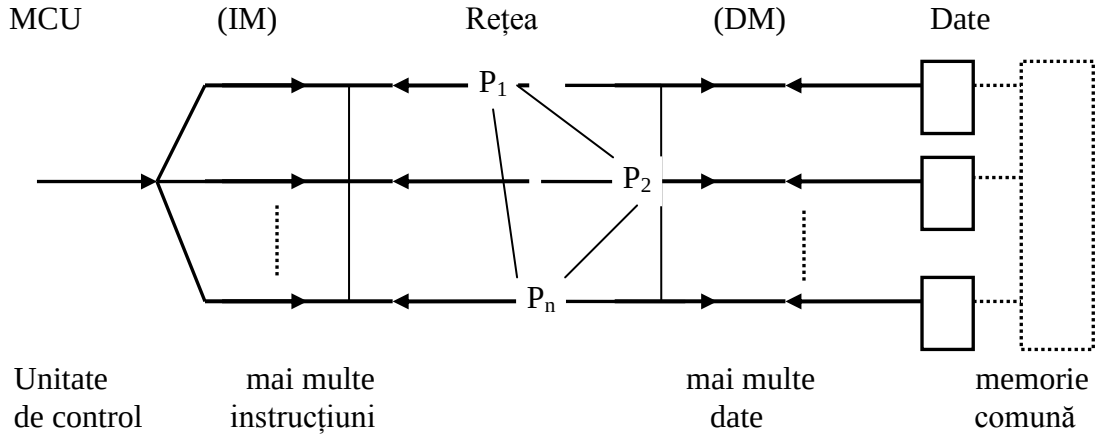
**SISTEME MISD**

Exemplele clasice de sisteme MISD sunt procesoarele pipeline care efectuează operații asupra unui set mic de date. Intr-un procesor pipeline o singură dată este operată de diferite faze ale unei unități funcționale, paralelismul fiind realizat prin simultana execuție a diferitelor etape asupra unui şir de date secvențiale. Figura următoare prezintă schema unui sistem MISD.




**SISTEME MIMD**

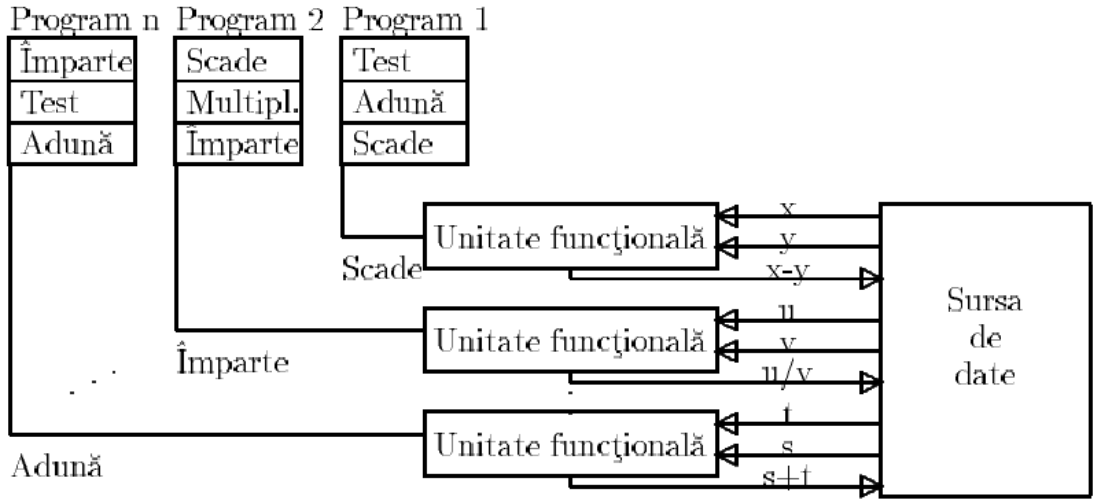
Intr-un sistem MIMD, fiecare procesor poate executa diferite operații pe date distincte de cele ale altor procesoare. Fiecare procesor funcționează independent de celelalte utilizând propriul contor de program şi setul propriu de instrucțiuni. Orice comunicare între două procesoare trebuie specificată explicit. Figura următoare prezintă schema unui sistem MIMD.




Seturile de instrucțiunile corespunzătoare unor procesoare distincte sunt independente unele de altele, execuția unei instrucțiuni nu influențează execuția alteia şi în consecință toate procesoarele pot opera la orice moment. Spre deosebire de sistemul

SIMD, în MIMD, modul de operare este asincron. Fiecare procesor are memoria sa locală, unitate aritmetică, contor de instrucțiuni şi poate comunica cu celelalte procesoare printr-o rețea.

Figura următoare prezintă funcționarea unui sistem MIMD.




In cazul unui SIMD sau unui MISD timpul de execuție a unei operații elementare necesită o unitate de timp, astfel încât este facilitată sincronizarea între elementele de procesare (procesoarele respectă acelaşi tact dat de un ceas comun). Sistemele MIMD permit executarea asincronă a proceselor unei aplicații (fiecare procesor are un ceas propriu).

Alegerea configurației rețelei are o influență importantă asupra vitezei de execuție. Sistemele actuale MIMD au un număr de procesoare mai mic decât cel al sistemelor SIMD.

## INSTRUCȚIUNI CONDIȚIONALE IN SISTEMELE SIMD ŞI MIMD

Execuția unui program pe un procesor într-un SIMD depinde de setul de date asupra căruia se operează. La prima vedere, în cazul unor instrucțiuni condiționale asupra acestor date, execuția pe două procesoare poate să fie diferită. Pentru a rezolva această problemă fiecare procesor dintr-un sistem SIMD deține un registru-permisiune (enable register) pentru calificarea operațiilor de scriere. Numai acele procesoare care au registrul-permisiune setat pe "true" pot scrie rezultatele calculelor în memorie.

Opus acestei situații este comportarea unui sistem MIMD care nu întâmpină dificultăți la întâlnirea expresiilor condiționale: fiecare procesor decide codul instrucțiunii proprii ce se va executa. Dificultăți apar la încercarea de sincronizare a proceselor datorită timpilor diferiți de execuție.

Se consideră, pentru exemplificare, un acelaşi algoritm rulat pe un sistem SIMD şi un sistem MIMD. In primul caz se efectuează un număr de operații independent de valoarea variabilei condiție, pe când în al doilea caz, numărul de instrucțiuni executate variază funcție de condiție.

```
* Secvență program SIMD *
instrucțiune 1
if not conditie then
        enable =False
instrucțiune 2
enable=not enable
instrucțiune 3
instrucțiune 4
enable=True
instrucțiune 5
instrucțiune 6
```

```
* Secvență program MIMD *
instrucțiune 1
if conditie then
instrucțiune 2
else
 {instrucțiune 3
  instrucțiune 4}
instrucțiune 5
instrucțiune 6
```

## SISTEME PARTIȚIONABILE

Un exemplu de sistem partiționabile este SIMD multiplu. Acesta este un sistem de procesare paralelă care poate fi reconfigurat dinamic pentru a opera ca una sau mai multe maşini SIMD independente, de dimensiuni diferite. O maşină multiplă SIMD constă dintr-un număr $n$ de elemente de procesare, o rețea de interconectări şi $q$ unități de control, cu $q < n$.

Un sistem partiționabil este capabil de o partiționare şi repartizare dinamică în maşini multiple. Partiționarea este realizată pe baza rețelei de interconectare. Un asemenea sistem permite:

• abilitatea de partiționare a rețelei în subrețele, fiecare menținându-şi funcționalitatea ca o rețea completă;

• independența submaşinilor, adică nici o maşină să nu interfere în execuția unei aplicații cu altă maşină, fără o instrucțiune adecvată.

Într-o maşină partiționabilă cu mod mixt toate submaşinile pot efectua o comutare independentă şi dinamică între modurile SIMD şi MIMD de paralelism.

Avantajele utilizării unui asemenea sistem sunt multiple:

1. dacă un procesor cedează, numai submaşinile care includ acel procesor sunt afectate;
2. mai multe submaşini pot executa acelaşi program pe acelaşi set de date şi compară rezultatele;
3. accesul simultan în sistem a mai multor utilizatori, fiecare executând diferite programe paralele.

Sistemele partiționabile sunt apropiate de noțiunea de rețele neuronale.

## TIPURI DE MIMD ŞI TRANSPUTERE

Funcție de numărul procesoarelor, sistemele MIMD se clasifică în:

1. sisteme ce utilizează switch-uri (ferme),
2. rețele (cuburi, caroiaj, ierarhie, reconfigurabil).

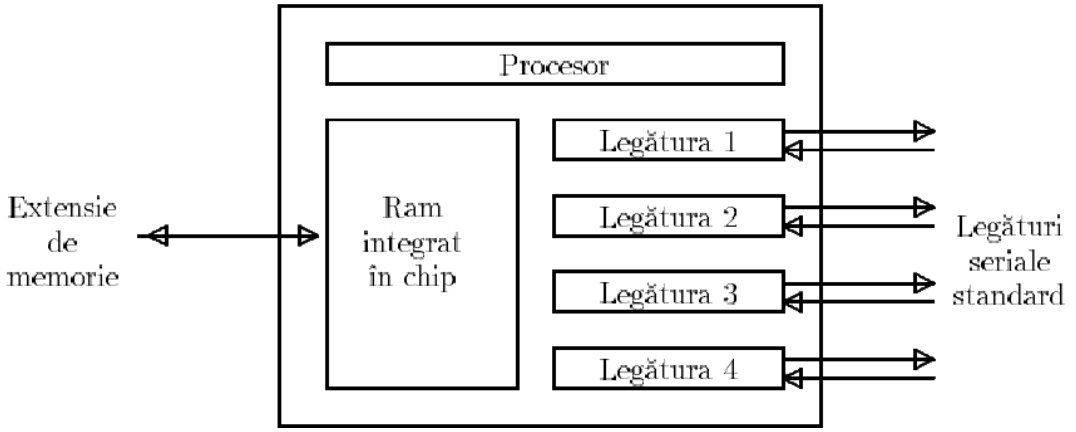
Un MIMD cu comutatoare presupune un număr mic (de ordinul zecilor) de procesoare. Procesoarele sunt conectate printr-o magistrală (bus) sau printr-o rețea de comutatoare (switch-uri). Comutatoarele dintre procesoare asigură comunicarea între procesoare (sistem cu memorie distribuită), iar comutatoarele procesor-memorie asigură transferul de date dintre diverse blocuri de memorie şi procesoare (sistem cu memorie comună). Fiecare procesor este autonom şi poate opera ca şi un calculator independent.

Sistemul este indicat în cazul unui paralelism de program, când problema este divizată într-un număr mic de subprobleme. Au fost construite compilatoare care recunosc comenzile specifice paralelismului într-un program secvențial. Problema hardware asociată cu acest sistem este construirea unei rețele de comutatoare rapide la un cost scăzut.

Un MIMD tip rețea este o maşină cu un număr mare de procesoare identice, conectate printr-o rețea. Sistemele actuale conțin de la 128 la 65536 de procesoare. In mod obişnuit, rețeaua formează un hipercub, însă pentru anumite scopuri practice, se construiesc şi alte conexiuni topologice. Problema hardware asociată cu acest sistem este alegerea unei rețele adecvate din punct de vedere al performanței unui anumit algoritm.

Unitatea indivizibilă, de bază a unui MIMD tip rețea, este elementul de procesare care este construit dintr-un cip microprocesor cu memorie adițională, unitate aritmetică şi facilități de comunicare. Denumirea standard pentru un asemenea "calculator pe un cip", este aceea de trasputer. Procesoarele nu sunt autonome şi trebuie dirijate de un computer, gazdă, separat ("host"). Datorită numărului mare de procesoare, utilizarea transputerului necesită o analiză atentă a problemei care se rezolvă. Adesea este necesară dezvoltarea de noi algoritmi care fac eficientă utilizarea simultană a procesoarelor şi minimizează efortul de comunicare dintre acestea.

În general, un transputer presupune 4 legături fizice. În figura de mai jos este prezentată schematic diagrama unui transputer.




Un exemplu este transputerul T800 ce conține:

1. un procesor pe 32 de biți (de 10 Mips: milioane de instrucțiuni per secundă),
2. 4 kbyte memorie locală (înglobată, on-chip)
3. coprocesor pentru operațiile în virgulă flotantă pe 64 de biți (capabil să efectueze 1Mflop: operații în virgulă mobilă per secundă),
4. interfață de memorie pentru accesarea a până la 4Gbytes memorie externă (off-chip),
5. patru legături de comunicare bidirecțională.

Fiecare legătură este un canal bidirecțional de comunicare. Legăturile dintre transputere pot fi cuplate electronic în orice configurare dorită. Comunicația printr-o legătură poate avea loc simultan cu alte comunicații sau cu efectuarea unor calcule. O maşină care are $4^n$ transputere se numeşte transputer n-dimensional.

## TEHNICA PIPELINE ŞI PROCESOARE PIPELINE

Termenul "pipeline" (conductă) provine din industria petrolieră, unde o serie de produşi hidrocarbonaţi sunt pompaţi printr-o linie de selecţie.

Tehnica pipeline (tehnica conductei) este similară cu cea a unei linii de asamblare dintr-o unitate productivă. La fiecare staţie de lucru se execută un pas al procesului şi toate staţiile lucrează simultan pentru diferite stadii ale produsului. Dacă la fiecare staţie de lucru se consumă un timp t, ritmul de producţie este de un produs per t unităţi de timp, chiar dacă acesta se obţine în t * numărul de staţii de lucru (egal cu ritmul de producţie pentru cazul în care un singur om sau robot deserveşte toate staţiile de lucru).

Se consideră cazul unei singure unităţi centrale, CPU. Părţi ale acesteia care sunt responsabile pentru anumite funcţii, cum ar fi operaţiile aritmetice, pot fi instruite pentru a opera simultan. Pentru realizarea unei singure operaţii, subunităţile sunt executate una după alta. O asemenea operaţie divizată presupune ca rezultatele furnizate de o subunitate să fie obţinute pe baza rezultatelor primite de la precedentul şi, apoi, transmise următoarei subunităţi. Fiecare subunitate este asociată unui procesor.

Fiecare subunitate este specializată pe anumit tip de operaţie. Intr-un procesor "pipeline" data de intrare este privită ca un vector. Operaţia ce urmează a fi efectuată este un vector de instrucţiuni.

Termenul asociat cu staţia de lucru este "pipe". Unitatea care efectuează operaţia este numită linie de "pipe"-uri sau "pipeline".

Parametrul de bază a unui algoritm ce utilizează un pipeline, parametru care influenţează performanţa algoritmului, este lungimea vectorului (setului) de date, adică numărul de elemente asupra căruia calculele sunt efectuate. Parametrii arhitecturali care determină performanţele în procesarea pe un pipeline este timpul unui ciclu, numărul de etape al liniei de procesare (costul de start al acesteia).

De obicei o linie de pipe-uri este dedicată unei singure operaţii pentru un tip specific de date. Un procesor pipeline este unifuncţional dacă este dedicat unei singure funcţii. Un procesor multifuncţional este capabil să execute operaţii diferite (se schimbă sarcina fiecărui procesor sau se interconectează diferite faze ale liniei de pipe-uri). Dacă configuraţia unui pipeline rămâne neschimbată, este static, altfel este dinamic.

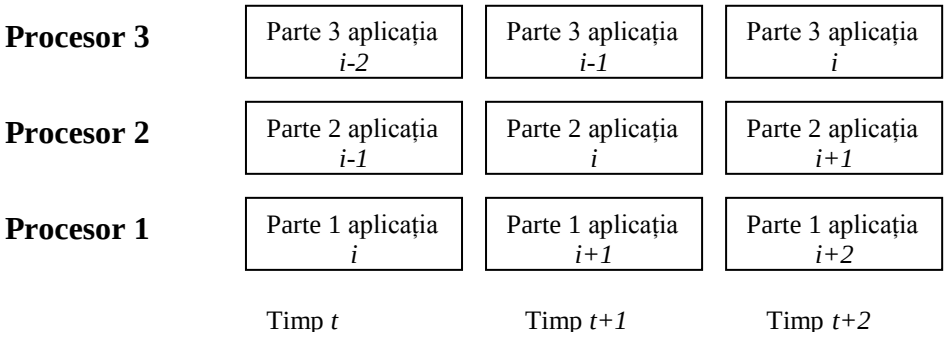
**Exemplul 1**. Se doreşte adunarea a doi vectori de dimensiune n, notaţi a şi b. Se presupune că unitatea centrală poate fi separată în trei subunităţi. Atunci, adunarea vectorilor şi afişarea rezultatului se poate desfăşura, în timp, astfel:

1. în prima unitate de timp, se accesează al;
2. în a doua unitate de timp, se accesează a2 şi simultan b1 este adunat la al;
3. în a treia unitate de timp, se accesează a3, simultan b2 este adunat la a2 şi se afişează al + b1;
4. procesul continuă până când rezultatul este complet afişat.

Timpul este redus la, aproximativ, o treime din timpul necesar pentru unitatea centrală nedivizată.

**Exemplul 2**. Presupunem că o aplicaţie este descompusă într-un număr *p* de procese secvenţiale cu proprietatea că fiecare asemenea proces constă în recepţionarea rezultatelor calculate de procesul anterior şi rezultatele procesului curent sunt utilizate de următorul proces. Aceste procese pot fi executate unul după altul pe un singur procesor sau pot fi alocate la un număr de *p* de procesoare distincte. Dacă aplicaţia este executată de mai multe ori, atunci mai multe procesoare pot fi utilizate în paralel. În

distribuția aplicație din figura de mai jos procesorul secund lucrează asupra celei de a i-a aplicații, în timp ce primul lucrează la aplicația i + 1, iar al treilea la aplicația i -1. In primul ciclu, al doilea şi al treilea procesor sunt blocate deoarece aşteaptă ca valorile lor de intrare să fie produse. Astfel primele două cicluri (timpul sau costul de start) sunt utilizate pentru inițializarea liniei de pipe-uri.

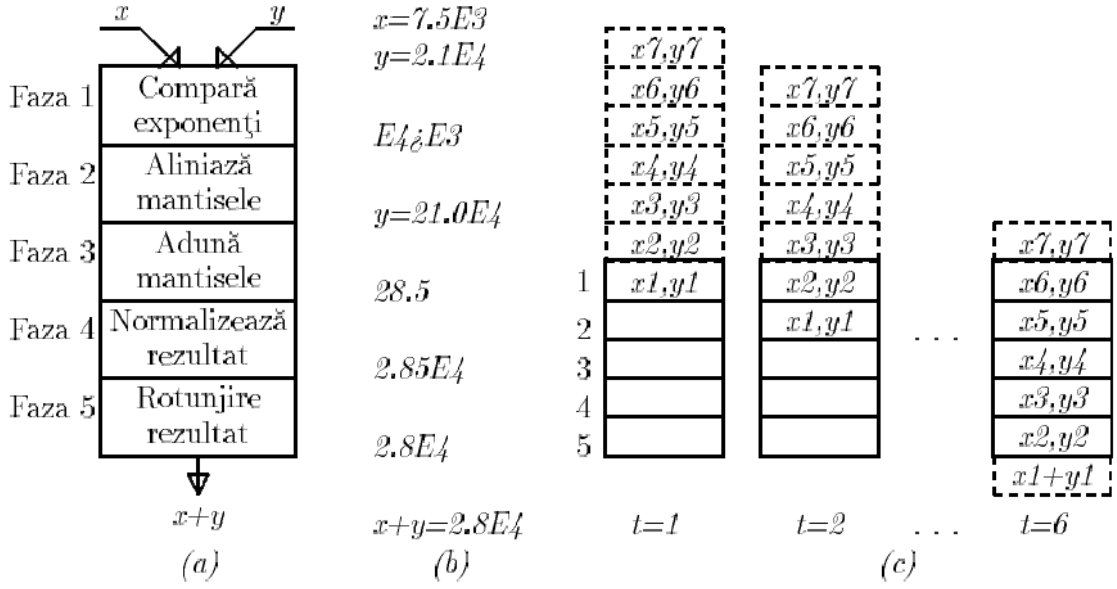



Timpul de execuție a unei singure aplicații va fi similar cu timpul execuției pe un singur procesor. Dacă se execută un număr mare de aplicații pe un procesor pipeline cu *p* procesoare, atunci timpul va fi aproximativ 1/*p* din timpul secvențial (presupunând că fiecare procesor execută procesul repartizat în acelaşi timp ca şi celelalte procesoare).

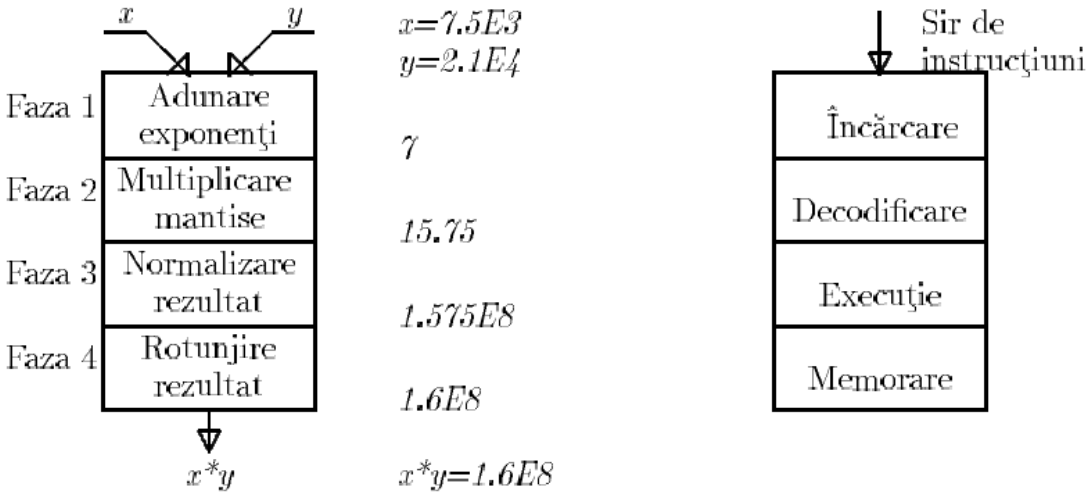
**Exemplul 3**. Operația de adunare a celor doi vectori poate fi divizată ținând seama de etapele adunării a doi scalari:

1. compară exponenții celor doi scalari;
2. aliniază mantisa scalarului cu exponentul cel mai mic;
3. adună mantisele celor doi scalari;
4. normalizează rezultatul;
5. ajustează exponentul sumei.

Modul în care datele trec prin pipeline-ul asociat este evidențiat în figura de mai jos.




**Exemplul 4**. Un exemplu similar de aplicare a tehnicii pipeline este multiplicarea în virgulă flotantă într-un procesor vectorial. Sunt necesare mai multe suboperații (stații de lucru) care sunt prezentate în figura următoare.




**Exemplul 5**. În executarea unei instrucțiuni pot fi izolate patru procese (la nivelul limbajului de asamblare): încărcarea instrucțiunii, decodificare, execuție şi memorare rezultate. Acestor etape li se poate asocia câte un procesor. In primul ciclu, prima, instrucțiune este încărcată de un procesor şi celelalte procesoare sunt neocupate. In ciclul al doilea, instrucțiunea a doua este încărcată, în timp ce prima este decodificată ş.a.m.d. Probleme apar la instrucțiunile condiționale, deoarece este necesară alegerea instrucțiunii care urmează. Una din soluții este utilizarea a două pipeline-uri: când la decodificare se recunoaşte o ramificație, se inițializează cel de al doilea pipe cu instrucțiunile unei ramuri a instrucțiunii condiționale, în timp ce primul pipeline primeşte instrucțiunile celeilalte ramuri.

Nivelurile logice la care se aplică tehnica pipelining sunt:

- la nivel de cuvânt: pipeline aritmetic;
- la nivel de instrucțiune: pipeline asupra instrucțiunilor;
- la nivel de program: macro-pipeline.

Exemplele 1 şi 5 prezintă pipeline-uri la nivel de instrucțiune, iar exemplele 3 şi 4 prezintă un pipeline-uri aritmetice. În cazul în care procesele din exemplul 2 nu sunt reduse la simple instrucțiuni, este vorba despre un macro-pipeline.

Tehnica pipeline la nivel de cuvânt este utilizată la segmentarea unei operații aritmetice, astfel încât procesoare separate sunt responsabile pentru diferite părți ale operației. Pipeline-urile aritmetice sunt utilizate pentru executarea unor operații asupra scalarilor sau vectorilor. Pipeline-ul scalar este utilizat în execuția unor operații aritmetice asupra scalarilor din cadrul unui ciclu. Ciclul asigură faptul că operația este repetată un număr de ori. Liniile de pipe-uri vectoriale sunt destinate unor operații vectoriale. Maşinile care încorporează pipeline-uri vectoriale sunt procesoarele vectoriale.

Aplicarea metodei la nivelul instrucțiunii este des întâlnită. Exemplul clasic este cel prezentat în figura ce ilustrează exemplul 4, coloana din dreapta.

Macro-pipeline poate fi realizat în două moduri:

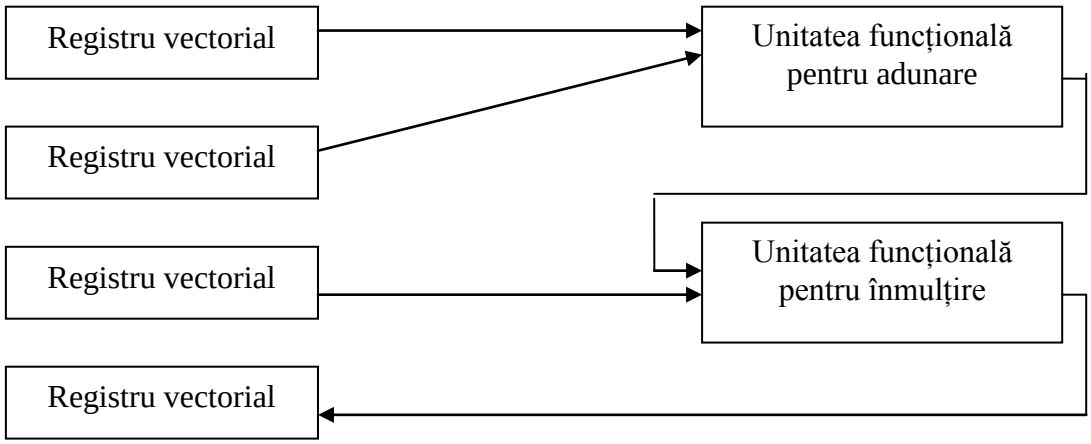
1. un număr de procesoare similare sunt interconectate astfel încât informația să se poată scurge într-o singură direcție. Procesoarele depun un efort de calcul similar;

2. un număr de procesoare ne-similare sunt interconectate astfel încât informația să se poată scurge într-o singură direcție. Procesoarele rezolvă anumite segmente ale problemei. Este posibil ca fiecare procesor să fie construit fizic cu anumită destinație. O asemenea arhitectură este dedicată rezolvării unei probleme specifice date.

Tehnica pipeline a fost utilizată la început pentru creşterea vitezei de execuţie a instrucţiunilor aritmetice fundamentale (procesoarele pipeline au fost în faza iniţială sisteme MISD). Din acest punct de vedere, tehnica pipeline nu este privită ca o tehnică specifică paralelismului. Următorul pas a fost făcut de calculatoarele care au instrucţiuni hardware care acceptă vectori ca operanzi. O instrucţiune hardware vectorială iniţiază scurgerea operanzilor prin pipeline şi, dacă instrucţiunea invocă doi vectori sursă, fiecare segment al liniei acceptă două elemente ale vectorilor, realizează funcţia sa particulară, transmite rezultatul la următorul segment şi primeşte două noi elemente din şirul de operanzi (procesoarele pipeline pot fi privite în această fază ca sisteme SIMD). Astfel mai multe perechi de operanzi sunt procesate concurent de pipeline, fiecare pereche fiind într-o etapă diferită de calcul.

## PROCESOARE VECTORIALE

O generalizare a tehnicii pipeline, implementată astăzi în supercalculatoarele de tip procesor vectorial, este tehnica de înlănţuire. Unitatea aritmetică a CPU este separată în subunităţi care operează simultan şi care pot transmite direct (înlănţuit) rezultatele altor subunităţi, ca în figura de mai jos.




În implementările actuale numărul subunităţilor nu este ridicat (de obicei 12 unităţi independente).

Operaţiile vectoriale pe un calculator vectorial sunt mai rapide decât pe un calculator serial dacă lungimea vectorilor permite depăşirea costului (timpul) de start (numărul de componente este mai mare decât numărul de pipe-uri).

Pentru un procesor vectorial dat se stabileşte o mulţime de instrucţiuni care includ operaţii vectoriale. Unitatea de control a procesorului vectorial care întâlneşte o asemenea instrucţiune trimite primul element al vectorului spre linia de pipe-uri specifică. Dacă *t* unităţi de timp sunt necesare pentru procesarea unui pipe, atunci, după un timp *t*, unitatea de control trimite un al doilea element de vector spre aceeaşi linie de pipe-uri.

Dacă asupra rezultatului unei operaţii vectoriale se face o altă operaţie vectorială se utilizează avantajele tehnicii de înlănţuire. Odată obţinută prima componentă a vectorului rezultat al primei operaţii, ea este trimisă direct la linia de pipe-uri corespunzătoare noii operaţii. Intr-un sistem fără înlănţuire, vectorul rezultat trebuie în prealabil stocat, se aşteaptă terminarea, în totalitate, a primei operaţii, apoi se trimite

prima componentă spre linia a doua de pipe-uri. Diferența între cele două situații constă în pierderea timpului prin startul întârziat celei de a doua linii (în varianta neînlănțuită).

Intr-un procesor vectorial "memorie-la-memorie", în fiecare ciclu al unei operații vectoriale, doi operanzi, componente a doi vectori, sunt citiți din memorie. Fiecare din subunitățile liniei de pipe-uri operează pe anumite elemente ale vectorilor şi un element din vectorul rezultat este scris în memorie. Într-un proces vectorial "bazat pe registru" există o singură cale de acces la memorie. Se permite stocarea de către procesoare a operanzilor şi rezultatelor în anumiți regiştrii (vectoriali). Fiecare registru conține câte un vector suficient de lung pentru a asigura funcționarea liniei de pipe-uri la întreaga capacitate.

O performanță deosebită față de maşinile seriale se obține când procesorul vectorial este utilizat în calcule repetitive şi mai puțin în operații condiționale.

Deşi procesoarele vectoriale aduc o îmbunătățire substanțială față de calculatoarele secvențiale, ele nu au viitor. Apare aceeaşi problemă, ca şi în cazul serial, de limitare a vitezei de procesare datorată posibilităților fizice.

## PROCESOARE MATRICEALE

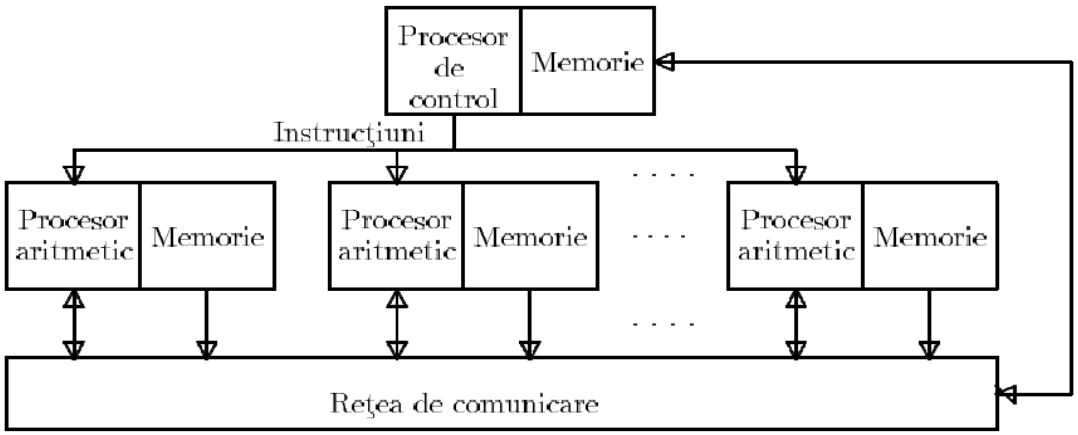
Un procesor matriceal este un agregat constituit din mai multe procesoare identice care execută aceeaşi instrucțiune simultan asupra unor date locale. Este un SIMD care încorporează un număr mare de procesoare conectate într-o topologie particulară care se comportă sincron sub acțiunea unei singure unități de control a cărei funcționalitate este direcționarea activității procesoarelor. Activitatea procesoarelor este sincronizată pentru execuția aceleiaşi instrucțiuni ce utilizează însă date diferite.

Modul de funcționare este asemănător activității dintr-o clasă de elevi atunci când profesorul de sport comandă o săritură şi toți elevii clasei părăsesc podeaua.

Referitor la modul de organizare a memoriei se disting două tipuri:

• sistem multiprocesor cu cuplare puternică, când memoria este comună pentru toate procesoarele. Memoria este divizată în module independente şi module de interacțiune;

• sistem multiprocesor cu cuplare slabă, când fiecare procesor are propria sa memorie şi comunicarea dintre procesoare se efectuează printr-o rețea de interconectare. Figura de mai jos prezintă schița scurgerii informației printr-un procesor matriceal slab cuplat.



Un procesor matriceal este construit din procesoare simple şi ieftine. Cea mai comună organizare a unui asemenea sistem paralel este matricea bidimensională cu elemente de procesare conectate cu cei mai apropiaţi vecini (tip grilă).

Procesorul de control este el însuşi un calculator: posedă memorie locală, unitate aritmetică, regiştrii şi unitate de control. Unitatea de control stabileşte dacă instrucţiunea este o operaţie multi-date. Dacă operaţia nu este multi-date, procesorul de control execută operaţia. Dacă este o operaţie cu date multiple, instrucţiunea este transmisă procesoarelor aritmetice, fiecare procesor deţinând o parte din data multiplă care va fi operată, în unitatea sa locală de memorie. In cazul cel mai simplu, un procesor aritmetic deţine o singură dată din mulţimea de date, dar, dacă dimensiunea datei-multiple depăşeşte numărul procesoarelor, poate conţine o submulţime de date. Toate procesoarele aritmetice primesc simultan instrucţiuni de la procesorul de control, iar operaţia asupra datei-multiple se face în paralel. Procesoarele aritmetice sunt, astfel, "sclavii" procesorului de control. Diferenţa crucială între procesorul controlor şi procesoarele aritmetice este aceea că primul este singurul capabil să interpreteze instrucţiunile condiţionale.

**Observaţie**. Termenul de procesor pipeline descrie o maşină bazată pe principiul scurgerii informaţiei printr-o bandă, pe când termenul de procesor matriceal desemnează un sistem în care procesoarele execută simultan aceeaşi instrucţiune.

**Exemplu**. Se adună doi vectori de 64 de componente. Utilizând un procesor matricial cu 64 de elemente de procesare, adunarea se poate realiza în trei paşi. Fiecare procesor primeşte două componente, una dintr-un vector şi componenta, corespunzătoare în indice, a celui de-al doilea. Unitatea de control comandă fiecărui procesor adunarea elementelor sale şi scrierea rezultatului în componenta corespunzătoare a vectorului final.

Dacă un procesor necesită un anumit timp $t$ pentru a executa o instrucţiune complexă, atunci $p$ procesoare execută instrucţiunea în acelaşi timp, dar timpul per instrucţiune este (aproximativ) redus cu factorul $1/p$.

La evaluarea unei expresii condiţionale procesorul de control are posibilitatea de a selecta procesoarele care să execute următorul set de instrucţiuni, printr-o "mască". Un procesor acceptă instrucţiuni de la procesorul de control numai dacă bitul corespunzător din mască este setat. Procesorul de control dispune setarea şi resetarea măştii.

## SISTEME CU MEMORIE COMUNĂ

În funcţie de modul de organizare a memoriei, sistemele MIMD se clasifică în
- sisteme cu memorie comună,
- sisteme cu memorie distribuită.

Într-un sistem cu memorie comună, fiecare procesor are acces, printr-un anumit mecanism, la o memorie globală. Procesoarele comunică prin obiectele plasate în memoria comună. Probleme apar referitor la controlul accesului la memorie. O reţea conectează fiecare procesor la fiecare bancă de memorie, însă nici un procesor sau bancă de memorie nu sunt conectate direct unul cu altul. Memoria comună este global adresabilă de toate procesoarele.

Există mai multe variante de interconectare în reţea:

1. printr-o singură magistrală comună, un bus comun: fiecare procesor accesează memoria comună concurând cu celelalte procesoare pentru bus şi ocupând magistrala în timpul tranzacţiei de memorie. Astfel, la un moment dat, doar un procesor poate accesa

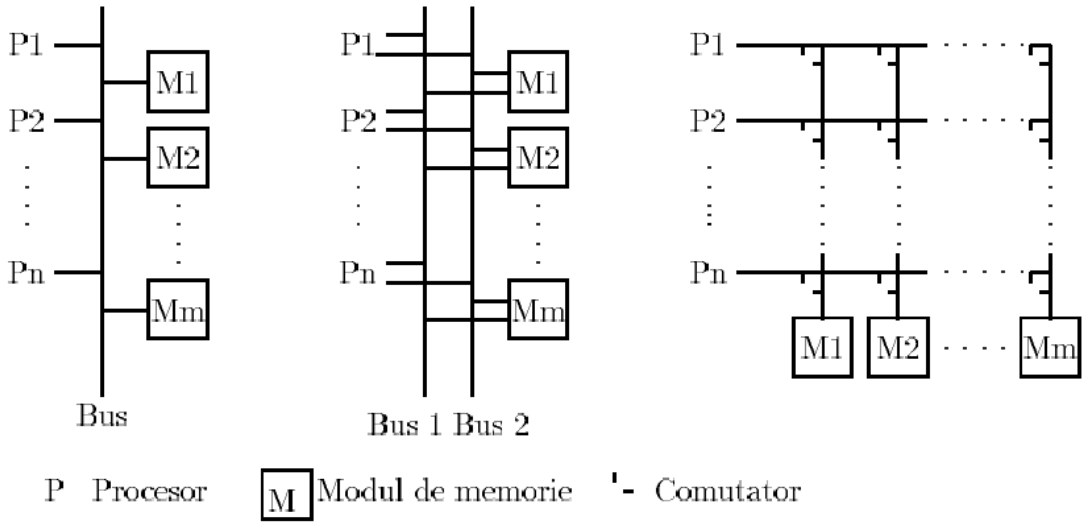
memoria comună. Maşini de acest tip sunt echipate, în mod uzual, cu o memorie specială, numită cache. Un cache poate fi privit ca o colecție de regiştrii rapizi în care un registru memorează adresa şi valoarea unei locații de memorie. De obicei un cache separat este asociat cu fiecare procesor din rețea. Locațiile frecvent referite de către un procesor sunt copiate în cache. Prin acest procedeu cerințele de citire ale unor asemenea locații sunt satisfăcute de către cache şi nu necesită referințe în memoria comună. Se reduce timpul de accesare pentru o locație individuală de memorie şi se îmbunătățesc performanțele sistemului reducând comunicările prin bus;

2. printr-o rețea cu magistrală multiplă: se permite accesul simultan a unui număr de procesoare egal cu numărul de magistrale existente la diferitele bănci de memorie;

3. prin linii de comutatoare (crossbar switch): se permit interconectări arbitrare între procesoare şi băncile de memorii;

4. prin comutatoare multietape: interconectări de acest tip caută balansarea cost-performanță. Cea mai comună conectare de acest tip este cea descrisă prin metoda amestecării perfecte.

Primele trei modele sunt schițate în figura de mai jos.




Modelul ideal al "paracomputerului" constă într-un SIMD care conține un număr nemărginit de procesoare care toate pot accesa o memorie comună fără nici un conflict şi fără cost în timp. Cauzele obişnuite ale ineficienței unui algoritm în implementare (întârzieri datorate transmiterii de date sau conflictelor de acces la memorie) sunt anulate, astfel încât paralelizarea completă a algoritmului conduce la performanța optimală. Opuse acestui model sunt calculatoarele paralele de astăzi cu număr fixat de procesoare şi de obicei mic relativ la dimensiunea problemelor.

Modelul RAM (Random Acces Memory) al calculatoarelor secvențiale a fost generalizat la PRAM (Parallel Random Acces Memory), model în care mai multe procesoare pot accesa simultan o memorie comună. Acest model are mai multe variante, în funcție de modul în care se realizează scrierea unei variabile comune:

1. EREW PRAM (Exclusive Read Exclusive Write): operația de citire şi cea de scriere este nedivizibilă între procesoarele sistemului;

2. CREW PRAM (Concurent Read Exclusive Write): față de modelul anterior se permite citirea simultană a unei variabile;

3. CRCW PRAM (Concurent Read Concurent Write): se permite atât citirea şi cât scrierea simultană. Scrierea se efectuează după anumite reguli:

(a) Common CRCW PRAM: modelul cere ca toate procesele care scriu simultan o valoare într-o locaţie de memorie să scrie aceeaşi valoare;

(b) Priority CRCW PRAM: se asociază un index fiecărui procesor şi când se încearcă scrierea simultană a mai multor valori provenite de la procese distincte, valoarea scrisă va fi cea asociată procesorului cu cel mai mic index;

(c) Arbitrary CRCW PRAM: valoarea memorată provine de la un procesor ales în mod aleator dintre cele care încearcă scrierea simultană.

PRAM este cel mai puternic model utilizat în construcţia algoritmilor paraleli. Modelul PRAM neglijează orice constrângere hard. Astfel în modeleul PRAM sunt posibile toate legăturile între procesoare şi oricare locaţie de memorie. Fiecare procesor este o maşină RAM cu operaţiile şi instrucţiunile uzuale. Modelul PRAM face parte din categoria SIMD. Procesoarele execută simultan acelaşi program pe date diferite. Execuţia pe un procesor a programului depinde de identificatorul de procesor. Toate procesoarele operează sincron sub controlul unui ceas comun.

## SISTEME CU MEMORIE DISTRIBUITĂ

Intr-un sistem cu memorie distribuită, fiecărui procesor îi este destinată o parte din memorie. Astfel, fiecare procesor are o memorie locală (privată). Procesorul împreună cu memoria locală constituie un element de procesare.

Sistemele cu memorie distribuită sunt de obicei maşini bazate pe transputere. Comunicarea se face pe baza conectărilor fizice dintre procesoare şi presupune transmiterea unor mesaje. Interconectarea tuturor procesoarelor, dacă numărul acestora este mare, este practic imposibilă. Soluţia practică adoptată este conectarea fiecărui procesor la un număr mic de alte procesoare. Se formează astfel structuri deosebite pentru reţeaua de interconectare. Structurile standard sunt laticea, arborele binar, hipercubul, structura liniară sau circulară.

Procesoarele sunt legate prin canale de comunicare. Aceste canale permit ca mesajele să fie transmise de la un proces şi să fie recepţionat de altul. Transmiterea şi recepţionarea se efectuează cu ajutorul unor primitive de tipul:

send *expresie* to *destinatie*

receive *variabila* from *sursa*

Adiţional, se transmit informaţii asupra subiectului mesajului astfel încât cele două părţi comunicante să cadă de acord asupra ceea ce se comunică. Un caz simplu este transmiterea unui întreg: procesul receptor trebuie să fie pregătit să recepţioneze un întreg.

Există mai multe variante de utilizare a canalelor de comunicare:

1. prin numirea directă a procesului emitent/receptor;
2. printr-o cutie poştală generală;
3. prin numirea canalului de comunicare.

În primul caz, comunicarea este viabilă dacă cele două procese invocate sunt pregătite să comunice, adică în procesul emitent există o primitivă send *expresie* to proces2, iar în procesul receptor există un receive *variabila* from proces1 (necondiţionat de exemplu printr-un if). Este necesară o evidenţă clară a numărului de primitive *send* şi *receive*.

O cutie poştală poate fi destinaţia oricărui *send* şi sursa oricărui *receive* care numesc acea cutie poştală. Primitivele au forma: *send* <*expresie*> to *mailbox1*, respectiv *receive* <*variabila*> from *mailbox1*. Procesul emitent nu controlează care proces recepţionează mesajul. Această formă de comunicare este utilizată în majoritatea limbajelor logice concurente: expeditorul adaugă mesaje la sfârşitului unui şir de mesaje, iar receptorul inspectează şirul, extrăgând eventual un mesaj dorit.

Un canal de comunicare este definit ca o legătură unidirecţională între două procese (unul este emiţător, iar altul receptor). Canalele de comunicare dintre procese pot fi statice când legăturile sunt setate în timpul compilării, sau pot fi dinamice când sunt create şi distruse în timpul execuţiei unei aplicaţii. Primitivele asociate transmiterii prin canale sunt *send* <*expresie*> via *canal2*, respectiv *receive* <*expresie*> via *canal1*.

Transmiterea de mesaje implică cooperarea ambelor părţi. Comunicarea se poate face:

1. unu la unu: prin specificarea identificatorului de proces, definirea unui canal distinct între procese sau definirea celor două procese ca singurele părţi care pot utiliza cutia poştală (tata dă lui Ioan un vas, dacă Ioan este pregătit să-l primească) ;
2. mai mulţi la unul: un singur proces este pregătit să accepte mesaje de la o mulţime de procese (mai multe persoane care strâng vasele, unul singur care le spală);
3. unul la mai mulţi: mai mulţi receptori potenţiali şi un singur emiţător. Există două variante:
   (a) unul dintre receptori va accepta comunicarea de la emiţător (mai mulţi spălători de vase, unul singur care aduce vasele);
   (b) toţi receptorii vor accepta comunicarea (tata spune copiilor să tacă);
4. mai mulţi la mai mulţi: mai mulţi receptori potenţiali şi mai mulţi emiţători, în variantele:
   (a) un receptor acceptă comunicarea de la un emiţător;
   (b) toţi receptorii acceptă comunicarea de la oricare dintre emiţători;
   (c) anumite combinaţii între receptori şi emiţători vor fi stabilite într-o comunicare.

## CLASIFICAREA REŢELELOR DE INTERCONECTARE

Într-un sistem paralel cu memorie distribuită procesoarele sunt conectate printr-o reţea prin intermediul căreia comunică.

Intr-un sistem ideal fiecare element este conectat cu oricare altul. In practică, acest tip de interconectare este posibil numai pentru un număr redus de procesoare.

In construirea calculatoarelor paralele se ţine seama de clasa de probleme care urmează a fi rezolvate cu maşina respectivă şi de limitarea numărului de conexiuni ale fiecărui element de procesare. Reconfigurarea (logică) este permisă, astfel încât este posibilă construirea unei varietăţi mari de configuraţii şi schimbarea lor pe parcursul calculului. Din păcate, reconfigurare are un efect negativ asupra timpului de calcul: dacă cerinţele de interconectare pentru un algoritm dat nu corespund configuraţiei reţelei, atunci comunicarea datelor afectează viteza de calcul.

Reţele de interconectare pot fi clasificate în trei categorii:

1. fiecare element este conectat cu un anumit număr de alte elemente. Structura reţelei depinde de problema care se rezolvă;
2. fiecare element este conectat cu oricare altul prin intermediul unei punţi de legătură, numită "switchboard" (punte de comutatoare). Prin această modalitate de

conectare se garantează că într-un număr mic de paşi pot fi conectate oricare două procesoare. Se utilizează şi conexiunile directe cu procesoarele învecinate, legătura fiind avantajoasă pentru un număr mare de algoritmi paraleli;

3. fiecare element este conectat direct cu oricare altul: această construcţie este, în prezent, practic inexistentă, exceptând maşinile cu număr mic de procesoare. Există mai multe încercări de conectare completă a elementelor de procesare:

(a) prin magistrală (bus): costul unei asemenea interconectări este redus, dar nu este practică pentru un număr mare de procesoare, deoarece apar întârzieri la transmiterea de date de la un proces la altul simultan. Dacă toate cele *n* procesoare doresc să comunice simultan, toate cele *n* comunicări vor fi seriale şi unul dintre procesoare va fi întârziat cu un timp egal cu comunicările celorlalte *n* - 1 procesoare;

(b) prin conexiuni directe: fiecare procesor este conectat la oricare alt procesor utilizând *n*(*n*-l) legături unidirecționale. Avantajul este posibilitatea de comunicare simultană a procesoarelor, adică nu există întârzieri datorate rețelei. Când *n* este mare, costul unei asemenea conectări este foarte mare. Dacă, de exemplu *n*=1024, sunt necesare 1.147.552 legături. Cum un procesor este implantat pe un singur cip, numărul de pini ceruți pentru realizarea tuturor conectărilor depăşeşte posibilitățile tehnologiilor actuale.

Rețeaua de interconectare poate fi specificată printr-un graf $G = (N, E)$, în care fiecare nod *i* din *N* reprezintă un procesor, iar fiecare latură (*i*,*j*) din E reprezintă o legătură uni- sau bi- direcțională între procesoarele *i* şi *j*.

Interconectarea este caracterizată prin doi parametri:

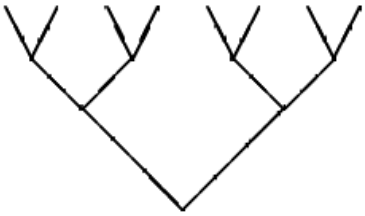
1. diametrul rețelei care este distanța maximă parcursă de un mesaj între două noduri (elemente de procesare) arbitrare ale rețelei;

2. gradul unui nod care este numărul de procesoare cu care nodul este conectat (măsoară costul interfeței hardware a fiecărui nod).

**Exemplul 1**. Structura liniara de interconectare. In structura liniară oricare procesor $P_i$ este conectat direct cu vecinii săi $P_{i-1}$ şi $P_{i+1}$. Utilizarea unei asemenea structuri este indicată pentru anumiți algoritmi, ca de exemplu sortarea par-impar. Diametrul rețelei cu *p* procesoare este *p*-1, iar gradul unui nod este doi.

**Exemplul 2**. Structura ciclica de interconectare. Conectarea ciclică a *n* procesoare este caracterizată prin diametrul n/2 şi gradul unui nod este 2.

**Exemplul 3**. **Structura de interconectare tip arbore binar**. Intr-un sistem construit pe baza unui arbore binar, fiecare procesor de la un nivel *k* este interconectat direct cu exact două sau zero procesoare de la nivelul *k* + 1. Fiecare procesor de la nivelul *k* + 1 este conectat direct cu un singur procesor de la nivelul *k*, exceptând procesorul de la nivelul zero, considerat nod rădăcină.



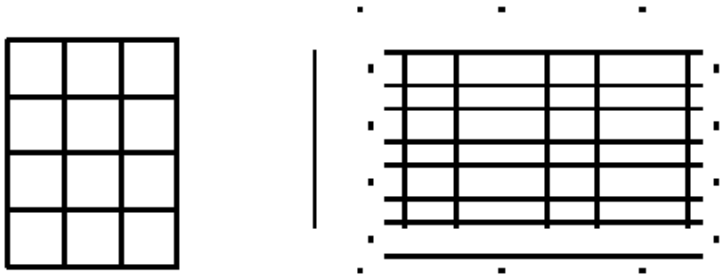
O asemenea structură de tip arbore de ordin *n*, notată $A_n$, (cu $2^n$ - 1 noduri), este, de exemplu, utilizată cu succes pentru adunarea a $2^n$ - 1 numere. În fiecare element de la nivelul *n* - 1 se memorează o valoare numerotată de la 1 la $2^{n-1}$ - 1. Fiecare procesor

aflat în nodurile interne ale arborelui are de efectuat o operație de adunare, iar nodul rădăcină furnizează rezultatul final.

Arborii binari sunt asociați unor algoritmi fundamentali, cum sunt, de exemplu, cei de sortare, căutare, evaluare a expresiilor algebrice ş.a.m.d. În algoritmii de evaluare a expresiilor aritmetice, fiecare procesor aflat într-un nod intern al arborelui efectuează o anumită operație, iar nodul rădăcină furnizează rezultatul final.

Numărul minim de niveluri ale arborelui indică numărul de etape în care poate fi evaluată o expresie aritmetică, în paralel. Numărul de etape este, evident, cel mult egal cu cel necesar într-o evaluare serială. A

**Exemplul 4**. **Structurile de tip grilă şi tor**. Intr-o grilă de *n* * *m* noduri-procesoare fiecare element este conectat cu patru vecini, ca în figura de mai jos.



In cazul particular al aceluiaşi număr de procesoare în ambele direcții, *n*, structura este numită latice şi este notată $L_n$. Dacă dimensiunea *n* este pară, laticea poate fi divizată în patru sub-rețele având aceleaşi proprietăți ca şi laticea inițială. Structura este des întâlnită în sistemele MIMD, deoarece permite, cu uşurință, reconfigurare a logică la structura de interconectare cerută de problema ce se rezolvă. O asemenea rețea cu *p* noduri are diametrul $\sqrt{p}$ şi gradul maxim al unui nod egal cu patru. Drumul mesajului de la un nod sursa la un nod destinație este numărul de mişcări pe linii şi coloane pe care le efectuează mesajul.

Apropiată structurii de latice este torul. Elementul de procesare corespunzător punctului (*i*,*j*) din tor, $P_{ij}$, este conectat cu

$$P_{ij-1}\ P_{ij+1}\ P_{i-1j}\ P_{i+1j}$$

unde $i \pm 1$ şi $j \pm 1$ sunt consideraţi modulo *n*, ca în conectarea ciclică.

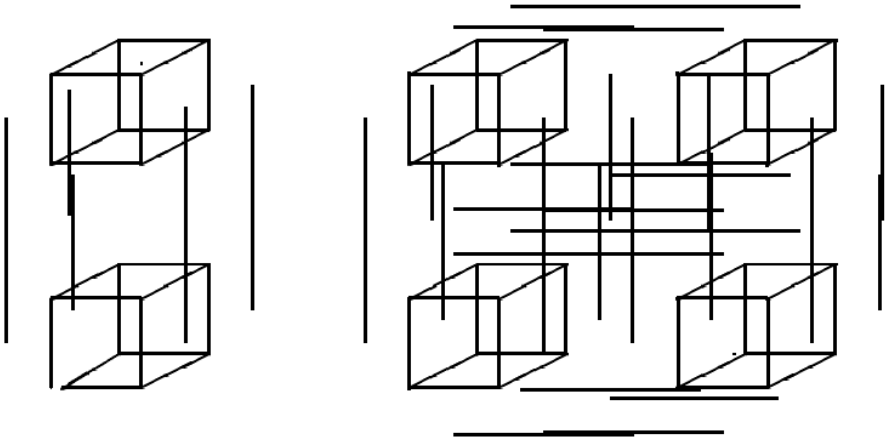
S-au construit de asemenea rețele cu conectarea celor mai apropiați opt vecini.

Prin conectarea vecinilor prin comutatoare se obține o rețea în X (X-net).

**Exemplul 5**. Structura de tip hipercub. Cele $2^n$ vârfuri şi $2^{n-1}n$ laturi ale unui cub *n*-dimensional constituie nodurile-elemente, respectiv interconectările unui hipercub, notat $H_n$.

O structură tip hipercub *n*-dimensional consistă în $2^n$ procesoare interconectate într-un cub *n*-dimensional după cum urmează. Fie $b_{n-1}\ b_{n-2}$ ... ba reprezentarea binară a lui *i*, unde $0 \le i < 2^n$. Atunci procesorul $P_i$ este conectat cu procesorul $P_j$ dacă *j* în reprezentare binară diferă într-un singur bit față de *i*, adică $j_{(10)} = b_{n-1}$ ... $\overline{b_k}$ ... $b_0$, $\overline{b_k} = 1 - b_k$, $0 \le k \le n-1$ dacă $i_{(10)} = b_{n-1}$ ... $b_k$ ... b0. Pentru cazurile particulare n= 1,2,3,4, reprezentările hipercubului sunt cele din figura de mai jos.

Hipercubul este o structură recursivă. Se observă că $H_{n+l}$ poate fi obținut din $H_n$ prin legarea nodurilor corespunzătoare a două copii ale lui $H_n$. Bitul semnificativ a identificatorilor primului cub $H_n$ este 0, respectiv 1 la copia acestuia. De exemplu, pentru n = 4,5 interconectările sunt prezentate în figura de mai jos.



Diametrul unui hipercub *n* dimensional este *n* (deoarece distanța dintre două procesoare este egală cu numărul de poziții binare în care cei doi identificatori diferă), iar gradul fiecărui nod este tot *n*.

Calculatoarele paralele cu topologie de tip hipercub sunt comecializate la momentul actual ca sisteme MIMD. Dimensiunea curentă este $n = 10$.

Apropiată de structura tip hipercub este cea a unui sistem cu transputere *n*-dimensional ce conține $4^n$ transputere, are gradul unui nod egal cu 4, iar diametrul $d_n$ satisface relația recursivă $d_n = d_{n-1} + 1$.

## SISTEME GAZDĂ

Pentru a menține funcționalitatea unui sistem în regim secvențial masivele de procesoare sunt privite ca un calculator independent care este "ataşat" unei aplicații lansate pe un calculator serial aflat într-o aceeaşi rețea de interconectare.

O cale directă pentru a permite unui utilizator să se cupleze la maşina paralelă este apelarea unor programe utilitare care permit utilizatorului să folosească calculatorul paralel pentru scopul său. Aceste programe rulează sub controlul sistemului de operare al calculatorului gazdă. Ele sunt lansate printr-un fel de interpretor de comenzi.

Complexul calculator - utilitare este numit sistem gazdă (host).

La nivelul sistemului de operare a maşinii gazdă se definesc:

1. nucleul: care permite intercomunicarea dintre procese;

2. kernel: este o extensie a nucleului pentru funcții precum crearea şi distrugerea obiectelor definite de procese, asocierea obiectelor la spațiul adresabil, propagarea evenimentelor speciale (întreruperi);

3. Pose (Parrallel Operating System Extension): se ocupă de organizarea memoriei şi a proceselor, manipularea fişierelor, a dispozitivelor de intrare/ieşire;

4. aplicația-utilizator: determină complexitatea şi distribuirea proceselor pe nodurile masivului de procesoare.

Sistemul gazdă creează un model de acțiune a unor procese virtuale pe baza unui soft ce acționează ca un plan între aplicație şi maşina paralelă.

## Capitolul 2

# PROGRAMARE PARALELĂ

## PROCESE CONCURENTE

Un program este o descriere formală a unor acţiuni şi date conform unui formalism convenţional oarecare. Un proces este o succesiune de acţiuni care sunt executate în mod secvenţial. Un proces este astfel activitatea rezultată ca urmare a execuţiei unui program de către un procesor. Un program poate reprezenta descrierea unui număr oarecare de procese şi mai multe procese pot fi executate pe acelaşi procesor (dispozitiv care execută instrucţiunile în mod secvenţial, într-o succesiune dată).

Un program secvenţial este descrierea unui singur proces. Un program concurent descrie mai multe procese care vor fi executate în mod concurent pe un sistem de calcul. Mai multe procese se execută în mod concurent (sunt concurente sau paralele) dacă executarea lor se suprapune în timp. Două procese sunt concurente dacă prima instrucţiune a unui proces este lansată înainte de încheierea ultimei instrucţiuni a celuilalt proces.

## MULTIPROGRAMARE ŞI MULTIPROCESARE

Sistemele cu multiprogramare permit prezenţa simultană în memoria centrală a mai multor programe secvenţiale (procese) ce se execută în paralel permiţând folosirea în comun a resurselor sistemului cu scopul îmbunătăţirii gradului lor de utilizare. În aceste sisteme concurenţa are loc între programe diferite şi nu între procese provenind din acelaşi program.

Paralelismul poate fi:

• **logic**, când există un singur procesor care este atribuit alternativ proceselor. La un moment dat se execută fizic o acţiune corespunzătoare unui singur proces. Având în vedere faptul că, alternativ, se execută acţiuni corespunzătoare diferitelor procese, acestea se desfăşoară concurent (se lansează instrucţiunile unui proces înainte de a se fi executat toate instrucţiunile corespunzătoare celorlalte). Aceste procese sunt multiprogramate pe un sistem monoprocesor;

• **fizic**, când fiecărui proces îi este atribuit în exclusivitate câte un procesor. În acest caz, la un moment dat, se desfăşoară efectiv instrucţiuni corespunzătoare mai multor procese. Funcţie de sistemul paralel utilizat denumirile sunt următoarele:

1. dacă procesoarele sunt legate la o memorie comună prin intermediul căreia se poate realiza schimbul de informaţie între ele, sistemul este numit multiprocesor, iar procesele sunt multiprocesate;

2. dacă se utilizează un sistem distibuit, format din noduri (unul sau mai multe procesoare legate la o memorie comună) legate între ele prin canale de comunicaţie, sistemul este numit reţea.

## COMUNICARE ŞI SINCRONIZARE

Problemele executate sub controlul unui sistem cu multiprogramare sunt exemple de procese paralele independente. In majoritatea cazurilor însă, natura

problemei de rezolvat impune interacțiunea între procesele unui program concurent din următoarele motive:

• utilizarea î comun de către procese a unor resurse cum ar fi memoria comună, echipamente periferice, zone tampon etc;

• cooperarea proceselor în sensul că un proces foloseşte anumite date rezultate din activitatea altuia.

Există două forme de interacțiune între procese paralele exprimate în următoarele primitive:

1. **comunicarea** între procese distincte (transmiterea de informații între procese);

2. **sincronizarea** astfel încât procesele să aştepte informațiile de care au nevoie şi nu sunt produse încă de alte procese (restricții asupra evoluției în timp a unui proces).

Primitivele de sincronizare sunt operații pe care nucleul sistemului de programare concurentă le pune la dispoziția programatorului în vederea rezolvării problemelor de sincronizare. Nucleul reprezintă o interfață între program şi suportul fizic.

Cele trei forme acceptate de sincronizare sunt următoarele:

1. excluderea mutuală: se evită utilizarea simultană de către mai multe procese a unui resurse critice. O resursă este critică dacă poate fi utilizată doar de singur proces la un moment dat;

2. sincronizarea pe condiție: se amână execuția unui proces până când o anumită condiție devine adevărată;

3. arbitrarea: se evită accesul simultan din partea mai multor procesoare la aceeaşi locație de memorie. În acest caz se realizează a secvențializare a accesului, impunând aşteptarea până când procesul care a obținut acces şi-a încheiat activitatea asupra locației de memorie.

Punctele dintr-un program unde elementele de procesare comunică între ele sunt numite puncte de interacțiune. Un punct de interacțiune împarte procesul în două etape. Comunicarea permite ca execuția operațiilor pe un procesor să fie influențată de execuția pe alt procesor. La sfârşitul primei etape procesoarele comunică între ele, după care trec într-o a doua etapă ce utilizează datele comunicate.

În execuția unui program paralel timpul asociat unei etape pentru un procesor oarecare este o variabilă aleatoare. Motivele sunt multiple:

• multi procesorul poate fi constituit din procesoare cu viteze diferite;

• operațiile efectuate de un procesor pot fi Întrerupte prin sistemul de operare;

• timpul de procesare pentru un element poate să depindă de datele de intrare.

**Sincronizarea în sistemele paralele cu memorie comună**

Într-un sistem paralel cu memorie comună procesele au acces la variabilele comune pe care le pot citi şi scrie. În acest caz sincronizarea este cerută pentru a preveni rescrierea unei date de către un proces înainte ca alt proces să efectueze citirea informației anterioare.

Intr-un program paralel asincron, procesele nu aşteaptă introducerea datelor, ci continuă corespunzător informației care este conținută curent în variabilele globale. Într-un MIMD fiecare procesor va opera sub controlul unui ceas separat. Dacă este necesară accesarea unei date de către un procesor responsabilitatea faptului că valoarea datei este cea corectă revine utilizatorului.

Sincronizarea poate fi implementată cu ajutorul unor semafoare sau unor monitoare prin intermediul cărora accesarea unei variabile devine o operație indivizibilă: dacă mai multe proces încearcă să acceseze în acelaşi tip o variabilă comună, atunci numai unul singur va avea succes. O operație indivizibilă este una care odată pornită nu poate fi întreruptă (de exemplu tipăririle sau asignările). Secțiunile de cod care sunt protejate la intervenția mai multor procese simultan sunt numite secțiuni critice. O secțiune critică este tratată ca o operație indivizibilă (sincronizare implicită). Cu ajutorul secțiunilor critice se poate defini excluderea mutuală: numai un proces este admis a se afla în interiorul unei secțiuni critice la un moment dat, iar dacă două sau mai multe procese solicită simultan să intre într-o secțiune critică, alegerea uneia dintre ele se face într-un interval finit de timp (nu se "invită" reciproc).

Semafoarele sunt des utilizate în mecanismele de sincronizare. Un semafor este o variabilă întreagă care poate lua numai valori pozitive. Pentru acest tip de dată se definesc două operații: incrementarea şi decrementarea valorii (operații indivizibile). Considerând un semafor s (variabilă întreagă), operațiile primitive admise asupra acestuia sunt

1. P(s): procesul aşteaptă până când s > 0, apoi decrementează s cu 1;
2. V(s): incrementează s cu 1.

Aceste operații sunt indivizibile. Operația P poate provoca, în anumite condiții, blocarea unui proces, iar V, relansarea unui proces anterior blocat. Semaforului îi este asociat un şir de aşteptare. Procesul blocat se introduce în şirul de aşteptare unde stă până la relansarea lui printr-o operație V. Dacă mai multe procese intenționează să execute simultan operații P sau V asupra aceluiaşi semafor, ele vor fi satisfăcute câte una într-o ordine arbitrară.

În scopul excluderii mutuale, secțiunile critice vor fi incluse între operații P şi V asupra aceluiaşi semafor s, inițializat cu valoarea 1. Primul proces care execută P(s) va intra în secțiunea critică, semaforul obținând valoarea zero. Astfel, orice tentativă din partea altui proces de a executa o secțiune critică se va solda cu punerea în aşteptare prin primitiva P. După execuția de către primul proces a operației V se admite accesul unui nou proces la resursa critică. Se exclud între ele secțiunile critice corespunzătoare aceleiaşi resurse, dar nu există nici un motiv pentru interzicerea accesului concomitent din mai multe procese la resurse critice diferite, caz în care se realizează o excludere mutuală selectivă.

Sincronizarea pe condiție se efectuează astfel: un proces este pus în aşteptare executând o operație P asupra unui semafor şi un alt proces îl relansează executând V asupra aceluiaşi semafor după ce a constatat îndeplinirea unei condiții.

Secțiunile critice sunt destinate excluderii mutuale. O declarație de forma

var v: shared i

introduce o variabilă v de tip i, indicând faptul că ea reprezintă o resursă comună partajată de mai multe procese. Accesul la această variabilă este permis doar în interiorul unei regiuni critice de forma *region* v *do* $instr_1$; ... ; $instr_n$ end. Excluderea mutuală este asigurată prin faptul că la un moment dat un singur proces poate executa instrucțiuni corespunzătoare unei regiuni critice referitoare la o variabilă v.

Execuția unei secțiuni critice se desfăşoară după cum urmează. Dacă nici un alt proces nu se află într-o regiune critică referitoare la aceeaşi variabilă v, procesul continuă (execută instrucțiunile); dacă există un proces în interiorul unei astfel de regiuni critice, noul proces va fi introdus într-un şir de aşteptare asociat resursei critice

respective. In momentul în care un proces părăseşte o regiune critică, se activează unul din procesele care aşteaptă în şirul asociat variabilei.

Pentru sincronizarea pe condiţie s-au introdus regiunile critice condiţionale care presupun punerea în aşteptare a unui proces până la îndeplinirea unei condiţii exprimate printr-o expresie logică.

**Monitorul** este asemenea unei "doici" a cărui permisiune sau ajutor trebuie cerut înaintea oricărei comunicări dintre două procese. Dacă semafoarele sunt primitive de sincronizare ce pot fi calificate ca fiind de nivel scăzut (şi dificil de utilizat), monitoarele oferă o sincronizare de nivel înalt. Un monitor este un tip abstract de dată care consistă dintr-un set permanent de variabile ce reprezintă resursa critică, un set de proceduri ce reprezintă operaţii asupra variabilelor şi un corp (secvenţă de instrucţiuni). Corpul este apelat la lansarea programului şi produce valori iniţiale pentru variabilele-monitor. Apoi monitorul este accesat numai prin procedurile sale. Accesul la aceste proceduri este permis numai procesoarelor selectate. Funcţia monitorului este îndeplinită în condiţiile în care codul de iniţializare este executat înaintea oricărui conflict asupra datelor şi numai una dintre procedurile monitorului poate fi executată la un moment dat. Monitorul creează o coadă de aşteptare a proceselor care fac referire la anumite variabile comune (astfel încât primul sosit la coadă este primul servit atunci când "uşa" este deschisă).

Excluderea mutuală este realizată prin faptul că la un moment dat poate fi executată doar o singură procedură a monitorului. Sincronizarea pe condiţie se realizează prin mijloace mânuite explicit de către programator prin variabile de tip condiţie şi două operaţii *signal* şi *wait*. Dacă un proces care a apelat o procedură de monitor găseşte condiţia falsă, execută operaţia *wait* (punere în aşteptare a procesului într-un şir asociat condiţiei şi eliberarea monitorului). În cazul în care alt proces care execută o procedură a aceluiaşi monitor găseşte condiţia adevărată, execută o operaţie *signal* (procesul continuă dacă şirul de aşteptare este vid, altfel este pus în aşteptare special pentru procesele care au pierdut controlul monitorului prin *signal* şi se va executa un alt proces extras din şirul de aşteptare al condiţiei).

Monitoarele au fost utilizate într-o primă etapă ca bază a limbajelor concurente.
O aplicaţie într-un limbaj concurent spre deosebire de una într-un limbaj de programare paralelă presupune ca procese multiple să împartă acelaşi spaţiu adresabil. Intr-o serie de limbaje a fost incorporat conceptul de monitor pentru protejarea utilizării variabilelor globale (Pascal concurent, Pascal plus). Un program în Pascal Concurent are două tipuri de componente esenţiale: procese şi monitoare. Un proces constă dintr-un număr de variabile locale şi operaţiile asupra lor (instrucţiuni de program), iar monitoarele permit comunicările dintre procese. Sincronizarea este realizată via o variabilă globală de tip coadă, asupra căreia sunt definite trei operaţii:

întârziere (*delay*), continuare (*continue*) şi golire (*empty*).

Etapa următoare în elaborarea limbajelor paralele a constat în alcătuirea unor primitive pentru calculul paralel pentru orice maşină cu memorie comună. Pentru acest proiect s-au considerat două limbaje adecvate: C şi Fortran. Paralelismul este reprezentat la nivel de limbaj prin monitoare. Implementarea monitoarelor este realizată printr-un set de macrouri care permit portabilitate, pentru crearea de procese noi, declararea variabilelor monitor, iniţializarea monitoarelor, intrarea şi ieşirea din monitoare, întârzierea şi execuţia proceselor din coadă.

Expresiile de drum sunt formate din numele unor operaţii reprezentate printr-un set de proceduri şi o serie de operatori care definesc secvenţele permise de executare a procedurilor. Se consideră următoarele exemple:

1. path x; y; z end; - operatorul de secvenţă ';' indică faptul că fiecare execuţie a operaţiei y trebuie să fie precedată de o execuţie încheiată a lui x (analog pentru z). Dacă, de exemplu, prima cerere de prelucrare a resursei vizează operaţia y, procesul apelant va fi pus în aşteptare până când un alt proces apelează şi termină operaţia x. O nouă operaţie x poate fi executată după încheierea lui z;

2. path x, y, z end; - operatorul de concurenţă ',' indică execuţia concurentă a operaţiilor x, y, z fără nici o restricţie în ceea ce priveşte ordinea de executare şi numărul de activări;

3. path x; (y+z); u end; - operatorul de selecţie '+' indică faptul că execuţia unei operaţii x poate fi succedată de o execuţie a lui y sau o execuţie a lui z, urmată de operaţia u.

Sincronizarea este asigurată prin evaluarea în mod automat a stării expresiilor de drum, înainte de acceptarea oricărei cereri din partea unui proces, ocazie cu care se ia decizia cu privire la activitatea operaţiei cerute sau punerea în aşteptare a procesului.

**Sincronizarea în sistemele paralele cu transmitere de mesaje**

Intr-un sistem paralel cu transmitere de mesaje, comunicarea şi sincronizarea sunt combinate într-un singur mecanism. Expedierea unui mesaj este asincronă: procesul emitent trimite mesajul şi continuă execuţia fără să aştepte recepţionarea mesajului. Responsabilitatea delivrării mesajului revine sistemului de operare. Comportarea asincronă este simulată prin introducerea unor procese-tampon (routing process) care acceptă mesajul de la sursă şi îl trimite la destinaţie. In majoritatea sistemelor de acest tip recepţionarea mesajelor este sincronă, adică procesul receptor este blocat până când mesajul dorit este disponibil. Mesajele asincrone pot să nu fie recepţionate în ordinea transmiterii. Există posibilitatea de priorizare a mesajelor importante şi receptorul ia cunoştinţă explicit de expedierea unui anume mesaj.

Sincronizarea se realizează prin programarea corectă a instrucţiunilor de transmitere şi recepţionare a mesajelor.

În anumite sisteme expedierea este şi ea sincronă, procesul emitent fiind blocat până când procesul receptor este pregătit să primească mesajul. Astfel, într-un algoritm sincronizat procesul ce ajunge la un punct de interacţiune este blocat până când procesul cu care urmează să comunice ajunge şi el la punctul de interacţiune. Odată ce mesajul a fost schimbat ambele proces continuă execuţia. Mesajele sincrone necesită ca ambele părţi să fie de acord să comunice, pe când mesajele asincrone permit unui mesaj să fie transmis şi apoi să "atârne" în sistem până când este recepţionat.

In transmiterea unui mesaj se impun trei condiţii:

1. specificarea mesajului emis, respectiv recepţionat (o expresie a cărei valoare reprezintă informaţia de transmis, la emitere, respectiv o listă a identificatorilor variabilelor care obţin valori din conţinutul mesajului, la recepţionare);

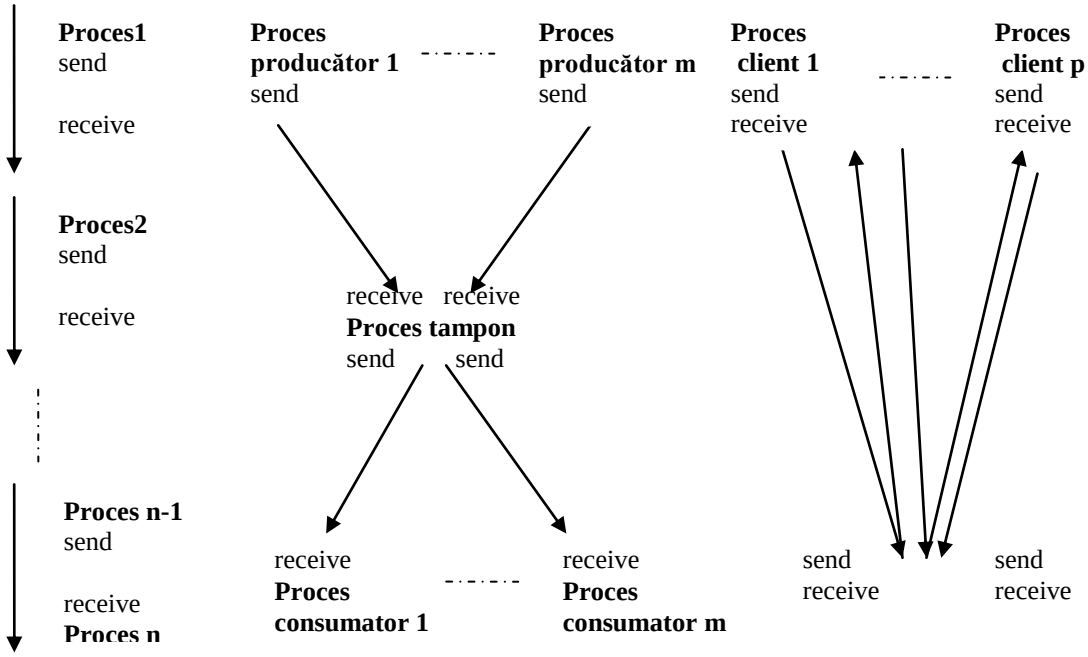
2. specificarea destinaţiei, respectiv sursei:

   (a) prin numirea directă a partenerului de comunicaţie (în special în modelul pipeline, al conductei - figura de mai jos, (a), când fiecare proces cunoaşte în permanenţă identitatea procesului de la care obţine, respectiv căruia îi transmite mesaje;

(b) prin numire globală, prin intermediul unui port de comunicare care joacă rolul unei interfețe (de exemplu în cazul modelului de interacțiune producător-consumator) – figura de mai jos, (b). Numele portului joacă rolul unei cutii poştale în care se depun şi din care se extrag mesaje. Porturile nu sunt asociate unui anumit proces, ci odată declarate sunt vizibile în mod egal din toate procesele. Numirea globală permite comunicarea fără explicitarea identității proceselor comunicante;

(c) prin numire selectivă, prin intermediul unui port de comunicare ataşat numai anumitor procese (cazul modelului de interacțiune în relația client-servant) – figura de mai jos, (c). Procesul posesor al portului nu specifică procesele cu care intră în comunicare, acestea din urmă fiind însă obligate să cunoască identitatea procesului care deține portul;

3. specificarea gradului de sincronizare între procesul emițător şi cel receptor.

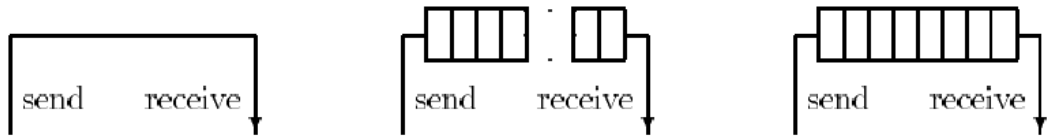



(a) (b) (c)

Modele de interacțiune a proceselor: (a) modelul conductei
(b) modelul producător-consumator (c) modelul client-servant

Fie două procese Pl şi P2 între care se transmit mesaje în sensul Pl → P2. În principiu, procesul Pl poate încheia operația *send,* continându-şi apoi execuția numai dacă au fost asigurate condițiile pentru recepționarea la destinație a mesajului transmis sau pentru memorarea acestuia în vederea recepționării ulterioare. Procesul P2 încheie execuția unei *receive* şi continuă rularea numai dacă există un mesaj disponibil şi acesta a fost recepționat. Pentru analiza gradului de sincronizare se porneşte de la condițiile generale de desfăşurare corectă a comunicării între procese, ilustrate în figura de mai jos:

1. transmitere sincronă (a)
2. transmitere asincronă (b)
3. formă intermediară între primele două (c).




(a) (b) (c)

În cazul în care implementarea operațiilor este de aşa natură încât nu se prevede posibilitatea de memorare a mesajelor (memorie tampon) pe calea dintre cele două procese, Pl va fi blocat în momentul executării operației *send* până când P2 execută o operație *receive*. Această formă presupune sincronizarea proceselor în vederea executării simultane a operațiilor *send* şi *receive*, adică are loc o transmitere sincronă.

Pentru a evita blocarea proceselor care execută *send,* implementarea operației trebuie să prevadă memorarea mesajului tampon în cazul în care P2 nu este gata de recepție (nu a executat *receive*). Aceasta impune existența unei memorii-tampon în care mesajul se depune la *send* şi din care este extras prin *receive.* În cazul în care zona tampon este nelimitată, este garantată în orice moment posibilitatea depunerii unui mesaj. Prin urmare, operația *send* poate fi executată neimpunând nici o aşteptare asupra lui Pl (transmiterea fiind numită asincronă).

In cazul în care memoria tampon este limitată, procesul emițător va aştepta la execuția unui *send* numai atunci când tamponul este plin şi mesajul transmis nu poate fi depus. Procesul blocat continuă după ce s-a creat un loc în tampon ca urmare a unui *receive.*

Blocarea la transmiterea sincronă poate fi evitată prin comunicarea selectivă prin instrucțiuni cu gardă de forma

*if* $g_1 \rightarrow i_1,\ g_2 \rightarrow i_2, \ldots,\ g_n \rightarrow i_n$ end;

*loop* $g_1 \rightarrow i_1,\ g_2 \rightarrow i_2, \ldots,\ g_n \rightarrow i_n$ end;

O gardă constă dintr-o expresie booleană urmată opțional de o operație *send* sau *receive*. La un moment dat garda este deschisă dacă expresia este adevărată şi executarea operației nu impune blocarea procesului. Garda este închisă dacă expresia este falsă. O expresie adevărată şi blocarea printr-un *send* sau *receive* pune garda într-o stare nedefinită.

Instrucțiunea alternativă se execută astfel: dacă există gărzi deschise, se alege la întâmplare una dintre ele şi se execută operația *send* sau *receive* prevăzută, după care se trece la secvența de instrucțiuni corespunzătoare. Dacă toate gărzile sunt închise, instrucțiunea se încheie, cazul fiind semnalat ca eroare. Dacă gărzile sunt în starea nedefinită, procesul aşteaptă până când una din gărzi se deschide (se execută o operație *send* sau *receive* dintr-o gardă în care expresia logică este adevărată).

Instrucțiunea repetitivă are o execuție asemănătoare cu cea alternativă, reluându-se până când toate gărzile sunt închise (instrucțiunea încheindu-se normal, fără semnalarea unei erori).

Blocarea unui proces care execută o instrucțiune cu gardă are loc dacă nici una din operațiile *send* sau *receive* nu este executabilă şi se deblochează în momentul în care oricare dintre ele se poate efectua.

Forma intermediară de transmitere cu tampon limitat se poate realiza şi prin program prin introducerea unui proces tampon. Producătorii se blochează numai dacă

tamponul este plin (se depăşeşte numărul maxim de mesaje recepționate şi care nu au fost re-emise), iar consumatorii se blochează dacă tamponul este gol.

## EFICIENȚA

O problemă importantă în introducerea calculatoarelor paralele este eficiența calculului paralel relativ la calculul secvențial.

Scopul procesării paralele depinde de problema cere se rezolvă. Astfel,

• scopul poate fi maximizarea numărului de sarcini independente realizate în paralel într-o secțiune de calcul cu scop general sau interactivă;

• scopul poate fi maximizarea numărului de procese paralele care cooperează şi minimizarea timpului de răspuns.

Teoretic, un program care lucrează cu $p$ procesoare este executat de $p$ ori mai rapid decât programul similar care lucrează cu un singur procesor. Din păcate, în practică, viteza este mult mai mică. O primă sursă este dificultatea de a diviza un program în unități executabile în aceeaşi perioadă de timp.

Resursele de măsurare a performanței unui algoritm secvențial sunt timpul şi spațiul de memorie. Pentru evaluarea performanței unui algoritm paralel, timpul este resursa majoră. Timpul în calculul paralel nu depinde numai de complexitatea operațiilor, ci şi de complexitatea operațiilor de tip comunicare, sincronizare, limitele schimbului de date. Viteza calculatoarelor paralele nu este derivată din puterea procesoarelor, ci din numărul lor.

Pentru un algoritm paralel se definesc **două tipuri de optimalitate**:

• un algoritm paralel pentru rezolvarea unei probleme date este optim din punct de vedere al timpului (**optimal în sens tare**) dacă se poate demonstra că timpul de execuție nu poate fi îmbunătățit de alt algoritm paralel cu aceeaşi complexitate de calcul, adică nu poate fi îmbunătățit fără schimbarea numărului de operații;

• un algoritm paralel pentru rezolvarea unei probleme date este optim din punct de vedere computațional (**optimal în sens slab**) dacă numărul total al operațiilor utilizate este asimptotic egal cu numărul de operații a celui mai rapid algoritm secvențial pentru problema dată.

**Timpul de calculul paralel** este perioada care s-a scurs de la inițierea primului proces paralel şi momentul când toate procesele paralele au fost terminate.

Într-un sistem cu memorie comună, timpul de calcul paralel a unui algoritm sincron se poate exprima ca sumă a mai multor valori:

1. timpul de procesare de bază, care este suma perioadelor de timp necesare pentru fiecare etapă (între două puncte de interacțiune);

2. timpul de blocare, care este perioada de timp în care procesul aşteaptă date de intrare într-un program sincron, sau intrarea într-o secțiune critică, într-un program asincron.

3. timpul de sincronizare a comunicărilor necesar pentru anumite operații asupra unor variabile comune cu alte procesoare sau necesar execuției secțiunilor critice.

Într-un sistem cu transmitere de mesaje, dacă timpul de execuție a unei operații aritmetice este mult mai mare decât timpul de transfer a datelor între două elemente de procesare, atunci întârzierea datorată rețelei este nesemnificativă, dar dacă este comparabil cu timpul de transfer, atunci timpul de transfer joacă un rol important în determinarea performanței programului. În sistemele actuale bazate pe transputere,

raportul dintre timpul necesar unei operații de comunicare şi timpul necesar unei operații aritmetice este de ordinul 500 - 1000.

Algoritmii pentru calculul paralel sunt diferiți de cei pentru calculul serial şi pentru măsurarea performanței reale este necesară recodificarea algoritmilor seriali existenți.

Problemele care se discută, relativ la eficiență, sunt:

1. viteza algoritmului paralel,

2. pierderea de eficiență per procesor unitate la execuția algoritmului pe un sistem paralel.

Corespunzător, există două măsuri ale performanței algoritmilor paraleli: **viteza** şi **eficiența**.

Fie P o problemă dată şi *n* dimensiunea datelor de intrare. Notăm cu $T_0(n)$ complexitatea secvențială a lui P, ceea ce presupune că există un algoritm care rezolvă P în acest timp şi, în plus, se poate demonstra că oricare algoritm secvențial nu rezolvă P mai repede. Fie A un algoritm paralel care rezolvă problema P în timpul $Tp(n)$ pe un calculator paralel cu *p* procesoare. Atunci viteza atinsă de A este definită prin

$$S_p(n) = \frac{T_0(n)}{T_p(n)}$$

Deoarece $S_p(n) \le p$, dorim să realizăm algoritmi care ating $S_p(n) \approx p$. În realitate, există o serie de factori care reduc viteza. Aceştia sunt, de exemplu, concurența insuficientă în calcul, întârzierile introduse de comunicare, suplimentele de timp datorate sincronizării activităților unor procesoare variate şi în controlul sistemului. În practică, curbele număr procesoare-viteză sunt apropiate de $p/\log_2 p$, maxim $0.3p$. Dacă viteza urmează curba $p/\log_2 p$, care are limita la $p \to \infty$ egală cu zero, se observă că anumiți algoritmi nu sunt eficienți pentru sisteme cu număr mare de procesoare.

De notat este faptul că $T_1(n)$, timpul de execuție a algoritmului paralel A când numărul de procesoare este egal cu unu, nu este necesar a fi identic cu $T_0(n)$. Viteza este măsurată relativ la cel mai bun algoritm secvențial. Este o practică comună aceea de a înlocui $T_0(n)$ cu timpul celui mai bun algoritm secvențial cunoscut, dacă complexitatea problemei nu este cunoscută.

Timpul de execuție a unui algoritm secvențial este estimat prin numărul de operații de bază cerute de algoritm ca funcție de dimensiunea datelor de intrare. În mod curent se atribuie o unitate de timp pentru operațiile de citire şi scriere în memorie şi pentru operațiile aritmetice şi logice (ca adunarea, scăderea, compararea, multiplicarea, sau logic etc). Costul unei asemenea operații nu depinde de lungimea cuvântului; se utilizează ceea ce se numeşte **criteriul costului uniform**.

O altă măsură a performanței unui algoritm paralel este eficiența, definită prin

$$E_p(n) = \frac{T_1(p)}{pT_p(n)}$$

Această măsură oferă o indicație a utilizării efective a celor *p* procesoare în implementarea algoritmului dat. O valoare a lui $Ep(n)$ aproximativ egală cu 1, pentru anumit *p*, indică faptul că algoritmul A rulează aproximativ de *p* ori mai repede utilizând *p* procesoare decât utilizând un procesor.

Există o limită superioară asupra timpului de rulare, notată cu $T_\infty(n)$, sub care algoritmul nu poate rula mai repede, indiferent de numărul de procesoare, adică

$T_p(n) \geq T_\infty(n)$, pentru orice valoare a lui *p*, astfel încât eficiența *Ep(n)* satisface $E_p(n) \leq T_1(n)/pT_\infty(n)$. De obicei eficiența unui algoritm descreşte rapid cu creşterea lui *p*.

Pe lângă definițiile de mai sus se întâlnesc şi o serie de alte variante de definire a vitezei şi eficienței:

1. viteza implementării pe un sistem cu p procesoare a unui algoritm paralel

$$S'_p(n) = T_1(n)/T_p(n)$$

2. viteza algoritmului paralel asociat unui algoritm serial:

$$S''_p = [(1-\alpha)+\alpha/p]^{-1}$$

unde $\alpha$ este procentul dintr-un algoritm serial care poate fi procesat în paralel, dacă timpul de execuție cu ajutorul unui singur procesor este normalizat la unitate. Dacă se ține seama de factorii perturbatorii în comunicarea datelor

$S''_p = [(1-\alpha)+\alpha/p+\sigma(p)]^{-1}$, unde $\sigma(p)$ reflectă influența transferului de date asupra timpului de execuție;

3. eficiența numerică a unui algoritm paralel: $E_{num}(n) = T_0(n)/T_1(n)$;

4. eficiența unei implementări a algoritmului paralel față de algoritmul serial optim: $E'_p(n) = T_0(n)/[pT_p(n)]$.

Majoritatea algoritmilor sunt proiectați în condițiile unei maşini paralele abstracte, care are un număr de procesoare suficient pentru a asigura realizarea într-un singur pas paralel a unei operații cu *n* date. În practică, numărul *n* de procesoare cerut nu este întotdeauna disponibil. De exemplu, n procesarea vectorială este acceptată doar o anumită lungime maximă a vectorului de date.

**Exemple**. Timpul necesar pentru efectuarea unei aceleiaşi operații pe *n* date, raportat la timpul necesar pentru o singură operație (notat cu *t*) depinde de caracteristicile maşinii. Pentru un procesor serial $T_0(n) = nt_s$, iar pentru un procesarea în paralel pe un sistem cu transmitere de mesaje $T_p(n) \leq [n/p]t_p + \Delta$, unde *p* este numărul de elemente de procesare disponibile, $t_p$ este timpul necesar pentru o operație (aritmetică) în paralel, [x] este numărul întreg cel mai mic care satisface $[x] \geq x$, iar $\Delta$ reprezintă perturbația temporală datorată comunicărilor şi sincronizărilor dintre procesoare. Pentru procesarea vectorială, $T_p(n) = [n/p]t_v + pt_v$, unde $pt_v$ este timpul de start pentru o operație vectorială, iar *p* este lungimea maximă a unui vector care poate fi procesat. Vectorii-date sunt partiționați în subvectori de maxim *p* componente.

Performanța calculului paralel este funcție de mărimea problemei. Dacă problema nu este aleasă conform configurației de interconectare a sistemului, nu se măsoară performanța reală.

Performanța unei arhitecturi paralele este măsurată prin indicele de performanță, care este o funcție reală definită pe spațiul parametrilor arhitecturii. De obicei, indicele de performanță este exprimat prin rata de utilizare a unui procesor în timp, *U*. De exemplu, dacă se consideră cazul evaluării unei funcții iterative prin metoda descompunerii în subprograme de complexitate egală, astfel încât procesoarele individuale execută setul de instrucțiuni asociat în aceeaşi perioadă de timp, atunci

$$U = t_0/t_1$$

unde $t_1$ este timpul total necesar unei iterații, iar $t_0$ este timpul mediu în care un procesor este angajat în executarea setului de instrucțiuni. Astfel, *U* reprezintă fracția de timp în care procesorul este ocupat, iar *1 - U*, fracția de timp în care procesorul nu este utilizat. Indicele de performanță exprimă efectul arhitecturii calculatorului paralel asupra algoritmului iterativ.

Problemele cunoscute astăzi ca având soluții paralele eficiente constituie clasa de probleme notată NC (Nick's class). Clasa NC conține problemele rezolvabile într-un timp polilogaritmic cu un număr polinomial de procesoare.

Se utilizează în capitolele următoare notația *T(n) = O(f(n))* pentru cazul în care există constantele pozitive c şi $n_0$ astfel încât $T(n) \le cf(n), \forall n \ge n_0$.

## ORGANIZAREA DATELOR

Se consideră cazul unei probleme care necesită un număr de procesoare care nu este disponibil în sistem.

Există două tehnici de acomodare a unei probleme mari la un număr mic de procesoare:

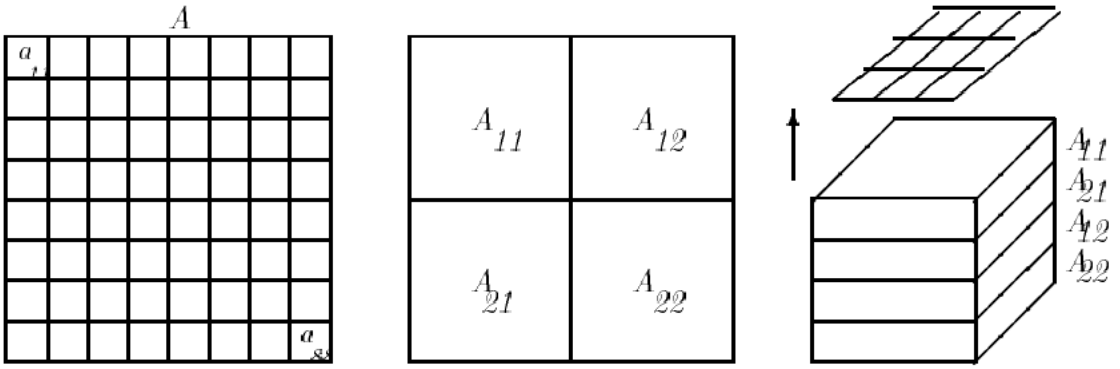
1. tehnica tăierii ("slicing")
2. tehnica încrețirii ("crinkling").

O altă problemă importantă în cazul unor probleme de dimensiuni mari este alegerea optimă a distribuției datelor pe procesoare cu realocare dinamică. Metoda consacrată rezolvării acestei probleme este numită **transferul datelor în paralel** (PDT).

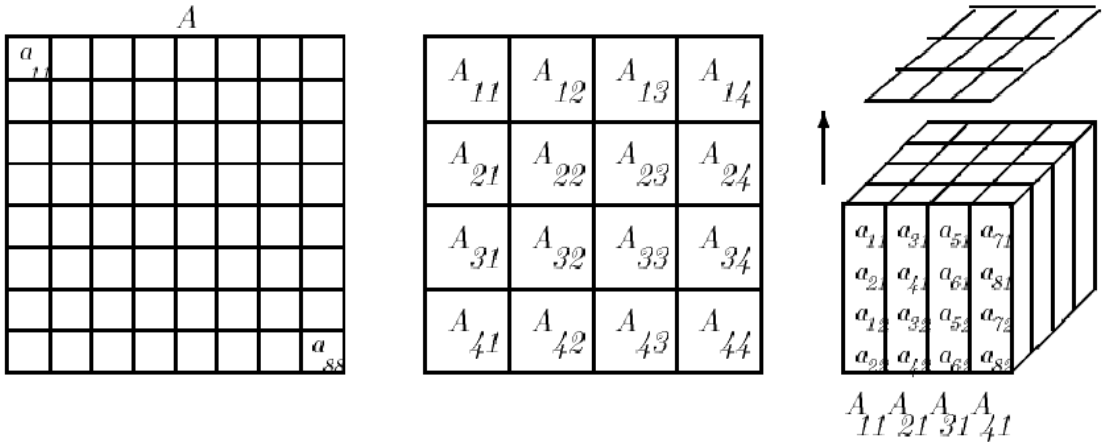
## TEHNICI DE DISTRIBUIRE A DATELOR

Utilizând tehnica tăierii, setul de date este partiționat în unități mai mici, astfel încât fiecare unitate în parte poate fi procesată individual de sistem. În tehnica încrețirii, se menține conectarea dintre elementele vecine ale setului de date.

**Exemplu**. Se caută rezolvarea unei probleme pe o grilă de dimensiune 8 x 8 cu un sistem matricial de procesoare de dimensiune 4 x 4 cu conectare între vecinii direcți. Tehnica de tăiere presupune împărțirea problemei în patru unități, care fiecare poate fi prelucrată în paralel, ca în figura următoare.




În tehnica de încrețire se împarte grila în mulțimi de blocuri de dimensiune 2 x 2, fiecare fiind asignat la un procesor , ca în figura de mai jos.



Alegerea unei tehnici sau a alteia se face în funcție de problema care se rezolvă. De exemplu, se cere calculul matricii de elemente

$$b_{ij} = \frac{1}{4}(a_{ij+1} + a_{i-1j} + a_{ij-1} + a_{i+1j}), i, j = 2,...,7$$

Calculul este realizabil în patru paşi, dacă se utilizează tehnica încrețirii, respectiv 12 paşi, dacă se utilizează tehnica de tăiere.

În anumite probleme pot fi utilizate succesiv ambele tehnici.

## TEHNICA DE TRANSFER A DATELOR

Metoda de transfer a datelor în paralel, PDT, este o metodă de organizare a datelor, utilă în reconfigurare a logică. Se pune problema determinării configurației logice optime pentru o anumită problemă şi o rețea fizică dată.

**Exemplul 1**. Se consideră un sistem cu *n* procesoare în conectare liniară, fiecare având o memorie care poate fi ocupată de o singură dată. In mod curent, se stabileşte o funcție între date şi procesoare. Alocarea a *n* date se poate face în n! moduri. Fie cazul *n*= 8. Se presupune că, inițial, data de index *i* este alocată procesorului de index *i*, unde *i*= 0,...,7. O realocare a datelor este exprimabilă printr-o permutare. De exemplu, prin amestecarea perfectă a datelor, se obține

$$\begin{pmatrix} 0 1 2 3 4 5 6 7 \\ 0 4 1 5 2 6 3 7 \end{pmatrix}$$

Prima linie reprezintă procesoarele, a doua indexul datelor. Simplificat, permutarea se poate scrie [0 4 1 5 2 6 3 7]. În sistem binar indexul unui procesor sau a unei date se reprezintă pe trei biți: $b_2b_1b_0$. Asociem permutării identice vectorul (2 1 0) corespunzător ordinii normale a biților ce reprezintă indexul. Vectorul (1 0 2) este asociat aranjării biților $b_1b_0b_2$. Dacă această schimbare de vector este aplicată fiecărui index de procesor corespunzător permutării identice se obține permutarea corespunzătoare amestecării perfecte [0 4 1 5 2 6 3 7]. De exemplu, data de index 4, aflată inițial în procesorul 4 - index care se scrie în binar 100 - prin aplicarea vectorului de permutare a biților indexului de procesor, se va afla în memoria procesorului 001, adică de index 1. Astfel, rearanjarea datelor constă într-o schimbare a unui vector, mai eficientă decât transferul fizic. Alte exemple: pentru (0 1 2), se obține [0 4 2 6 1 5 3 7], pentru (0 2 1), [0 2 4 6 1 3 5 7].

**Exemplul 2.** Metoda PDT se extinde la cazul datelor multiple pe un singur procesor, date multidimensionale şi matrice de procesoare conectate prin diferite tipuri de rețele.

Considerăm cazul a 16 date care sunt accesate de patru procesoare conectate liniar. Prin tehnica de tăiere datele sunt împărțite în patru grupuri şi sunt accesate ca elemente ale unui vector:

| Adresa/Procesor | **0** | **1** | **2** | **3** |
|---|---|---|---|---|
| **0** | 0 | 1 | 2 | 3 |
| **1** | 4 | 5 | 6 | 7 |
| **2** | 8 | 9 | 10 | 11 |
| **3** | 12 | 13 | 14 | 15 |

Analog exemplului anterior, funcțiile de asociere fizică sunt [0 1 2 3], [4 5 6 7], [8 9 10 11], [12 13 14 15]. Datele de index 0, 4, 8 şi 12 sunt conținute în memoria procesorului 0. Prin tehnica încrețirii, repartizarea se face astfel:

| Adresa/Procesor | **0** | **1** | **2** | **3** |
|---|---|---|---|---|
| **0** | 0 | 4 | 8 | 12 |
| **1** | 1 | 5 | 9 | 13 |
| **2** | 2 | 6 | 10 | 14 |
| **3** | 3 | 7 | 11 | 15 |

Datele de index 0, 1, 2, 3 sunt memorate la procesorul 0.

Poziția unei date poate fi caracterizată prin numărul de procesor şi adresa în memoria acestuia. Se asociază astfel fiecărei date indexul $b_3b_2b_1b_0$, unde

• în tehnica de tăiere, $b_3b_2$ reprezintă adresa, iar $b_1b_0$ reprezintă procesorul. De exemplu, data 13 (1101 în binar) este memorată la adresa 3 (11) a procesorului 1 (01);

• în tehnica încrețirii, $b_3b_2$ reprezintă procesorul, iar $b_1b_0$ adresa. De exemplu, 13 este memorat la adresa 1 a procesorului 3.

## CÂND O PROBLEMĂ ESTE PARALELIZABILĂ?

Multe probleme par a fi mai puțin adaptabile la procesarea paralelă, în pofida prezenței unui număr mare de procesoare.

Problemele pot fi clasificate relativ la noțiunea de *paralelizabilitate*. Aceasta tratează problema prin intermediul a doi parametrii: timpul şi numărul de procesoare. Astfel,

• o problemă poate fi definită ca fiind paralelizabilă dacă poate fi rezolvată cu atât mai rapid cu cât numărul de procesoare este crescut;

• o problemă este paralelizabilă dacă poate fi rezolvată într-un timp polilogaritmic (adică $O(log^k n)$, unde $n$ este dimensiunea problemei, iar $k$ un număr natural) cu un număr polinomial de procesoare (problema aparține clasei NC).

Gradul de paralelism al unui algoritm se defineşte astfel:

1. teoretic este numărul de operații aritmetice ce sunt independente şi pot fi executate concurent;

2. în implementarea pe un procesor pipeline pentru care operanzii sunt definiți ca vectori, este lungimea liniei de procesare;

3. în implementarea pe un masiv de procesoare, este egal cu numărul de operanzi prelucrați în paralel.

## GENERAREA ALGORITMILOR PARALELI

Când se implementează o problemă pe un calculator paralel, prima sarcină este aceea de a descompune procesul, în aşa manieră, încât procesoarele să lucreze concurent la o problemă. Comunicarea dintre elementele de procesoare poate produce probleme deosebite. In principiu, se urmăreşte sincronizarea proceselor şi minimizarea numărului de comunicaţii al căror timp este limitat fizic.

În principiu, există trei modalităţi de generare a algoritmilor paraleli:

1. **prin unităţi independente**: fiecare procesor execută acelaşi program izolat de celelalte procesoare;

2. **prin paralelism geometric**: fiecare procesor execută acelaşi program asupra unor date corespunzătoare unei sub regiuni a sistemului; datele de pe frontiera subregiunii sunt date de comunicare cu procesoarele învecinate care operează asupra sub regiunilor limitrofe;

3. **prin paralelism în algoritm**, în care fiecare procesor este responsabil pentru o parte din algoritm şi toate datele trec prin fiecare procesor.

In problemele de simulare apare adesea necesară execuţia unui singur program pentru o mulţime de date. Cel mai simplu mod de exploatare a paralelismului este executarea mai multor copii ale unui program serial pe un număr de procesoare independente, fiecare copie operând asupra unui set diferit de date de intrare. Dacă natura problemei este de aşa manieră încât fiecare simulare presupune aceeaşi perioadă de timp, atunci implementarea este "balansat încărcată", iar viteza depinde liniar de numărul de procesoare. Eficienţa calculului paralel este mică dacă implementarea nu este balansat încărcată.

O problemă este geometric paralelizabilă dacă algoritmul corespunzător implică numai operaţii asupra unor date care pot fi grupate local. Fie exemplul simulării comprimării unui lichid. Regiunea din spaţiu ocupată de lichid poate fi divizată în subregiuni egale în număr egal cu numărul procesoarelor disponibile. Fiecare procesor este responsabil de evoluţia lichidului în sub regiunea sa. Datele referitoare la frontiera subregiunii trebuie transmise între unităţile de memorie ale elementelor de procesare.

Comunicaţiile într-o maşină bazată pe transputere produc probleme, în cazul unei descompuneri geometrice. Procesoarele care comunică trebuie să fie apropiate pentru o balansare cât mai eficientă între timpul de calcul şi timpul de comunicare. Astfel, reţeaua de interconectare trebuie să fie cât mai apropiată de structura cerută de problemă. Dacă se execută o descompunere geometrică în trei dimensiuni, de exemplu, pe cuburi, fiecare procesor trebuie să comunice cu şase alte procesoare. O transpunere directă a structurii cerute pe sistemele actuale cu transputere cu patru legături, nu este posibilă. Se utilizează tehnici speciale de proiecţie pe structura reală.

Un exemplu de problemă geometric paralelizabilă este "jocul vieţii". Se caută simularea evoluţiei unei colonii de celule din spaţiul bidimensional. Fiecare celulă se află într-unul din stadiile: vie sau moartă. La fiecare pas (generaţie), stadiul unei celule depinde de cei mai apropiaţi opt vecini. O celulă vie supravieţuieşte în generaţia următoare, dacă are 2 sau 3 vecini vii, altfel moare de singurătate sau supra-aglomerare. O celulă moartă înconjurată de exact trei vecini reînvie în generaţia următoare, altfel rămâne moartă.

Modelul trecerii tuturor datelor prin toată reţeaua de procesoare este utilizat pentru probleme care presupun interacţiuni de lungă distanţă, cum ar fi problemele electrostatice sau gravitaţionale. Fie, de exemplu, cazul simulării dinamicii moleculare

ale unui sistem de *n* particule care interacționează pe baza unor forțe electrostatice. În cazul general, este necesară calcularea a *n(n-1)/2* interacțiuni la fiecare pas. Se consideră o structură de p procesoare conectate într-o structură circulară. Se distribuie aleator *n/p* particule la fiecare procesor, care are sarcina de a urmări mişcarea particulelor repartizate. Primul pas efectuat de fiecare procesor constă în alegerea unei particule din memoria locală şi transmiterea masei şi coordonatelor acesteia următorului procesor din inel. In pasul al doilea, fiecare procesor calculează forța de interacțiune a particulelor sale cu particula "călătoare" şi transmite informația (masă şi coordonate) despre această particulă următorului procesor din inel. Procedura se repetă până când fiecare particulă a vizitat fiecare procesor (un pas al procedurii de simulare a dinamicii moleculare). Forțele care acționează asupra unei particule, datorate celorlalte particule, se acumulează, determinând evoluția particulei.

## Capitolul 3

## ALGORITMI PARALELI FUNDAMENTALI

## DIVIDE ET IMPERA

Majoritatea algoritmilor paraleli numerici urmează principiul"divide et impera". Acesta presupune împărțirea problemei în subprobleme mici care pot fi tratate independent. Gradul de independență este o măsură a efectivității algoritmului pentru a determina cantitatea şi frecvența comunicărilor şi sincronizării.

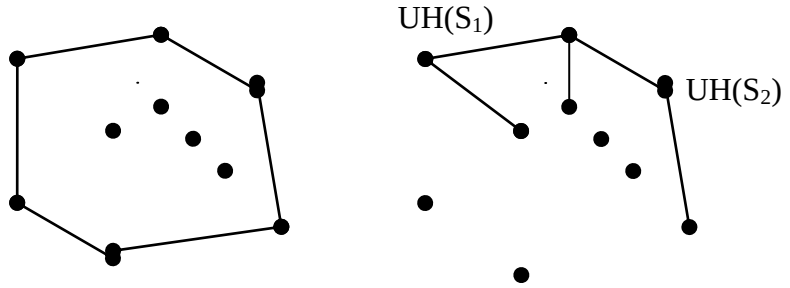
Strategia" divide et impera" consistă în trei paşi principali:

1. partiționarea problemei în subprobleme de dimensiuni aproximativ egale;
2. rezolvarea recursivă a subproblemelor şi concurentă a întregului set de subprobleme;
3. combinarea soluțiilor subproblemelor într-o soluție pentru problema generală.

**Exemplul 1**. Aplicăm conceptul "divide et impera" la calculul sumei $\sum_{i=1}^{n} a_i b_i$.

Produsul $a_i b_i$ este calculat de procesorul $P_i$. Procesoarele cu număr de identificare *i* par transmit rezultatul la procesoarele *i - 1*. Operația de însumare este astfel "divizată" între *n /2* procesoare: procesorul $P_i$ cu *i* impar efectuează suma $a_i b_i + a_{i+1} b_{i+1}$.Procedeul este repetat de $[log_2 n]$ ori până când suma este obținută în procesorul $P_l$.

**Exemplul 2**. Se pune problema închiderii convexe a unei mulțimi de puncte din plan, reprezentate în coordonate carteziene, $S = \{p_1, p_2, ..., p_n\}$.. Închiderea convexă a lui S este cel mai mic poligon convex care conține toate cele *n* puncte ale lui *S*:




Problema constă de fapt în determinarea unei subliste ordonate de puncte din S care să definească poligonul convex de închidere a lui S, notat CH(S).

Fie p şi q punctele lui S cu cea mai mică, respectiv cea mai mare abscisă. În mod natural, $p, q \in CH(S)$. Se notează cu *UH*(S) lista ordonată a punctelor din CH(S) dintre *p* şi *q* parcurse în sensul acelor de ceasornic, iar *LH*(S), de la *q* la *p* în acelaşi sens. Evident $UH(S) \cup LH(S) = CH(S)$, $UH(S) \cap LH(S) = \{p,q\}$. Se determină separat *UH*(S) şi *LH*(S).

Se sortează punctele $p_i$ după abscise. Fie $x(p_1) < x(p_2) < ... < x(p)$ unde $x(p_i)$ este abscisa lui $p_i$. Fie $S_1 = (p_1, ..., p_{\lfloor n/2 \rfloor}), S_2 = (p_{\lfloor n/2 \rfloor + 1}, ..., p_n)$. Presupunând că *UH(S1)* şi *UH(S2)* au fost determinate, *UH*(S) poate fi uşor constituit pe baza determinării tangentei comune care uneşte două puncte din *UH(S1)* respectiv *UH(S2)*.

Procesarea paralelă presupune determinarea concurentă a lui *UH(S1)* şi *UH(S2.)* Numărul de procese concurente se dublează la fiecare apel recursiv.

## MULTIPLICAREA A DOUA MATRICI

Pentru calculul serial al produsului C a două matrici A şi B sunt necesare trei cicluri. Există şase moduri de aranjare a acestor cicluri, iar fiecare diferă în modul în care sunt accesate matricile, pe linie sau coloană, ca scalari, vectori sau matrici. Modul specific de acces afectează performanțele algoritmului pe un calculator paralel cu o structură dată.

Dacă matricile sunt pătratice, de dimensiune $n$, şi se consideră un sistem cu elemente de procesare dispuse în spațiul tridimensional, pe fiecare direcție cel puțin $n$, atunci fiecare produs scalar poate fi evaluat de un singur procesor. Timpul de procesare este de ordinul $O(log_2 n)$ şi necesită $n^3$ procesoare. Un algoritm care necesită doar $n^2$ procesoare şi un timp de procesare de ordin $O(n)$ este cel pentru care fiecare procesor evaluează o componentă a matricii produs.

**Exemplu**. Se consideră două matrici pătratice, A şi B, de dimensiune $n = 2^d$. Se dispune de un sistem cu memorie distribuită de tip cub 3d-dimensional, adică cu $n^3$ procesoare. Se indexează procesoarele prin tripletele $(l,i,j)$ astfel încât $P_{lij}$ reprezintă procesorul $P_r$, unde $r = ln^2 + in + j$ (scriindu-l pe $r$ în binar se obțin $d$ cei mai semnificativi biți corespunzând lui $l$, următorii $d$ biți corespunzând lui $i$, apoi ultimii $d$ cei mai puțin semnificativi corespunzând lui $j$). Matricea A este stocată într-un subcub determinat de procesoarele $P_{li0}$, $0 \le l, i \le n-1$, astfel încât $a_{il}$ se află în $P_{li0}$. In mod similar, B este stocat în subcubul format de procesoarele $P_{l0j}$, unde $P_{l0j}$ reține pe $b_{lj}$. Elementele $c_{ij}$ ale matricii produs sunt determinate în trei etape:

1. datele de intrare sunt distribuite astfel încât $P_{lij}$ va deține valorile $a_{il}$ şi $b_{lj}$, pentru $0 \le l, i, j \le n-1$;

2. procesorul $P_{ijl}$ calculează produsul $c'_{lij} = a_{il}\, b_{lj}$, $0 \le l, i, j \le n-1$;

3. pentru fiecare $0 \le i, j \le n-1$, procesoarele $P_{lij}$, $0 \le l \le n-1$ calculează suma $c_{ij} = \sum_{l=0}^{n-1} c'_{lij}$.

Pentru multiplicarea a două matrici de dimensiuni diferite există o serie de strategii. Fie A şi B două matrici de mărime $m$ x $n$, respectiv $n$ x $p$. Pentru calculul matricei produs C sunt necesare $mnp$ înmulțiri. Fie * operația de înmulțire a două matrici pe componente.

**Cazul 1**. Numarul de procesoare satisface $N \ge mnp$. Atunci toate multiplicările se pot efectua printr-o singură operație *. De exemplu, fie $m = n = p = 2$,

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}\begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{11} \\ a_{21} & a_{21} \end{pmatrix} * \begin{pmatrix} b_{11} & b_{12} \\ b_{11} & b_{12} \end{pmatrix} + \begin{pmatrix} a_{12} & a_{12} \\ a_{22} & a_{22} \end{pmatrix} * \begin{pmatrix} b_{21} & b_{22} \\ b_{21} & b_{22} \end{pmatrix}$$

Valorile $a_{ik}b_{kj}$ sunt elementele matricii rezultat

$$\begin{pmatrix} a_{11} & a_{11}a_{12} & a_{12} \\ a_{21} & a_{21}a_{22} & a_{22} \end{pmatrix}\begin{pmatrix} b_{11} & b_{12}b_{21} & b_{22} \\ b_{11} & b_{12}b_{21} & b_{22} \end{pmatrix}$$

Care poate fi rearanjată astfel

$$\begin{pmatrix} a_{11} & a_{12}a_{11} & a_{12} \\ a_{21} & a_{22}a_{21} & a_{22} \end{pmatrix}\begin{pmatrix} b_{11} & b_{21}b_{12} & b_{22} \\ b_{11} & b_{21}b_{12} & b_{22} \end{pmatrix}$$

Pentru cazul general se consideră

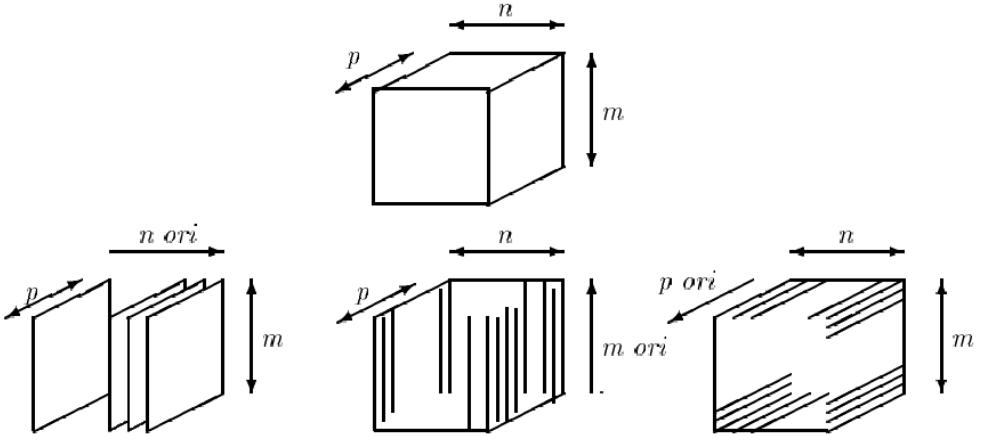
$$(A_{1*}A\cdots A_{*1})\begin{pmatrix} B_{1*}^T & \cdots & B_{p*}^T \\ \vdots & \vdots & \vdots \\ B_{1*}^T & \cdots & B_{p*}^T \end{pmatrix}$$

unde $B_{1*}$ este linia *i* a matricei B. Pentru obținerea matricii C se adună câte *n* numere din matricea de mai sus.

**Cazul 2**. Numărul de procesoare satisface $mnp > N \geq \max(mn, np, mp)$. Atunci este posibil a se efectua *mp*, *mn* sau *np* multiplicări simultan (printr-o singură operație de înmulțire *) în fiecare ciclu care e repetă de *n*, *p* sau *m* ori. Apar 3 situații.

1. min(*m*,*n*,*p*)=*n*. Se utilizează algoritmul produsului extern. Se observă că $C = \sum_{k=1}^{n} C_k$, unde $(C_k)_{ij} = a_{ik}b_{kj}$. Matricea $C_k$ poate fi obținută printr-o singură operație *:
$$C_k = \begin{pmatrix} a_{1k} & \cdots & a_{1k} \\ \vdots & \vdots & \vdots \\ a_{mk} & \cdots & a_{mk} \end{pmatrix} * \begin{pmatrix} b_{k1} & \cdots & b_{kp} \\ \vdots & \vdots & \vdots \\ b_{k1} & \cdots & b_{kp} \end{pmatrix}$$
unde matricele au dimensiunea *m*x*p*.
2. min(*m*,*n*,*p*) = *m*. Se utilizează algoritmul produsului intern pe linii. Linia $C_{i*}$ a lui C (i=1,...,*m*) poate fi obținută prin însumarea elementelor corespunzătoare de pe fiecare linie a matricii
$$\begin{pmatrix} a_{i1} & \cdots & a_{i1} \\ \vdots & \vdots & \vdots \\ a_{in} & \cdots & a_{in} \end{pmatrix} * \begin{pmatrix} b_{11} & \cdots & b_{1p} \\ \vdots & \vdots & \vdots \\ b_{n1} & \cdots & b_{np} \end{pmatrix}$$
unde matricele au dimensiunea *n*x*p*.
3. min(*m*,*n*,*p*) = *p*. Se utilizează algoritmul produsului intern pe coloane. Linia $C_{*j}$ a lui C (j=1,...,*p*) poate fi obținută prin însumarea elementelor corespunzătoare coloanelor matricii
$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \vdots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} * \begin{pmatrix} b_{1j} & \cdots & b_{nj} \\ \vdots & \vdots & \vdots \\ b_{1j} & \cdots & b_{nj} \end{pmatrix}$$
unde matricele au dimensiunea *m*x*n*.

Figura următoare schițează înmulțirea a două matrici pe un sistem de *N*<*mnp* procesoare:



**Cazul 3**. Numărul de procesoare satisface $N < \max(mn, np, mp)$. Matricile trebuie partiționate astfel încât fiecare unitate să fie prelucrată în paralel.

Alegerea algoritmilor de multiplicare descrişi mai sus ține cont de minimizarea numărului de multiplicări. Adeseori şi numărul de adunări care se efectuează poate influența considerabil timpul de calcul. Teoretic nu se ține cont de timpul de acces la operanzi. Acest timp suplimentar în mod frecvent reduce performanțele algoritmilor. În practică, operanzii sunt accesați dintr-o memorie comună sau fiecare procesor deține propria sa memorie şi trebuie ținut seama de timpul de transfer a datelor între procesoare.

Principiul partiționării poate fi utilizat şi în calculul produsului dintre o matrice şi un vector. Fie A o matrice pătratică de dimensiune $n$ şi un vector x cu $n$ componente, ambele stocate într-o memorie comună. Se distribuie calculul între $p$ procese, unde $p$ are proprietatea că $r = n / p$ este un număr întreg.

**Algoritm asincron**. Se partiționează $A = (A_1, A_2, \ldots, A_p)^T$ unde fiecare bloc $A_i$ are dimensiunea $r$ x $n$. Pentru fiecare $1 \le i \le p$, procesul asociat, $P_i$, operează asupra lui $A_i$ şi $x$ din memoria comună, având sarcina de a calcula $Z_i = A_i x$ şi de a stoca cele $r$ componente ale vectorului rezultat $Z_i$ în componentele corespunzătoare ale variabilei comune $y$ care reprezintă vectorul rezultat, $y = Ax$. În acest algoritm, fiecare proces execută acelaşi program, setul de date depinzând de identificatorul de procesor. Se realizează o citire comună a aceleiaşi variabile $x$, dar oricare două procese nu scriu în aceeaşi locație de memorie, ele nefiind în mod necesar sincronizate.

**Algoritm sincron**. Se partiționează $A = (A_1, A_2, \ldots, A_p)$, $x = (x_1, x_2, \ldots, x_p)$, astfel încât $A_i$ are dimensiunea $n$ x $r$, iar $x_i$, $r$ componente. Atunci vectorul rezultat se exprimă sub forma $y = A_1x_1 + A_2x_2 + \ldots + A_px_p$. Produsele $z_i = A_ix_i$ pot fi calculate simultan după citirea datelor $A_i$ şi $X_i$ din memoria comună, pentru fiecare $1 \le i \le p$. Suma finală este calculată numai după ce toate procesele au terminat calculul matriceal. Astfel, o primitivă explicită de sincronizare trebuie plasată în fiecare proces după calculul lui $z_i$ pentru a forța toate procesele să se sincronizeze. Într-un sistem cu transmitere de mesaje, pentru calculul sumei se poate utiliza conectarea liniară între procese.

## EVALUAREA EXPRESIILOR ARITMETICE

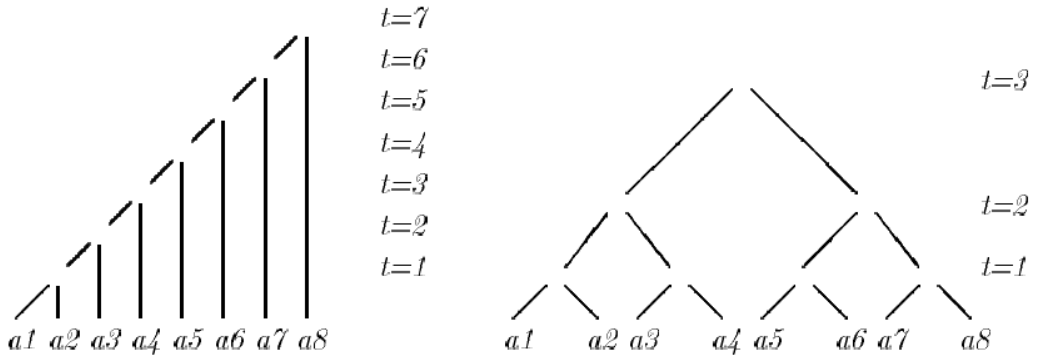
Expresiile aritmetice ocupă un rol central în calcul şi de aceea prezintă un interes deosebit pentru calculul paralel.

Paralelismul se dovedeşte extrem de eficient la evaluarea expresiilor aritmetice. Tehnicile utilizate sunt aplicabile la

- nivelul unei expresii aritmetice;
- nivelul instrucțiunilor de atribuire a unor expresii aritmetice.

## TEHNICA DUBLĂRII RECURSIVE

Se consideră $o$, un operator asociativ, ce se aplică perechilor de obiecte matematice (numere, vectori, matrici şi altele). Fie cazul în care compunerea a $n$ numere este unic definită, independent de ordinea de efectuare a operațiilor (de introducere a parantezelor). Modul optim de evaluare a unei asemenea compuneri depinde de caracteristicile sistemului. Un exemplu concludent este prezentat în figura următoare pentru compunerea a $n = 8$ obiecte notate $a_i$, $i = 1,\ldots,n$.




Calculele la fiecare nivel sunt realizate în paralel. Dacă se compun *n* obiecte, rezultatul este produs în [$\log_2 n$] paşi. Tehnica se numeşte **dublare recursivă.** Volumul de calcul este divizat succesiv în două unităţi de complexitate egală care sunt executate în paralel. Pentru obţinerea eficienţei maxime se recomandă transpunerea pe o reţea de tip arbore binar.

Tehnica dublării recursive este adesea utilizată în practică. Exemple concludente sunt adunarea sau produsul a *n* numere, determinarea maximului şi minimului unei mulţimi de numere. Operaţiile pot fi executate şi asupra vectorilor şi matricilor (de exemplu, în cazul evaluării relaţiilor recursive).

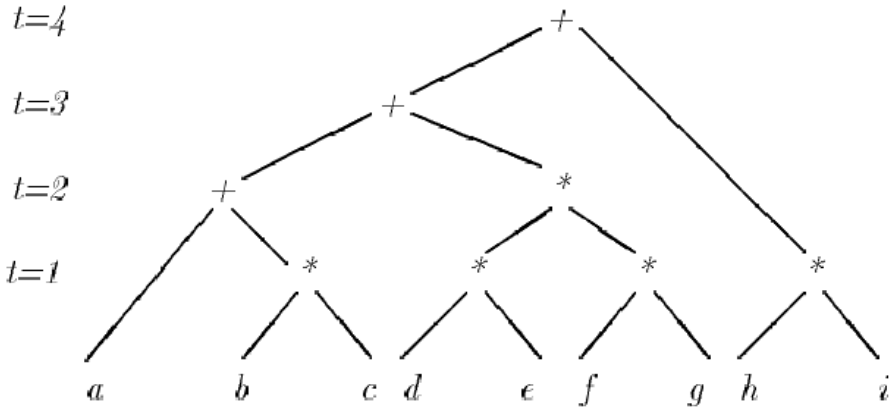
## PARALELISM LA NIVELUL EXPRESIILOR ARITMETICE

Se consideră că o expresie aritmetică este constituită din variabile şi operatori de tip adunare, scădere, înmulţire şi împărţire.

În procesarea paralelă, evaluarea unei expresii aritmetice este bazată pe selectarea unei expresii echivalente care poate fi efectuată într-un număr minim de paşi.

Două expresii aritmetice $E$ şi $\overline{E}$ sunt echivalente dacă este posibilă trecerea de la $E$ la $\overline{E}$ prin aplicarea unui număr finit de reguli de comutativitate, asociativitate şi distributivitate.

**Exemplul 1**. Se consideră expresia $E_1 = a+b+c+defg+hi$. Dacă Într-o unitate de timp se pot efectua mai multe operaţii simultan, timpul de calcul este mult redus:

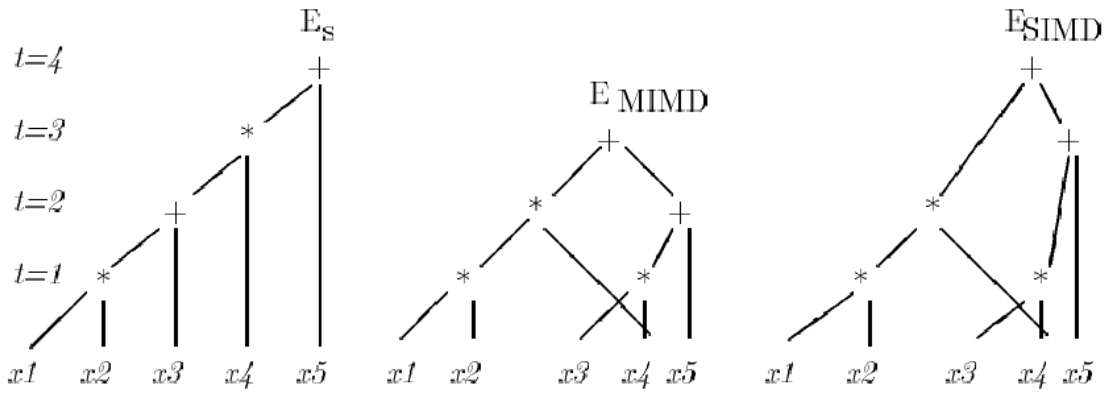



**Exemplul 2**. Fie expresia $E_2 = (x_1x_2 + x_3)x_4 + x_5$.

Intr-un calculator secvenţial, expresia este evaluată prin $E_s = ((((x_1 \cdot x_2) + x_3) \cdot x_4) + x_5)$.

Pentru calculul paralel se recomandă o expresie echivalentă $E_2 = x_1x_2x_4 + x_3x_4 + x_5$, care se evaluează prin $E_p = ((((x_1 \cdot x_2) \cdot x_4) + ((x_3 \cdot x_4) + x_5))$.

Se notează cu *t*, numărul de paşi necesari pentru evaluare (număr de unităţi temporale, presupunând că adunarea şi înmulţirea consumă o unitatea de timp), *p*, numărul de procesoare şi *s*, numărul total de operaţii. Etapele evaluării seriale, respectiv prin procesarea cu $p = 2$ elemente, sunt prezentate în următoare, ce prezintă evaluarea: (a) secvenţială a expresiei $E_2$ (b) în paralel a expresiei $E_p$ pe un MIMD (c) pe un SIMD.




(a) (b) (c)

Pentru cazul paralel, se efectuează simultan toate operaţiile unui nivel. Numărul de paşi a scăzut cu o unitate, dar numărul total de operaţii a crescut: pentru cazul serial *p*=1, t=4, *s*=4, iar pentru cazul paralel, *p*=2, *t*=3, *s*=5.

Exemplul corespunde cazului utilizării unui calculator de tip MIMD în care sunt posibile simultan diferite operaţii. Dacă se utilizează un sistem SIMD, evaluarea expresiei echivalente se face în patru paşi *t*=4.

**Exemplul 3**. Expresiile echivalente optime depind de sistemul utilizat. De exemplu, expresia evaluată secvenţial

$$E_3 = ((((x_1 \cdot x_2) + x_3) \cdot x_4) + x_5)x_6 + x_7$$

este echivalentă cu $E_{MIMD} = ((((x_1x_2)(x_4x_6)) + (x_3(x_4x_6))) + ((x_5x_6) + x_7))$,
care este evaluată în patru paşi pe un sistem MIMD şi în cinci paşi pe un sistem SIMD. Pe de altă parte $E_{SIMD} = ((((x_1x_2) + x_3)(x_4x_6)) + ((x_5x_6) + x_7))$
este evaluată în patru paşi pe un sistem SIMD.

Trecerea de la o expresie la alta echivalentă estimabilă prin mai puţini paşi paraleli se realizează pe baza analizei arborelui binar de evaluare. Se determină un arbore binar echivalent care are o înălţime mai redusă.

## PARALELISM LA NIVELUL INSTRUCŢIUNILOR

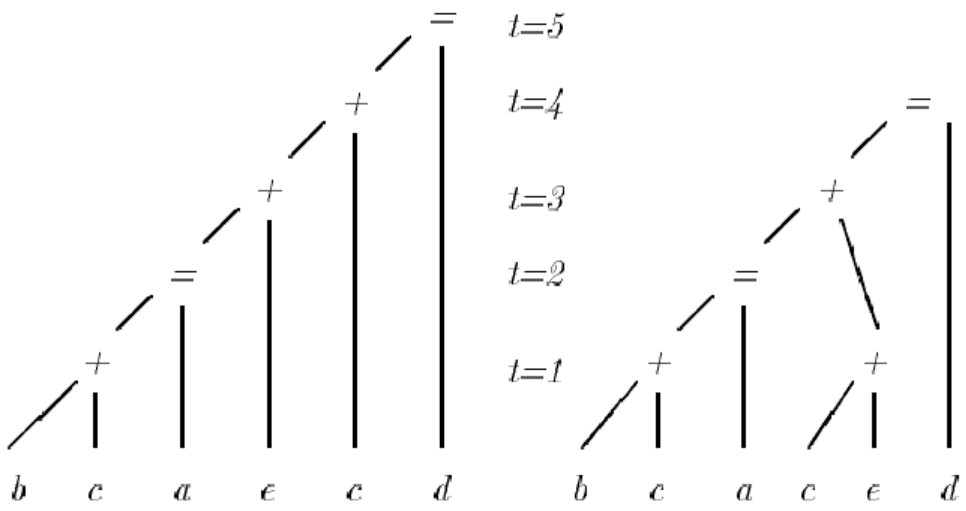
Analog grafului de dependenţă date-operaţii dintr-o expresie dată, se poate introduce graful de dependenţă instrucţiune-date. Modul convenţional de a privi programul ca o listă de instrucţiuni, care manipulează date, aflate în locaţii fixe, într-o ordine secvenţială, nu mai este actual. Pentru exploatarea paralelismului la nivel de instrucţiune este necesară descompunerea programului.

**Exemplu 1**. Se consideră secvenţa de atribuiri

a = b + e,
d = e + f,

g = a * d.

Primele două atribuiri pot fi realizate în paralel, după cum se vede în figura de mai jos, (a):




(a) (b)

**Exemplul 2**. Se consideră secvența de atribuiri

a = bc,

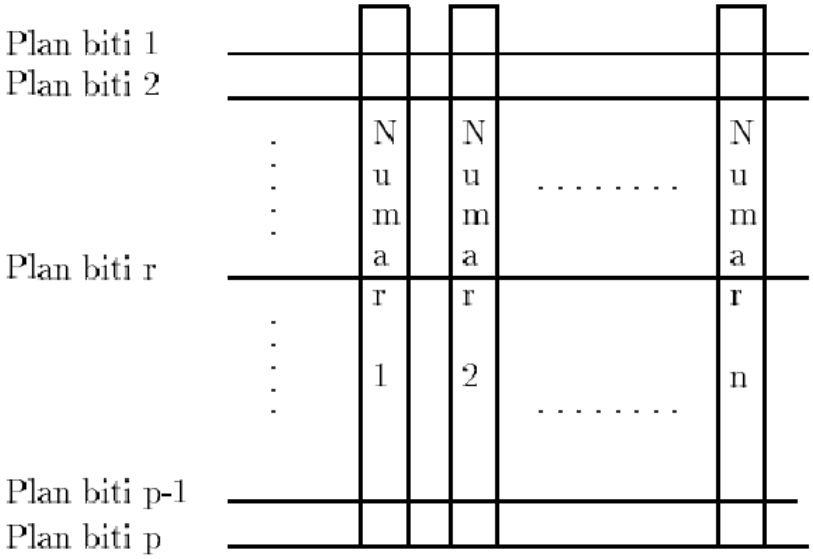
d = c + e + a.

Evaluările paralele se realizează mult mai rapid decât evaluările seriale, după cum se vede în figura de mai sus, (b).

## ALGORITMI PENTRU SISTEME ORGANIZATE PE BIȚI

Intr-un sistem organizat pe biți, fiecare procesor operează asupra biților unei date în paşi consecutivi.

Dacă se consideră o reprezentare pe verticală a datelor, memoria poate fi privită ca fiind compusă din mai multe plane de biți, ca în figura următoare:



Operațiile asupra unui întreg plan de biți se realizează în paralel. Dacă se consideră *n* numere cu *p* biți (în reprezentare verticală), la un moment dat sunt accesați biții din planul *r* al fiecărui număr (*r =1,...,p*).

Presupunând că se dispune de un număr suficient de procesoare, algoritmii pe biți nu depind în cazul reprezentării verticale ale datelor, de numărul acestora, ci de lungimea în biți a datelor.

**Exemplul 1**. Se determină maximul unei mulțimi de numere naturale. Inițial, toate numerele sunt considerate candidați. Se examinează succesiv fiecare plan de biți. La o etapă se elimină din competiție numerele care prezintă zero în planul examinat. De exemplu, se determină maximul dintre 6, 8, 12, 13, 7 şi 13. Se notează cu * numerele rămase în competiție la fiecare etapă:

| | 6 | 8 | 12 | 13 | 7 | 13 |
|---|---|---|---|---|---|---|
| Etapa 0 | * | * | * | * | * | * |
| | 0 | 1 | 1 | 1 | 0 | 1 |
| Etapa 1 | | * | * | * | | * |
| | 1 | 0 | 1 | 1 | 1 | 1 |
| Etapa 2 | | | * | * | | * |
| | 1 | 0 | 0 | 0 | 1 | 0 |
| Etapa 3 | | | * | * | | * |
| | 0 | 0 | 0 | 1 | 1 | 1 |
| Etapa 4 | | | | * | | * |

**Exemplul 2**. Când se utilizează tehnica dublării recursive pentru însumarea a *n* numere, numărul de procesoare utilizate se diminuează (50% la primul pas, 25% la al doilea ş.a.m.d). Într-un sistem organizat pe biți, aritmetica operațiilor este împărțită între procesoare în scopul utilizării maxime. La pasul *k* al procesului de dublare recursivă fiecare operație de adunare este distribuită între procesoarele disponibile. Un grup de procesoare poate lucra asupra biților cei mai semnificativi, iar alt grup asupra biților mai puțin semnificativi. Astfel toate procesoarele sunt utilizate la maxim.

**Exemplul 3**. Algoritmii pe biți asociați funcțiilor elementare presupun înmulțiri şi împărțiri în binar care se efectuează simultan, astfel încât timpul de estimare este comparabil cu cel al unei înmulțiri sau împărțiri. Fie de exemplu, problema evaluării funcției exponențiale. Se cere evaluarea lui $e^x$, $0 < x < 1$, în condițiile în care valorile ei, $e_i^a, a_i = 2^{-i}, i = 1,...,p$ sunt cunoscute. Se presupune că *x* se reprezintă În binar prin $b_1b_2...b_p$. Atunci $e^x = e^{b_1a_1}..e^{b_pa_p}$. Se calculează mulțimea valorilor

$E_0 = 1, E_k = E_{k-1}e^{e_k}, b_k = 1, E_k = E_{k-1}, b_k = 0, k = 1,...,p$. Atunci $E_p = e^x$.

Evaluarea se efectuează printr-o singură parcurgere a planelor de biți a lui *x*.

Exemple similare se pot construi pentru problemele determinării rădăcinii pătrate, determinării logaritmului, a valorilor funcțiilor trigonometrice sau ridicarea la pătrat.

Algoritmii la nivel de bit sunt de asemenea utilizați la sortare şi prelucrarea imaginilor.

## SORTARE

Considerând numerele $z_1,...,z_n$ poziționate inițial în locațiile 1,...,*n*, se cere reaşezarea acestora, astfel încât pe prima poziție să se afle cel mai mic număr din secvență, pe poziția a doua, al doilea în secvență crescătoare ş.a.m.d. Există o serie de strategii clasice de sortare, cum sunt:

• **sortarea prin inserție**, când fiecare număr este inserat pe poziția corectă relativ la o secvență anterior sortată,

• **sortarea prin selecție**, când se selectează cel mai mic număr şi se scoate din secvență, apoi se selectează cel mai mic din secvența rămasă ş.a.m.d,

• **sortarea prin numărare**, când fiecare număr este comparat cu celelalte şi numărul celor mai mici decât acesta indică poziția în secvența finală,

• **sortarea prin interschimbare**, când numerele din două locații curente îşi schimbă poziția între ele dacă ordonarea în secvență crescătoare o cere.

O atenție deosebită în cazul procesării paralele a fost acordată algoritmilor de interschimbare. Intr-un calculator secvențial fiecare interschimbare se execută separat. Algoritmii paraleli permit, într-un singur pas, mai multe comparări şi interschimbări simultan. Un algoritm serial de sortare a *n* numere necesită cel puțin $O(n\log_2 n)$ comparări. Dacă într-un algoritm paralel pot fi efectuate simultan *n* comparații la un pas, atunci timpul de procesare este de ordinul $O(\log_2 n)$. Din păcate, algoritmii seriali care se realizează cu un număr minim de comparații nu pot fi restructurați pentru cazul paralel, astfel Încât să se obțină acest optim teoretic. Totuşi este posibilă paralelizarea unor algoritmi seriali de sortare care necesită un număr de comparații de ordinul $O(n^2)$, astfel încât timpul de calcul paralel să fie proporțional cu $O(n)$. Astfel, sortarea par-impar şi sortarea cu arbori a *n* numere care utilizează $O(n)$ procesoare $O(n)$ în paralel se realizează în $O(n)$ paşi.

Calculatoarele paralele actuale recepționează datele secvențial, fapt care produce o mare întârziere. Dacă, de exemplu, calculatorul poate recepționa o dată per unitatea de timp, atunci *n* unități de timp sunt necesare pentru încărcarea datelor. Se speră că în viitorul apropiat va fi posibilă introducerea şi afişarea datelor în paralel.

Se presupune, în cele ce urmează, că datele se află inițial în memoria procesoarelor.

## SORTAREA PRIN NUMĂRARE

Modele teoretice pentru procesoare paralele cu memorie comună au fost propuse pentru sortarea a *n* numere prin $O(\log_2 n)$ comparații, cu $O(n^2)$ procesoare.

Sortarea este realizată prin determinarea indexului în secvența finală pe baza comparării unui număr cu toți ceilalți. Acest lucru este posibil într-un calculator paralel cu memorie comună, unde oricare procesor are acces la oricare locație din memorie. Fiecare număr este comparat simultan cu toate celelalte într-un singur pas utilizând $n(n-1)$ procesoare. Fiecărui număr i se asociază o mulțime de biți: pentru o pereche de numere $(x, y)$, bitul lui $x$ este 1 dacă $x > y$, altfel este 0. Se contorizează numărul de biți nenuli. Dacă numărul de biți nenuli corespunzători lui $x$ este $p$, atunci poziția lui $x$ în secvența sortată este $p + 1$ (sortare prin numărare).

**Exemplu**. Fie $n$=4 şi $z$=(4,2,1,3). Prima etapă este achiziția contoarelor. Se consideră 16 procese care fiecare evaluează rezultatul comparării a câte două valori ale şirului:

$$\begin{bmatrix} 4:4 & 4:2 & 4:1 & 4:3 \\ 2:4 & 2:2 & 2:1 & 2:3 \\ 1:4 & 1:2 & 1:1 & 1:3 \\ 3:4 & 3:2 & 3:1 & 3:3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix}.$$

În etapa a doua se calculează rangul fiecărui element:
linia 1 (cheia 4): 1+1+1+1=4
linia 2 (cheia 2): 0+1+1+0=2
linia 3 (cheia 1): 0+0+1+0=1
linia 4 (cheia 3): 0+1+1+1=3

In faza a treia se reaşează datele conform rangului nou calculat.

Dacă se dispune doar de *n* procesoare, determinarea rangului în şirul ordonat se poate desfăşura în modul următor. Fie *c* vectorul ce se construieşte pentru rangurile elementelor, iar *a* un vector intermediar. Ideea este de a compara simultan *n - 1* perechi succesive din *z* privit ca un inel de date cu ajutorul lui *a*. Dacă $a_i < z_i$, atunci $c_i$ este incrementat cu unu. Se efectuează o mutare ciclică la stânga a elementelor lui *a* care inițial este identic cu *z*. Dacă operația se repetă de *n* ori, fiecare cheie va fi comparată cu fiecare altă cheie. Ultima operație este rearanjarea lui *z* conform lui *c*.

**Exemplu**. Fie *n* = 6 şi *z* = (6,3,4,5,2,1). La sfârşitul fiecărui pas în paralel se obține:

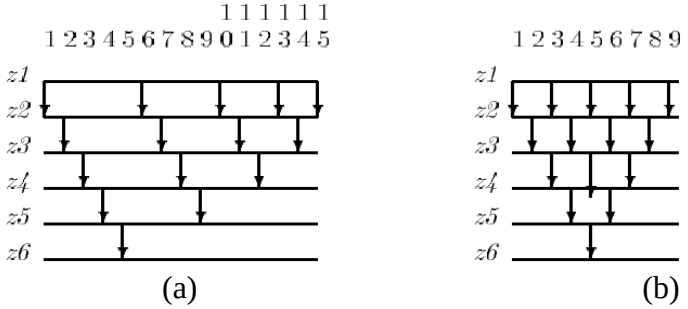
i = l, c = (0,0,0,0,0,0) a = (3,4,5,2,1,6)
i = 2, c = (1,0,0,1,1,0) a = (4,5,2,1,6,3)
i = 3, c = (2,0,1,2,1,0) a = (5,2,1,6,3,4)
i = 4, c = (3,1,2,2,1,0) a = (2,1,6,3,4,5)
i = 5, c = (4,2,2,3,1,0) a = (1,6,3,4,5,2)
i = 5, c = (4,2,2,3,1,0) a = (1,6,3,4,5,2)
i = 6, c = (5,2,3,4,1,0) a = (1,2,3,4,5,6) este z sortat

## PROCEDEUL BULELOR

Algoritmul serial al bulelor se bazează pe compararea succesivă a două elemente alăturate din secvență (cu interschimbare), astfel încât, după o parcurgere a secvenței, maximul va fi plasat pe ultima poziție. La pasul următor se efectuează comparările în secvența rămasă prin excluderea maximului. Numărul de paşi este $\frac{1}{2}n(n-1)$.

În figura de mai jos (a) se prezintă modul în care evoluează comparările pentru cazul sortării a *n* = 7 numere. Liniile orizontale reprezintă locațiile, săgețile comparările, vârful săgeții, locul pe care îl va ocupa maximul, iar numerele de sus, etapele.

Într-o generalizare naturală pentru calculul paralel, fiecare etapă porneşte când dispune de toate elementele necesare. Pentru exemplul anterior etapele sunt prezentate în figura de mai jos (b).




(a) (b)

Numărul de paşi efectuaţi în paralel este 2n - 3. In cazul utilizării unui sistem paralel cu transmitere de mesaje, algoritmul este optim pentru o conectare liniară a procesoarelor.

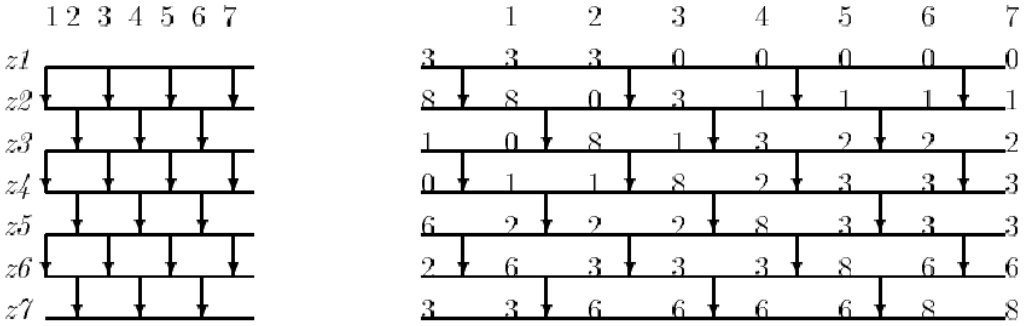
## SORTAREA PAR-IMPAR

Metoda de sortare par-impar presupune repetarea ciclică a operaţiilor:

(a) $z_{2i-1} \leftarrow \min\{z_{2i-1}, z_{2i}\}, \quad z_{2i} \leftarrow \max\{z_{2i-1}, z_{2i}\}$

(b) $z_{2i} \leftarrow \min\{z_{2i}, z_{2i+1}\}, \quad z_{2i+1} \leftarrow \max\{z_{2i}, z_{2i+1}\}$

Sortarea par-impar necesită mai puţini paşi decât metoda bulelor. In *n* paşi sortarea este terminată şi oricare alt algoritm care utilizează interconectarea liniară nu sortează numerele într-un număr mai mic de paşi.




## SORTARE CU ARBORI

O listă de $n=2^k$ numere este sortată prin determinarea minimului, înlăturarea acestuia, determinarea următorului minim, înlăturarea acestuia ş.a.m.d.

Pentru exemplificare, în figura de mai jos, se consideră primii 6 paşi ai algoritmului pentru *n=8* şi lista (2, 3, 9, 7, 1, 8, 3, 4).

Iniţial:

```
2   3   9   7   1   8   3   4
  *       *       *       *
      *               *
          *
```

Pas 1:

```
*   3   9   *   *   8   *   4
  2       7       1       3
      *               *
          *
```

Pas 2:

```
2   *   9   *   *   8   *   4
  *       7       *       3
      2               1
          *
```

Pas 3:

```
*   *   9   *   *   *   *   4
  3       7       8       3
      2               *
          1
```

Pas 4:

```
        *    *    9    *    *    *    *    4
             3         7         8         *
                  2                   3
                            *
```

Pas 5:

```
        *    *    9    *    *    *    *    *
             3         7         8         4
                  *                   3
                            2
```

Pas 6:

```
        *    *    9    *    *    *    *    *
             *         7         8         4
                  3                   3
                            *
```

Fiecare procesor de la un nivel *i* răspunde de o anumită locație şi este conectat direct la două elemente de procesare "fiice" (locații) de la nivelul *i*+1, care după fiecare pas al algoritmului pot fi ocupate sau libere. Inițial toate locațiile procesoarelor interne arborelui sunt libere. La fiecare pas se aplică următoarele reguli:

1. fiecare procesor liber primește informații despre locațiile fiice şi dacă amândouă sunt ocupate, compară valorile şi reține pe cea mai mică în locația sa, trimițând pe cea mai mare la locația de unde provine. Locația din care provine valoarea selectată este eliberată;
2. dacă procesorul rădăcină este ocupat cu un număr, atunci acesta este scos;
3. toate procesoarele din întreaga rețea operează simultan;
4. minimul este eliminat prin nodul rădăcină.

Când se utilizează un sistem paralel cu transmitere de mesaje, algoritmul se recomandă pentru cazul unei rețele de interconectare de tip arbore binar.

## SORTAREA RAPIDĂ

In sortarea rapidă serială, rezultatul fiecărei comparații determină care elemente vor fi în continuare comparate. Astfel, ordinea în care au loc comparațiile este cunoscută numai la faza de execuție. Introducerea paralelismului este destul de dificilă.

Fiind date *n* numere, $z_1,..., z_n$, metoda rapidă clasică rearanjează numerele astfel încât un anumit $z_i$ se află pe poziția finală: elementele din stânga sunt mai mici decât acesta, iar cele din dreapta mai mari. Procesul se repetă recursiv asupra sublistelor din ambele părți ale lui $z_i$ şi subsublistelor acestora. Fiecare sublistă poate fi asociată cu un procesor dintr-un sistem paralel. Paralelismul potențial se dublează la fiecare trecere la subliste. Se recomandă în cazul unui sistem cu transmitere de mesaje şi o rețea de interconectare tip arbore binar.

Pentru a obține un arbore de descompunere cât mai balansat, se utilizează metoda medianei. Un pas presupune:

- compararea elementului din mijlocul listei curente cu toate celelalte pentru determinarea numerelor mai mici şi a celor mai mari;
- divizarea listei curente în trei: elementul din mijloc, sublista numerelor mai mici decât acesta şi sublista numerelor mai mari decât acesta.

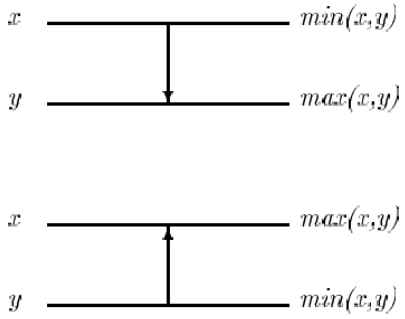
Prima etapă poate fi procesată eficient în paralel deoarece toate comparările pot fi efectuate simultan. Etapa a doua presupune procesarea paralelă a sublistelor.

Dacă se consideră un sistem organizat pe biți, prima etapă nu mai este necesară. Numerele cu bitul cel mai semnificativ egal cu 1 aparțin sublistei cu elemente mari, iar cele cu bitul semnificativ egal cu 0 aparțin primei subliste. În pasul al doilea este studiat bitul următor din fiecare sublistă. Pentru exemplificare, în figura de mai jos se consideră *z* = (13, 7, 13, 6, 4, 12). În mod analog, sortarea se poate face pornind de la bitului cel mai puțin semnificativ.

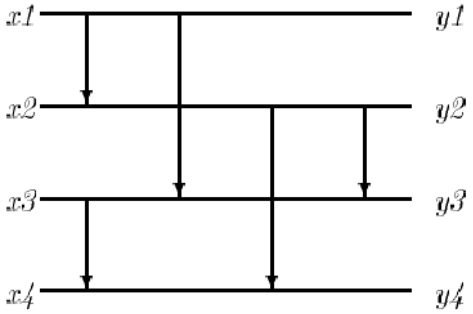
| Inițial | | Binar | Etapa 1 | Etapa 2 | Etapa 3 | Etapa 4 | | Final |
|---|---|---|---|---|---|---|---|---|
| 13 | = | 1101 | 0111 | 0111 | 0100 | 0100 | = | 4 |
| 7 | = | 0111 | 0110 | 0110 | 0111 | 0110 | = | 6 |
| 13 | = | 1101 | 0100 | 0100 | 0110 | 0111 | = | 7 |
| 6 | = | 0110 | 1101 | 1101 | 1101 | 1100 | = | 12 |
| 4 | = | 0100 | 1101 | 1101 | 1101 | 1101 | = | 13 |
| 12 | = | 1100 | 1100 | 1101 | 1100 | 1101 | = | 13 |
| | | ↑ | ↑ | ↑ | ↑ | | | |

## SORTAREA BITONICĂ

Se defineşte un *comparator* ca fiind un modul a cărui intrări sunt două valori *x* şi *y* iar ieşirile sunt min{*x*,*y*} şi max{*x*,*y*}:




O rețea de comparare este realizată din comparatori. Un exemplu de asemenea rețea este prezentată în figua de mai jos, unde *xl*, *x2*, *x3*, *x4* sunt intrările, iar *y1*, *y2*, *y3*, *y4*, ieşirile (ce reprezintă datele de intrare ordonate crescător).




Dimensiunea unei rețele de comparare este numărul de comparat ori utilizați în rețea. Adâncimea este lungimea celui mai lung drum dintre o intrare şi o ieşire. In figura anterioară, dimensiunea rețelei este 5, iar adâncimea este 3.

O **rețea de sortare** este o rețea de comparare a cărui ieşiri reprezintă valorile de intrare în ordine crescătoare.

In procesarea secvențială se caută o rețea de sortare cu cea mai mică dimensiune. In procesarea paralelă se caută construirea unei rețele de sortare cu cea mai mică adâncime şi din toate rețelele cu cea mai mică adâncime, cea cu cea mai mică dimensiune. O asemenea rețea optimă este cea introdusă de algoritmul sortării bitonice.

O secvență de numere ($z_1$,..., $z_n$) se numeşte bitonică dacă, după o anumită deplasare ciclică, constă dintr-o secvență ascendentă urmată de una descendentă, adică există un $j<n$ (numărul de deplasări ciclice) şi un $l<n$ (lungimea secvenței crescătoare) pentru care $x_{j \bmod n} \le x_{(j+1) \bmod n} \le ... \le x_{l \bmod n}$ şi $x_{(l+1) \bmod n} \ge ... \ge x_{(j+n-1) \bmod n}$. De exemplu, secvența (3, 5, 6, 6, 4, 2, 1, 3) este bitonică, deoarece prin $j$=2 rotații se poate obține (1, 3, 3, 5, 6, 6, 4, 2). Secvența (-5, -9, -10, -5, 2, 7, 35, 37) este de asemenea bitonică, având caracteristicile $j$ = 2, $l$ = 6.

Algoritmul de sortare bitonică se bazează pe observația că, dacă $z=(z_1,..., z_{2n})$ este o secvență bitonică şi se fac comparațiile şi interschimbările necesare astfel încât să se obțină secvența ($L(z)$, $R(z)$) unde

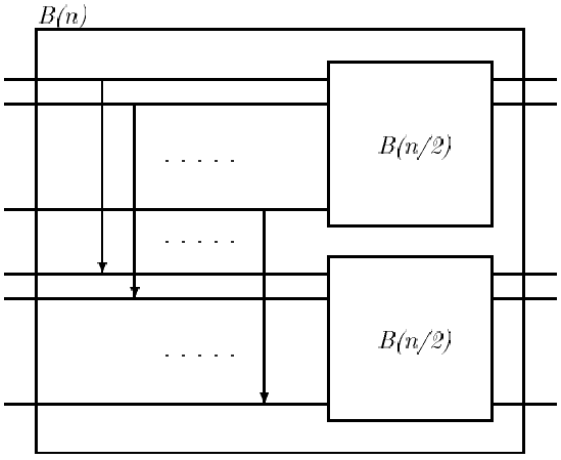
$$L(z) = (\min\{z_1, z_{n+1}\},...,\min\{z_n, z_{2n}\}, \quad R(z) = (\max\{z_1, z_{n+1}\},...,\max\{z_n, z_{2n}\}$$

atunci cele două jumătăți ale secvenței sunt bitonice şi conțin cele $n$ cele mai mici numere, respectiv cele $n$ cele mai mari numere ale secvenței inițiale. Dată o secvență bitonică de lungime $n = 2^k$, aplicând procedeul divide et impera la întreaga secvență, (la fiecare jumătate a ei, la jumătățile acestora ş.a.m.d.), atunci în $k$ paşi paraleli (un pas fiind constituit din operațiunea de determinare a unei perechi ($L$,$R$)) secvența este sortată.

Rețeaua de sortare se construieşte astfel:

• rețeaua de sortare $B(2)$ cu două intrări constă dintr-un singur comparator ce realizează ordonarea crescătoare a intrărilor;

• rețeaua de sortare $B(m)$ este constituită dintr-o coloană de $m/2$ comparatori urmată de două rețele de sortare $B(m/2)$. Scopul coloanei de $m/2$ comparatori este generarea lui $L(z)$ în primele $m/2$ linii orizontale şi $R(z)$ în următoarele $m/2$:

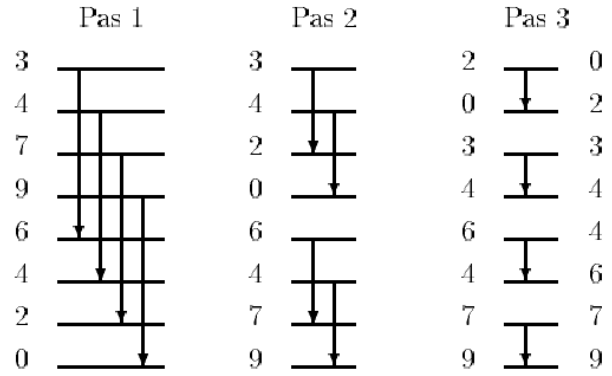



Pentru un număr de intrări $n$, putere a lui doi, adâncimea rețelei este $D(n)= \log_2 n$, iar numărul de comparatori este $C(n) = n \log_2 n/2$. Aceste valori se obțin din recursiile următoare:

$D(n)$=l+$D(n/2)$, $D(2)$=1,

$C(n)=n/2+2C(n/2)$, $C(2)$=1.

De exemplu, fie $k$ = 3 şi z =(3, 4, 7, 9, 6, 4, 2, 0). Procedeul de sortare bitonică este reprezentat în figura următoare:




## CĂUTARE

**Problema**. Fie $X=(x_1,...,x_n)$ elemente distincte dintr-o mulțime ordonată $S$ astfel încât $x_1 < x_2 < ... < x_n$. Dat un element $y$ din $S$, se cere determinarea indexului $i$ pentru care $x_i \le y \le x_{i+1}$ unde, $x_0 = -\infty, x_{n+1} = \infty$, iar $-\infty$ şi $\infty$ sunt două elemente care adăugate la $S$ satisfac $-\infty < x < \infty, \forall x \in S$.

**Algoritmul secvențial**. Metoda secvențială a căutării binare rezolvă această problemă într-un timp $O(\log_2 n)$. Ea constă în compararea lui $y$ cu mijlocul listei $X$. Căutarea este terminată sau problema este restricționată la jumătatea stângă sau dreaptă a lui $X$. Procesul este repetat până când un element $x_i$ a fost descoperit astfel încât $y=x_i$ sau dimensiunea sublistei este egală cu unu.

**Algoritmul paralel**. O extensie naturală a metodei de căutare binară la procesarea paralelă este căutarea paralelă în care $y$ este comparat concurent cu mai multe elemente, fie $p$ elemente, ale lui $X$ şi se separă $X$ în $p + 1$ segmente de lungime aproximativ egale. Pasul de comparare conduce la identificarea unui element $x_i$ egal cu $y$ sau la restricționarea căutării la unul dintre cele $p + 1$ segmente. Se repetă acest proces până când un element $x_i$ este găsit astfel încât $y=x_i$ sau numărul de elemente din sublista sub considerare este mai mic decât $p$.

## INTERCLASARE

**Problema**. Fie $A=(a_1,...,a_m)$ şi $B=(b_1,...,b_n)$ două secvențe sortate. Se cere determinarea secvenței ordonate $C=(c_1,...,c_{n+m})$ care conține toate elementele celor două secvențe ordonate.

**Algoritmul secvențial.** $A$ şi $B$ sunt partiționate în perechi de sub secvențe astfel încât să se obțină o secvență ordonată prin interclasarea perechilor.

**Algoritmul paralel**. Se realizează mai multe interclasări simultan.

Se notează cu rang($a_i$ : $A$) numărul de componente ale lui $A$ mai mici sau egale cu $a_i$. Rangul poate determinat prin căutare binară sau paralelă.

Fie, de exemplu, cazul în care $n$ nu este prim ci are un divizor $k$. Atunci se caută $k$ perechi ($A_i$, $B_i$) de secvențe ale lui $A$ şi $B$ astfel încât

• numărul elementelor lui $B_i$ este $n / k$;

• fiecare element a lui $A_i \cup B_i$ este mai mare decât oricare elemente din $A_{i-1} \cup B_{i-1}$, pentru $i=1,...,k$-1.

O soluție este partiționarea $B_i = (b_{in/k+1},....,b_{(i+1)n/k})$, $A = (a_{j(i)},...,a_{j(i+1)})$, unde $j(i)$ = rang($b_{in/k}$ : $A$) şi interclasarea concurentă a perechilor ($A_i$, $B_i$), $i$=1,...,$k$.

De exemplu, $A$ = (4,6,7,10,12,15,18,20), $B$ = (3,9,16,21). Atunci $n$=4, $k$=2. Cum rang(9:$A$)=3, se partiționează $A_1$=(4,6,7), $B_1$=(3,9) şi $A_2$=(10,12,15,18,20), $B_2$=(16,21) care sunt interclasate concurent (eventual utilizând recursivitatea).

## PROBLEMA COLORĂRII UNUI GRAF

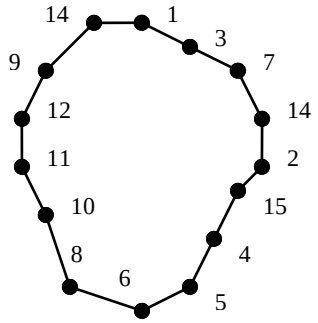
O $k$-colorare a unui graf $G$=($V$, $E$) este o funcție definită pe mulțimea culorilor desemnate prin numere $c : V \to \{0,1,...,k-1\}$ cu proprietatea $c(i) \neq c(j)$ dacă $(i, j) \in E$.

Se consideră cazul particular al grafului unui inel. Pentru un asemenea graf este posibilă o 3-colorare.

**Algoritmul secvențial**. Se traversează inelul pornind de la un vârf oarecare şi se asignează culori arbitrare din mulțimea {0, 1} la vârfuri adiacente. A treia culoare poate fi necesară pentru terminarea ciclului. Acest algoritm secvențial este optimal, dar nu conduce la un algoritm paralel rapid.

Problema pentru cazul general al unui graf oarecare constă în partiționarea mulțimii vârfurilor în clase astfel încât fiecare clasă să fie asignată unei aceleiaşi culori.

Presupunem că arcele lui $G$ sunt specificate printr-o funcție S astfel încât $S(i)=j$ dacă $(i, j) \in E$. Presupunem că inițial colorarea lui $G$ este c($i$) = $i$, pentru orice $i$. Se poate reduce numărul culorilor prin următoarea procedură simplă care ține seama de reprezentarea binară a numerelor asociate culorilor. Dacă $(i)_{10} = (i_{t-1}..i_k..i_0)_2$, atunci al $k$-lea bit cel mai puțin semnificativ al lui $i$ este $i_k$. Se asociază fiecărui vârf un proces. Procesul asociat vârfului $i$, determină cel mai puțin semnificativ bit în care $c(i)$ şi $c(S(i))$ diferă şi se modifică funcția de culoare $c'(i) = 2k + c(i)_k$, unde $c(i)_k$ este al $k$-lea cel mai puțin semnificativ bit al lui $c(i)$. Calculele se efectuează concurent. In exemplul din figura de mai jos, numărul culorilor este redus de la 15 la 6.




| ***Nr*** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ***i*** | 1 | 3 | 7 | 14 | 2 | 15 | 4 | 5 | 6 | 8 | 10 | 11 | 12 | 9 | 13 |
| ***c*** | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| ***k*** | 1 | 2 | 0 | 2 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 2 | 2 |
| ***c'*** | 2 | 4 | 1 | 5 | 0 | 1 | 0 | 1 | 3 | 2 | 0 | 1 | 0 | 4 | 5 |

Fie $t$ numărul de biți utilizați pentru reprezentarea culorilor prin $c$. Atunci, fiecare culoare utilizată în $c$' poate fi reprezentată cu $[\log_2 t] + 1$ biți, unde [.] desemnează

cel mai mic întreg mai mare decât valoarea dintre paranteze. Procedura de reducere a culorilor poate fi aplicată de mai multe ori, atâta timp cât t > [$\log_2 t$] + 1, adică $t > 3$. Pentru cazul $t = 3$, rezultă o colorare cu cel mult 6 culori, $0 \le c(i) \le 5, \quad 0 \le k \le 2$.

Numărul de culori poate fi redus la trei după cum urmează. Procedura adițională de recolorare constă în trei etape, fiecare ocupându-se de vârfurile de anumită culoare *l* numerotată de la 3 la 5. Fiecare vârf *i* de culoare *l* se recolorează cu culoarea cu cel mai mic număr posibil din mulțimea {0, 1, 2}, adică cea mai mică culoare diferită de culorile predecesorului şi succesorului în cerc. După ultima etapă se obține o 3-colorare.

In exemplul din figura anterioară, culorile vârfurilor după aplicarea procedeului de reducere sunt 0, 1, 2, 3, 4, 5. Vârful 6 este unicul de culoare 3. Deoarece vecinii săi, vârfurile 5 şi 8, sunt colorate cu 1 şi 2, vârful 6 este recolorat cu 0. Apoi, vârfurile 3 şi 9 sunt recolorate cu 0, respectiv 1. În final, vârfurile 13 şi 14 sunt recolorate cu 0, respectiv 2.

## Capitolul 4

# ALGORITMI NUMERICI PARALELI

## MODALITĂȚI DE CONSTRUIRE A ALGORITMILOR NUMERICI PARALELI

În modelarea algoritmilor paraleli numerici există o serie de căi:

• analiza algoritmului serial şi convertirea sa într-o procedură care operează cu obiecte matematice compuse, cum ar fi vectorii şi matricile, astfel încât mai multe date să fie procesate simultan. De multe ori nu cel mai eficient algoritm serial este cel mai bun în paralel. Metode mai puțin eficiente în procesarea serială pot poseda un înalt grad de paralelism şi ca urmare sunt adaptabile calculului paralel;

• algoritmi iterativi echivalenți cu anumite metode directe posedă un înalt grad de paralelism şi pot fi organizați, sistematic, pe un calculator paralel;

• calculul poate fi divizat în subunități şi distribuit între procesoare.

**Exemplu.** Cele mai comune exemple de paralelism în analiza numerică sunt calculele cu vectori şi matrici. De exemplu, cele *n* multiplicări necsare produsului scalar a doi vectori pot fi evaluate simultan, la fel şi cele *n* - 1 adunări. Modalitatea cea mai banală de paralelizare este distribuirea sarcinilor procesoarelor la componentele vectorului soluție. Un alt exemplu sunt iterațiile Jacobi pentru rezolvarea sistemului liniar *Ax = b*, ce se scriu recursiv

$x^{(n+l)} = (I - A)x^{(n)} + b.$

Fiecare componentă a membrului drept poate fi calculată de către un procesor dintrun sistem cu memorie comună. Deşi algoritmul se poate implementa în paralel, este executat într-o modalitate secvențială. Se poate spune că un asemenea algoritm are o natură dublă: şi secvențial şi paralel.

## EVALUAREA RELAȚIILOR RECURSIVE

Soluția problemelor numerice se reduce adesea la evaluarea ultimului termen sau a tuturor termenilor unei relații recursive.

Procesarea paralelă standard a relațiilor recursive are la bază una din următoarele tehnici:

• tehnica dublării recursive,

• tehnica de reducere ciclică.

**Exemplul 1**. Fie relația recursivă

$$x_i = a_i x_{i-1} + b_i x_{i-2}, \quad i \geq 2$$

Valorile $x_0, x_1, a_i, b_i$ sunt cunoscute. Relația recursivă poate fi reformulată vectorial

$$y_i = M_i y_{i-1}, \quad i \geq 2$$

unde $M_i = \begin{pmatrix} a_i & b_i \\ 1 & 0 \end{pmatrix}, \quad y_i = \begin{pmatrix} x_i \\ x_{i-1} \end{pmatrix}$

Atunci $y_n = M_n M_{n-1} .. M_2 y_1$.

Produsul matricilor poate fi calculat în paralel prin tehnica dublării recursive descrise la evaluarea expresiilor aritmetice.

**Exemplul 2**. Anumite relații de recurență neliniare pot fi reduse la cazul liniar şi, deci, rezolvate în paralel cu tehnica menționată. De exemplu,

$z_i = a_i + b_i / z_{i-1}$

se reduce la relația recursivă în *x* din exemplul anterior, dacă $z_i = z_0 z_1 .. z_i$. O asemenea ecuație se obține la descompunerea *LU* a unei matrici tridiagonale (*U* şi *L* sunt, în acest caz, matrici bidiagonale).

**Exemplul 3**. Într-o serie de probleme apar recursii de tipul

$$x_i = \frac{a_i x_{i-1} + b_i}{c_i x_{i-1} + d_i}$$

Soluția finală este de tipul $x_n = \frac{Ax_0 + B}{Cx_0 + D}$

unde $\begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} a_n & b_n \\ c_n & d_n \end{pmatrix} .. \begin{pmatrix} a_2 & b_2 \\ c_2 & d_2 \end{pmatrix} \begin{pmatrix} a_1 & b_1 \\ c_1 & d_1 \end{pmatrix}$

Produsul matricial poate fi calculat, de asemenea, prin tehnica dublării recursive.

**Exemplul 4**. In anumite condiții, metoda dublării recursive se poate aplica iterațiilor de forma $x_i = f_i(x_{i-1}), \quad i \geq 1, \quad x_0$ dat.

Soluția are forma $x_n = f_n(f_{n-1}(...f_1(x_0)...))$.

Dacă compunerea este asociativă în clasa funcțiilor $f_i, \quad i \geq 1$, atunci se poate aplica tehnica dublării recursive:

$$x_n = (...((f_n \circ f_{n-1}) \circ (f_{n-2} \circ f_{n-3}))...(f_2 \circ f_1)...)(x_0)$$

## POLINOAME

Fie polinomul

$$P(x) = \sum_{i=0}^{n} a_i x^i$$

Algoritmul secvențial pentru evaluarea valorii polinomului în $x_0$, construit pe baza regulii Horner, presupune procesul iterativ

$b_j = a_j + x_0 b_{j+1}, \quad j = n-1,...,0, \; b_n = a_n$

Valoarea cerută este P(xo) = ba.

Algoritmul Dorn pentru calculul paralel cu *r* procesoare presupune aplicarea simultană a regulii Horner la r subpolinoame, obținute prin partiționarea polinomului inițial. Fie, de exemplu, *r* = 2. Procesul iterativ corespunzător este următorul:

$b_n = a_n, \quad b_{n-1} = a_{n-1}, \quad b_j = a_j + x_0^2 b_{j+2}, \quad j = n-2,...,0$

Atunci $P(x_0) = b_0 + b_1 x_0$.

Polinomul este partiționat astfel:

$$P(x_0) = [a_0 + x_0^2(a_2 + x_0^2(...+ x_0^2(a_{2\lfloor n/2 \rfloor})...] + x_0(a_1 + x_0^2(a_3 + x_0^2(...+ x_0^2(a_{2\lfloor (n-1)/2 \rfloor + 1})...).$$

O altă variantă de paralelizare este aplicarea tehnicii dublării recursive pentru procesul iterativ corespunzător regulii Horner.

## METODE NUMERICE PARALELE DE REZOLVARE A SISTEMELOR DE ECUATII LINIARE

### SISTEME LINIARE TRIDIAGONALE

Sistemele tridiagonale formează o clasă foarte importantă a ecuațiilor algebrice liniare şi intervin în rezolvarea numerică a unor probleme,cum ar fi aproximare cu diferențe finite a ecuațiilor diferențiale cu derivate parțiale.Există metode seriale performante pentru rezolvarea unor asemenea sisteme (exemplu: metoda Gauss de eliminare parțială ) dar noi vom studia doua metode potrivite rezolvării cu mai multe procesoare,utilizând tehnica reducerii ciclice par-impar (care consta in reformularea calculului in vederea evidentierii paralelismului dintr-un calcul serial) şi a dublării recursive (care consta in descompunerea repetata a calculului in perechi de calcule de complexitate egala care pot fi efectuate simultan).

**METODA REDUCERII CICLICE**

Fie sistemul

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

El poate fi scris sub forma unei recurențe liniare cu trei termeni:

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i, \quad i = \overline{1,n},$$

unde $a_1 = c_n = x_0 = x_{n+1} = 0$.

Pentru determinarea necunoscutei $x_i$ ,observăm că aceasta apare în ecuațiile :

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

Prin eliminarea necunoscutelor $x_{i-1}$ şi $x_{i+1}$, se obține o nouă ecuație :

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

Reluând această operație pentru fiecare $i = \overline{1,n}$, se obține tot un system tridiagonal dar numărul coeficienților $a_i^{(1)}$ va fi $n-2$.

Continuând raționamentul şi utilizând cele trei ecuații ale noului system care îl conțin pe $x_i$ ,se obține:

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

numărul coeficienților $a_i^{(2)}$ fiind $n-4$.După $k$ paşi se obțin ecuații de forma:

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

cu $k \le \lfloor \log_2 n \rfloor$, unde coeficienții se obțin recursiv din coeficienții $a_{i-2^{k-1}}^{(k-1)}, a_{i+2^{k-1}}^{(k-1)}, c_{i-2^{k-1}}^{(k-1)}, c_{i+2^{k-1}}^{(k-1)}, b_{i-2^{k-1}}^{(k-1)}, b_{i+2^{k-1}}^{(k-1)}, d_{i-2^{k-1}}^{(k-1)}, d_{i+2^{k-1}}^{(k-1)}, b_i^{(k-1)}, d_i^{(k-1)}$, numărul coeficienților $a_i^{(k)}$, respectiv $c_i^{(k)}$ fiind $n-2^k$ .În acest moment (deci după $k = \lfloor \log_2 n \rfloor$ pași ),

$$b_i^{(k)} x_i = d_i^{(k)} \Rightarrow x_i = d_i^{(k)} / b_i^{(k)}$$

Apoi, prin substituții regresive, se determină valorile tuturor necunoscutelor.Un prim algoritm de determinare a soluției pe baza coeficienților matricilor transformate şi care este optim din punct de vedere al numărului de operații, este următorul:

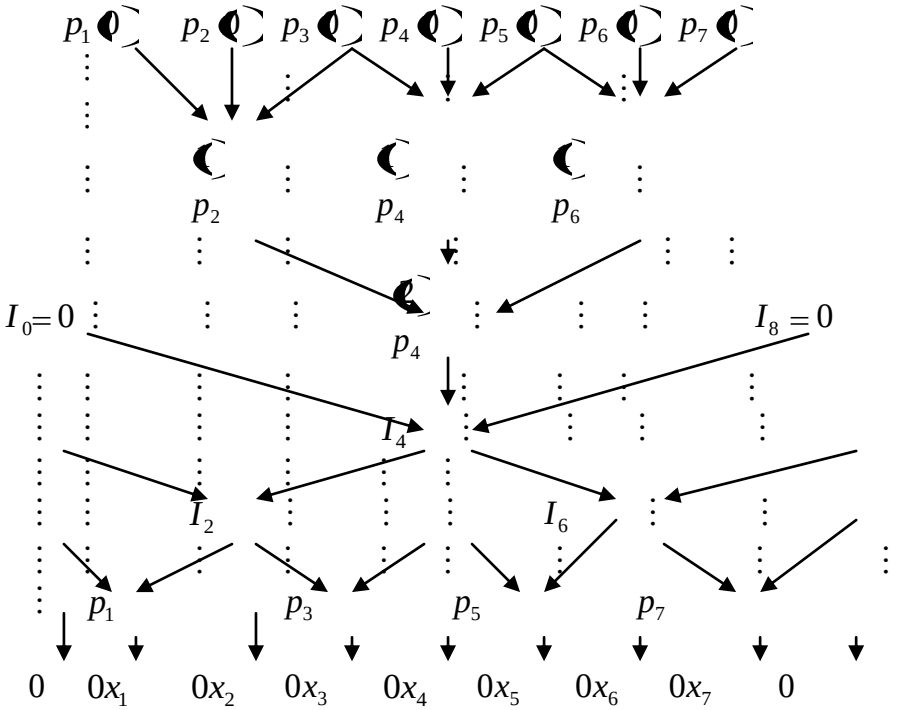
1. Se determină $x_n$ şi apoi, prin substituții regresive, celelalte componente ale vectorului soluție:

a) pentru fiecare $s = \overline{1,k}$ , calculează

$$p^s = (a_i^{(s)}, b_i^{(s)}, c_i^{(s)}, d_i^{(s)}), \text{ unde } i = 2^s, 2^{s+1}, \ldots$$

b) pentru fiecare $s = k, k-1, \ldots, 0$ , se calculează soluția $x_i, i = 2^s, 2^s + 2^{s+1}, \ldots, n$

În figura urmatoare se prezintă algoritmul în cazul unui sistem tridiagonal de dimensiune 7.




Determinarea soluției unui sistem tridiagonal de dimensiune 7 cu un număr minim de operații

Un al doilea algoritm optim din punct de vedere al numărului de paşi care pot fi executați în parallel este prezentat în continuare. În acest caz, pentru eliminarea completă a elementelor extradiagonale se completează matricea inițială cu 1 pe diagonala principală, cu 0 pe subdiagonale şi cu 1 în vectorul liber până când numărul

de ecuații ale sistemului este putere a lui doi minus unu. Fiecare $x_i$ poate fi calculată ca $x_{(n+1)/2}$ dacă se impun condițiile $a_i^{(k)} = c_i^{(k)} = d_i^{(k)} = 0, b_i^{(k)} = 1$, unde $i \le 0$ sau $i \ge n+1$ şi $k \ge 0$ pentru ca $x_i = 0$, unde $i \le 0$ sau $i \ge n+1$. Atunci soluția finală poate fi exprimată prin

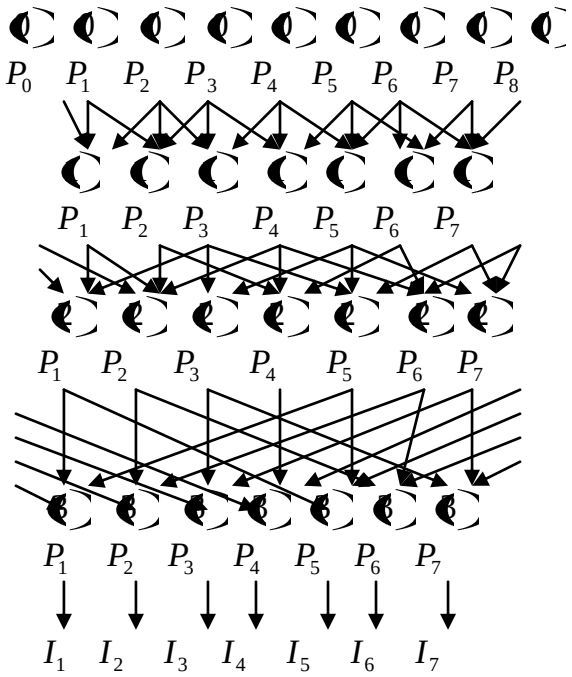
$x_i = d_i^{(k)} / b_i^{(k)}, k = \log_2 (n+1) \in \mathrm{N}$. Algoritmul este următorul:

2.Se calculează simultan toate valorile soluției :

a) se calculează coeficienții $p^s = (a_i^{(k)}, b_i^{(k)}, c_i^{(k)}, d_i^{(k)}), s = \overline{1,k}$

b) se determină $x_i = d_i^{(k)} / b_i^{(k)}$.

În figura urmatoare se prezintă algoritmul, în cazul unui sistem tridiagonal de dimensiune 7.




Determinarea soluției unui sistem tridiagonal de dimensiune 7 cu un nuăr minim de paşi paraleli

**REDUCEREA DIMENSIUNII PROBLEMEI**

O altă metodă adecvată execuției paralele a relațiilor iterative este descompunerea în subprobleme. Fie, de exemplu:

$$x^{(k+1)} = Ax^{(k)} + c,$$

unde $A$ este o matrice $n \times n$, iar $c$ este un vector de dimensiune $n$.Componentele vectorului $x^{(k+1)}$ pot fi calculate în parallel cu ajutorul mai multor procesoare.Dacă luăm în considerare cazul a două procesoare şi $n$ număr par ,vectorul $x^{(k+1)}$ este descompus

în două segmente, $x_1^{(k+1)}$ şi $x_2^{(k+1)}$, cu câte $n/2$ componente fiecare,ce se actualizează conform relației recursive:

$$\begin{bmatrix} x_1^{(k+1)} \\ x_2^{(k+1)} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} x_1^{(k)} \\ x_2^{(k)} \end{bmatrix} + \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$$

La fiecare pas,câte un processor actualizează jumătate din componente ,iar următoarea iterație este realizată numai după ce ambele procesoare au terminat actualizarea.

## SISTEME LINIARE CU MATRICI DENSE

Fie sistemul

$$Ax = b,$$

unde $A$ este o matrice densă $n \times n$, iar $x$ şi $b$ sunt vectori cu $n$ componente.Ne interesează posibilitățile de paralelizare ale metodelor de rezolvare cunoscute, cât şi abordarea unor metode specifice execuției cu mai multe procesoare.Vom avea în vedere atât metode directe, cât şi metode iterative.

## METODE DIRECTE

### Metoda Gauss paralelă

Metoda Gauss ( metoda eliminării parțiale ) este eficientă într-o execuție serială, dar poate fi executată şi cu ajutorul mai multor procesoare, urmărind, la fiecare pas, operațiile care pot fi executate independent unele de altele. Inițial avem:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n \end{cases}$$

În prima etapă,presupunând $a_{11} \neq 0$, toate cele $n$ procesoare vor lucra simultan la eliminarea necunoscutei $x_1$ din ecuațiile $2,3,\ldots,n$.După finalizarea primei etape, se obține sistemul echivalent

$$\begin{cases} x_1 + a_{12}^{(1)}x_2 + \ldots + a_{1n}^{(1)}x_n = b_1^{(1)} \\ \quad a_{22}^{(1)}x_2 + \ldots + a_{2n}^{(1)}x_n = b_2^{(1)} \\ \quad \vdots \\ \quad a_{n2}^{(1)}x_2 + \ldots + a_{nn}^{(1)}x_n = b_n^{(1)} \end{cases}$$

În etapa a doua se va elimina necunoscuta $x_2$ din ecuațiile $3,4,\ldots,n$ lucru care se va realiza cu ajutorul a $n-1$ procesoare.Mai departe, în etapa a treia ,vor fi active $n-2$ procesoare, ş.a.m.d. În final, se ajunge la sistemul superior triunghiular:

$$\begin{cases} x_1 + a_{12}^{(0)} x_2 + \ldots + a_{1,n-1}^{(0)} x_{n-1} + a_{1n}^{(0)} x_n = b_1^{(0)} \\ \qquad x_2 + \ldots + a_{2,n-1}^{(1)} x_{n-1} + a_{2n}^{(1)} x_n = b_2^{(1)} \\ \qquad\qquad \ldots \\ \qquad\qquad\qquad x_{n-1} + a_{n-1}^{(n-2)} x_n = b_{n-1}^{(n-2)} \\ \qquad\qquad\qquad\qquad a_{nn}^{(n-1)} x_n = b_n^{(n-1)} \end{cases}$$

Rezolvarea acestui sistem se va afce prin substituție regresivă, începând cu ultima ecuație.Dacă la inceput un singur processor va fi activ şi va executa operația de împărțire

$$x_n = b_n^{(n-1)} / a_{nn}^{(n-1)},$$

în continuare vor lucra 2 procesoare la determinarea valorii $x_{n-1}$, trei pentru calcularea valorii lui $x_{n-2}$, ş.a.m.d,utilizând tehnica dublării recursive pentru obținerea valorilor.Cu alte cuvinte, procesoarele se reactivează unul câte unul.

În concluzie, putem afirma că metoda Gauss se poate paraleliza, dar algoritmul Gauss parallel nu este sufficient de performant, din cauza neutilizării procesoarelor la capacitatea maximă.

**Metoda Gauss-Jordan paralelă**

Meotda Gauss-Jordan ( sau metoda eliminării totale ) poate fi la rândul său paralelizată.Astfel,considerând un calculator parallel cu $n$ procesoare,după o etapă, sistemul devine:

$$\begin{cases} x_1 + a_{12}^{(0)} x_2 + \ldots + a_{1n}^{(0)} x_n = b_1^{(0)} \\ \quad a_{22}^{(1)} x_2 + \ldots + a_{2n}^{(1)} x_n = b_2^{(1)} \\ \quad \vdots \\ \quad a_{n2}^{(1)} x_2 + \ldots + a_{nn}^{1} x_n = b_n^{(1)} \end{cases}$$

unde

$$a_{1j}^{(1)} = a_{1j} / a_{11}, \quad j = \overline{2,n}, \quad b_1^{(1)} = b_1 / a_{11}$$

$$a_{ij}^{(1)} = a_{ij} - a_{1j}^{(1)} a_{i1}, \quad b_i^{(1)} = b_i - b_1^{(1)} a_{i1}, \quad i, j = \overline{2,n},$$

calculele fiind făcute în felul următor: un procesor calculează valorile $a_{12}^{(1)}, a_{13}^{(1)}, \ldots, a_{1n}^{(1)}, b_1^{(1)}$ în această ordine. În momentul în care $a_{12}^{(1)}$ a fost calculat, restul de $n-1$ procesoare vor calcula elementele $a_{22}^{(1)}, \ldots, a_{n2}^{(1)}$, trecând apoi mai departe la calcularea lui $a_{23}^{(1)}, \ldots, a_{n3}^{(1)}$, ş.a.m.d., până la $b_2^{(1)}, \ldots, b_n^{(1)}$. Se presupune că în acest

moment procesorul unu a terminat cu prima linie de împărțiri, şi începe împărțirile necesare eliminării necunoscutei $x_2$: $a_{23}^{(2)} = a_{23}^{(1)} / a_{22}^{(1)}$. În momentul în care $a_{23}^{(2)}$ a fost calculat, acest procesor trece la următoarea împărțire: $a_{24}^{(2)} = a_{24}^{(1)} / a_{22}^{(1)}$, etc., iar restul $n-1$ procesoare vor calcula valorile pe coloane, deasupra şi dedesubtul elementului de pe diagonala principală.

Se continuă acest proces până când sistemul devine:

$$\begin{cases} x_1 & & & = b_1^{(n)} \\ & x_2 & & = b_2^{(n)} \\ & & \vdots & \\ & & x_n & = b_n^{(n)} \end{cases}$$

deci ajunge la forma diagonală care indică direct soluția.

După cum se observă, în cazul algoritmului paralel al eliminării totale, toate procesoarele sunt active, până la obținerea soluției. Observația care trebuie totuşi făcută este aceea că, pe tot parcursul execuției, un procesor execută numai operații de împărțire, în timp ce celelalte $n-1$ execută o adunare şi o înmulțire. Pentru ca să nu avem perturbări în execuția algoritmului, trebuie sincronizați timpii de execuție ai unei împărțiri cu cei ai unei adunări plus o înmulțire.

**Metoda factorizarii *WZ***

Factorizarea *WZ* este o variantă a metodei eliminării utilizată în implementarea pralelă. Ideea este aceea de a descompune matricea *A* într-un produs de matrice *WZ*, astfel încât calculele ulterioare să poată fi executate simultan, de către mai multe procesoare.

Fie sistemul de ecuații

$Ax = b$

şi $A = WZ$, unde

$$W = \begin{bmatrix} 1 & & & & 0 \\ w_{21} & 1 & 0 & 0 & w_{2n} \\ \vdots & & 1 & & \vdots \\ w_{n-1,1} & 0 & 0 & 1 & w_{n-1,n} \\ 0 & & & & 0 \end{bmatrix} \quad \text{şi} \quad Z = \begin{bmatrix} z_{11} & & \cdots & & z_{1n} \\ 0 & z_{22} & \cdots & z_{2,n-1} & 0 \\ \vdots & 0 & z_{ii} & 0 & \vdots \\ 0 & & & & 0 \\ z_{n1} & & \cdots & & \end{bmatrix}$$

adică

$$w_{ij} = \begin{cases} 1, & \text{daca } i = j \\ 0, & \text{daca } i+1 \le j \le n-i+1 \text{ sau } n-i+1 \le j \le i-1 \\ \text{nenul,} & \text{altfel} \end{cases}$$

$$z_{ij} = \begin{cases} 0, & \text{daca } j+1 \le i \le n-j \text{ sau } n-j+2 \le i \le j-1 \\ \text{nenul,} & \text{altfel} \end{cases}$$

Elementele nenule ale matricilor *W* şi *Z* se calculează astfel:

a) Prima şi ultima linie a lui *Z* sunt identice cu cele ale lui *A*

$z_{1j} = a_{1j}$ şi $z_{nj} = a_{nj}$, $j = \overline{1,n}$

b) Pentru prima şi ultima coloana din *W*, se rezolvă $n-2$ sisteme de 2 ecuaţii cu 2 necunoscute de forma:

$$\begin{cases} z_{11}w_{i1} + z_{n1}w_{in} = a_{i1} \\ z_{1n}w_{i1} + z_{nn}w_{in} = a_{in} \end{cases}, \quad i = \overline{2, n-1}$$

Pentru restul elementelor, pentru $k = 1, \ldots, \left[\frac{1}{2}(n+1)\right]$, calculele sunt următoarele:

$$z_{kl}w_{ik} + z_{n-k+1,l}w_{i,n-k+1} = a_{il} - \left(\sum_{s=1}^{k-1} + \sum_{s=n-k+2}^{n}\right) w_{is}z_{sl}, \quad l = \overline{k, n-k+1}.$$

O dată calculate elementele matricilor *W* şi *Z*, vom rezolva sistemul iniţial astfel. Rescriem

$Ax = b$

în forma

$WZx = b$

şi vom rezolva, mai întâi, sistemul

$Wp = b$,

determinând cele $n$ componente ale vectorului $p$, iar apoi sistemul

$Zx = p$

pentru a obţine soluţia finală.

Vom avea, deci:

$$\begin{bmatrix} 1 & & & & 0 \\ w_{21} & 1 & 0 & 0 & w_{2n} \\ \vdots & & 1 & & \vdots \\ w_{n-1,1} & 0 & 0 & 1 & w_{n-1,n} \\ 0 & & & & 1 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_{n-1} \\ p_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix}$$

La început se determină $p_1$ şi $p_n$, la pasul următor $p_2$ şi $p_{n-1}$, ş.a.m.d. La pasul $i$ vom avea

$p_i = b_i^{(i)}, \quad p_{n-i+1} = b_{n-i+1}^{(i)}$,

## METODE ITERATIVE

Literatura de specialitate prezenta metode iterative de rezolvare a sistemelor de ecuaţii liniare. In capitolul de faţă vom încerca să dăm o variantă paralelă posibilă a unor metode şi vom prezenta şi alte abordări specifice unei implementări paralele.

### Metoda Jacobi paralelă

Presupunând că avem un sistem cu *n* procesore, fiecare poate calcula o componentă a soluției după formulele:

$$x_i = \frac{1}{a_{ii}}\left( b_i - \sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}x_j \right), i = \overline{1,n}.$$

S-a văzut că această metodă se bazează pe o descompunere a matricei *A* într-o matrice diagonală *D*, una strict inferior triunghiulară *L* şi alta strict triunghiulară *U*, deci

*A=D-L-U*.

Pentru o execuție paralelă eficientă, se consideră descompunerea dublu-diagonlă a matricei *A*:

*A=X-W-Z*,

unde

$$X = \begin{bmatrix} * & 0 & * \\ & * & * & \\ 0 & * & 0 \\ & * & * & \\ * & 0 & * \end{bmatrix} \qquad -W = \begin{bmatrix} 0 \ldots & \ldots 0 \ldots & \ldots 0 \\ * & & & & * \\ * & * & & & * & * \\ * & * & * & 0 & * & * & * \\ * & * & & & * & * \\ * & & & & & * \\ 0 \ldots & \ldots 0 \ldots & \ldots 0 \end{bmatrix}$$

$$-Z = \begin{bmatrix} 0 & * & * & * & * & * & 0 \\ & & * & * & * & & \\ & & & * & & & \\ \vdots & & & 0 & & & \vdots \\ & & & * & & & \\ & & * & * & * & & \\ 0 & * & * & * & * & * & 0 \end{bmatrix}$$

cu alte cuvinte

$$x_{ij} = \begin{cases} 0, & i \neq j \ si \ i \neq n-j+1 \\ a_{ij}, & in \ caz \ contrar \end{cases}$$

$$-w_{ij} = \begin{cases} 0, & i \leq j \leq n-i+1 \ si \ n-i+1 \leq j \leq i \\ a_{ij}, & in \ caz \ contrar \end{cases}$$

$$-z_{ij} = \begin{cases} 0, & j \le i \le n-j+1 \; si \; n-j+1 \le i \le j \\ a_{ij}, & in \; caz \; contrar \end{cases}$$

In acest caz, metoda Jacobi este cunoscută sub numele de "metoda Jacobi dublu-diagonală" şi generează iterația:

$$Xx^{(p+1)} = (W+Z)x^{(p)} + b.$$

Schema poate fi privită ca o metodă iterativă de blocuri, de dimensiune 2X2, calculele fiind executate simultan de către mai multe procesoare.

**Metoda Gauss-Seidel paralelă**

In cazul metodei Gauss-Seidel, datorită utilizării succesive a componentelor necunoscute deja actualizate, procesoarele pot lucra simultan la calcularea primei componente a unei iterații, apoi lucrează toate la calcularea primei componente a unei iterații, apoi lucrează toate la calcularea celei de-a doua componente, ş.a.m.d. Tehnica utilizată este cea a dublării recursive.

Metoda prezentată anterior ia în considerare numerotarea lexicografică a necunoscutelor: $x_1, x_2, \ldots, x_n$. Caracteristica metodei de a utiliza în calculul valorii unei necunoscute valorile unor necunoscute deja calculate, permite şi o altfel de implementare paralelă, dacă utilizăm numerotarea "roşu-negru" (sau "alb-negru", "tablă-şah", etc.) pentru componente. Ideea este aceea că, pe baza formulelor prezentate în paragraful precedent, actualizăm simultan componentele "negre" (adică pe cele cu indicii impari): $x_1, x_3, \ldots$ şi apoi, simultan, componentele "roşii" (adică pe cele cu indicii pari): $x_2, x_4, \ldots$, care vor folosi deja valorile calculate la pasul anterior. Putem vizualiza procedeul cu ajutorul unei linii din tabla de şah.

Execuția "tabla-de-şah" a metodei Gauss-Seidel

**Metoda paralelă a relaxării (SOR paralel)**

Analog cu metoda Jacobi dublu-diagonală, cu ajutorul descompunerii matricei *A*:

*A=X-W-Z*

se poate introduce metoda SOR dublu-diagonală, conform căreia o iterație se descrie astfel:

$$(X-\omega W)x^{(p+1)}=[\omega Z+(1-\omega)X]x^{(p)}+\omega b.$$

unde ω este factorul de relaxare al metodei SOR.

Se obține astfel o metodă bloc-iterativă, convenabilă unei execuții cu mai multe procesoare.

## METODE NUMERICE PARALELE DE REZOLVARE A ECUATIILOR NELINIARE

Pentru rezolvarea ecuațiilor neliniare pe R există diferite metode seriale. Vom încerca în acest capitol să abordăm unele dintre ele din punct de vedere al implementării lor paralele.

### Metoda paralelă a înjumătățirii intervalului (metoda bisecției)

Fie ecuația neliniară

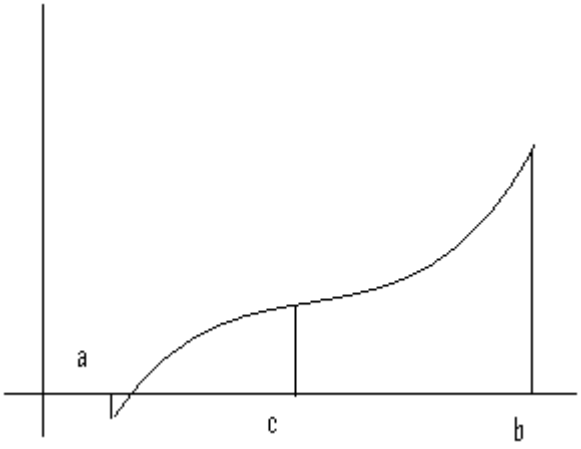
(1) $f(x)=0,$

unde $f:[a,b]\rightarrow R$ este o funcție continuă.

Dacă $f(a)f(b)<0$, atunci ecuația dată are cel puțin o soluție reală în intervalul (a, b). Se cere determinarea aproximativă, cu o eroare dată, a soluțiilor reale ale ecuației (1).

Din punct de vedere al unei execuții cu mai multe procesoare, putem considera următoarele cazuri:

Cazul I. Ecuația (1) are o singură soluție reală în intervalul $[a,b]$.



Situația unei rădăcini reale în intervalul [*a,b*]

In această situație, metoda înjumătățirii intervalului constă în definirea a trei şiruri $\{a_n\}$, $\{b_n\}$, $\{c_n\}$, astfel:

Inițializare: $a_0=a$, $b_0=b$, $c_0=(a+b)/2$.

Fiind constuiți termenii $a_{n-1}$, $b_{n-1}$ şi $c_{n-1}$ definim, pentru $n\geq 1$

$a_n=c_{n-1}$ şi $b_n=b_{n-1}$ dacă $f(a_{n-1})f(c_{n-1}) >0$

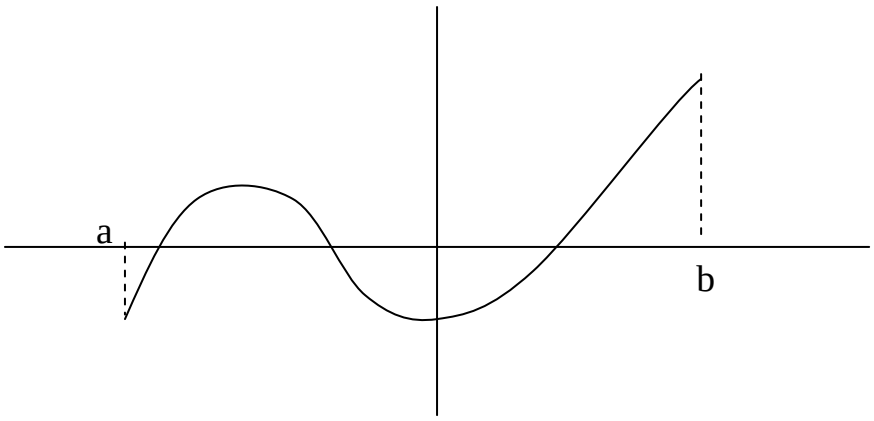
$a_n=a_{n-1}$ şi $b_n=c_{n-1}$ dacă $f(a_{n-1})f(c_{n-1}) >0$

$c_n=(a_n+b_n)/2$

Dacă $f(c_{n-1})=0$ atunci $a_m= a_{n-1}$, $b_m=b_{n-1}$, $c_m=c_{n-1}$, $\forall$ $m\geq n$.

Determinarea elementelor celor trei şiruri poate fi făcută simultan pe un sistem paralel, fiecare şir fiind calculate de către un processor. Un al patrulea poate fi folosit la calcularea valorii funcției în diferite puncte şi la obținerea rezultatului.

**Cazul II.** Ecuația (1) are mai multe rădăcini reale în intervalul [*a,b*].




Situația în care există mai multe rădăcini reale în [*a,b*]

In această situație, problema se descompune în mai multe subprobleme, după cum intervalul inițial [*a,b*] se descompune în subintervale care izolează câte o singură rădăcină reală.

Lucrând cu mai multe procesoare, fiecare poate determina soluția aproximativă corespunzătoare subproblemei repartizată.

**Metoda paralelă a lui Newton (metoda tangentei)**

Iterația Newton pentru rezolvarea unei ecuații neliniare

$f(x) = 0$

este

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \quad x_0 \text{ dat.} \tag{2}$$

Si în acest caz putem aborda paralelizarea metodei în două feluri:

a) O manieră naturală ar diviza calculul în două procese:

$f_1(x_k)=f(x_k)$ şi $f_2(x_k)=f(x_k)$

Cu alte cuvinte, un processor ar evalua tot timpul valoarea derivatei pe punct, iar celălalt valoarea funcției pe punct. Un al treilea processor ar determina noua componentă $x_{k+1}$. Această posibilitate de execuție paralelă nu asigură, totuşi, o eficiență prea ridicată, deoarece timpii de evaluare a celor două funcții sunt, în general diferiți.

b) In cazul în care $x_k \neq 0$ , pentru k=0,1,…, recurența (2) se poate scrie astfel:

$$\begin{bmatrix} x_{k+1} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & -\alpha_k \\ x_k^{-1} & 0 \end{bmatrix} \begin{bmatrix} x_k \\ 1 \end{bmatrix}, \text{ cu } \alpha_k = \frac{f(x_k)}{f'(x_k)},$$

respectiv

$$\begin{bmatrix} x_n \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & -\alpha_{n-1} \\ x_{n-1}^{-1} & 0 \end{bmatrix} \begin{bmatrix} 1 & -\alpha_{n-2} \\ x_{n-2}^{-1} & 0 \end{bmatrix} \cdots \begin{bmatrix} 1 & -\alpha_0 \\ x_0^{-1} & 0 \end{bmatrix} \begin{bmatrix} x_0 \\ 1 \end{bmatrix}$$

După evaluarea valorilor ak, produsul matriceal se poate executa cu mai multe procesoare, conform tehnicii dublării recursive.

**Metoda paralelă a secantei (metoda coardei)**

Fie ecuația

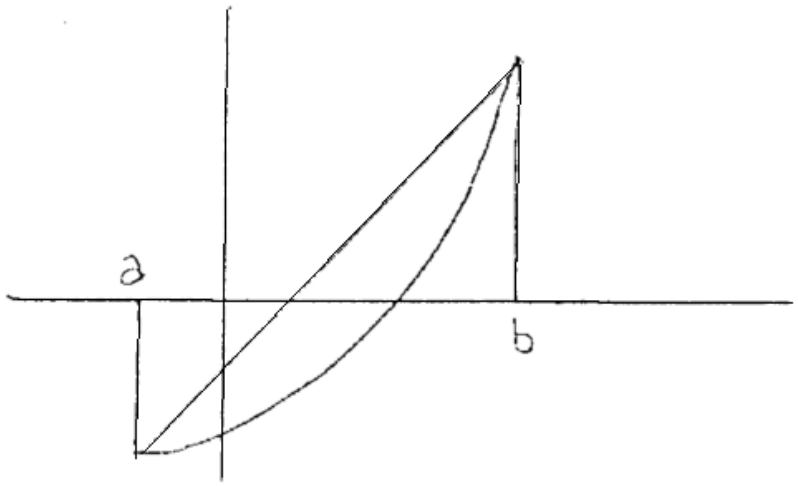
*f(x)=0,*

unde *f:[a,b]*→ R este o funcție continuă.

Fără restricție de generalitate, putem presupune

*f(a)<0 şi f(b)>0,*

ca în figura ce urmeaza




Ca si la metoda bisectiei, din punct de vedere al abordarii paralele, avem mai multe cazuri:

**Cazul I.** Situatia in care in intervalul [a,b] exista o singura radacina reala. Tinand cont de expresia iteratiei ce caracterizeaza metoda secantei, si lucrand simultan cu trei procesoare, se pot construe, simultan, sirurile $\{a_n\}$, $\{b_n\}$ si $\{c_n\}$, astfel:

Initializare: $a_0 = a, b_0 = b, c_0 \frac{a_0 f(b_0) - b_0 f(a_0)}{f(b_0) - f(b_0)}$

Fiind construite elementele $a_{n-1}, b_{n-1}, c_{n-1}$, pentru n≥1 definim:

$a_n = c_{n-1}, b_n = b_{n-1}, daca f(c_{n-1}) < 0$

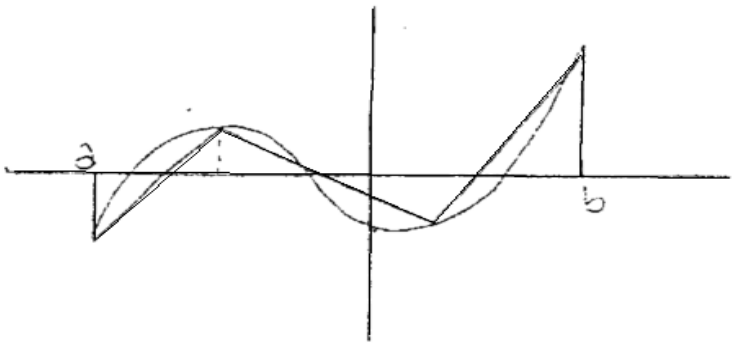
$a_n = a_{n-1}, b_n = c_{n-1}, daca f(c\, c_{n-1}) > 0$

$$c_n = \frac{a_n f(b_n) - b_n f(a_n)}{f(b_n) - f(a_n)}$$

Daca $f(c_{n-1}) = 0$ atunci $a_m = a_{n-1}, b_m = b_{n-1}, c_m = c_{n-1}, \forall m \geq n$

Se demonstreaza ca sirul $\{c_n\}$ converge catre solutia exacta a ecuatiei f(x)=0.

**Cazul II.** Situatia in care ecuatia are mai multe radacini reale in intervalul [a,b], ca in figura urmatoare.




In acest caz, se separa radacinile reale, astfel incat sa ne situam in ipotezele aplicarii metodei secantei, si fiecare processor va executa metoda pe cate un subinterval.

**Metoda tangenta-secanta**

Fiind data ecuatia

$$f(x)=0$$

cu f:[a,b] -> R continua pe [a,b], si presupunand existenta unei singure radacini reale a ecuatiei in (a,b), metoda tangenta-secanta presupune construirea unui sir de intervale approximate, de forma:

$$(x_1,^t, x_1^s) \supset (x_2,^t, x_2^s) \supset \ldots \supset (x_n,^t, x_n^s) \supset \ldots.$$

unde sirul $\{ x_n^t \}$ se genereaza cu metoda tangentei, iar sirul $\{ x_n^s \}$ cu metoda secantei.

Procedeul se opreste atunci cand

$$| x_n^t - x_n^s | < \varepsilon,$$

unde ε este o eroare prestabilita. In acest moment, oricare dintre cele doua aproximante aproximeaza solutia reala exacta a ecuatiei cu precizia dorita. Metoda se poate paraleliza usor, daca vom considera ca un processor genereaza sirul $\{ x_n^t \}$, simultan cu altul care genereaza sirul $\{ x_t^s \}$.

**CUADRATURI NUMERICE PARALELE**

Integrarea numerica a functiilor ocupa un loc important in analiza matematica, deci este de mare interes o abordare a acestei probleme si din punctual de vedere al calculului parallel.

Vom face referire doar la cuadraturile de tip interpolator, dar acest lucru nu reprezinta o restrictie, caci rationamente similare sunt valabile si pentru alte tipuri de formule de integrare numerica. Vom considera, in continuarem cuadraturile de tip interpolator obtinute pe noduri echidistante, in cazul in care calculul integralei se face repetat, pe subintervale in care se divide intervalul initial. Cu alte cuvinte ne vom referi la formula Newton-Cotes repetate.

**Formula repetata a trapezului**

(1)

$$\int_a^b f(x)dx \cong \frac{b-a}{2a}[f(a) = f(b) + 2\sum_{k=1}^{n-1} f(x_k)]$$

Lucrand simultan cu mai multe procesoare, am putea calcula valoarea numerica a integralei astfel:

a) in forma (1), n-2 procesoare pot evalua $\sum_{k=1}^{n-1} f(x_k)$, cu ajutorul tehnicii dublarii recursive (vezi si paragraful "evaluarea expresiilor aritmetice"), iar altele doua, f(a) si f(b).
b) Daca scriem formula repetata a trapezului

$$\int_a^b f(x)dx = \sum_{k=1}^{n} I_k, \text{ cu } I_k = \int_{x_{k-1}}^{x_k} f(x)dx,$$

adica

(2)

$$\int_a^b f(x)dx \cong \sum_{k=1}^{n} \{\frac{x_k - x_{k-1}}{2}[f(x_{k-1}) + f(x_k)]\}$$

atunci, lucrand cu n procesoare, putem considera ca fiecare processor evalueaza valoarea numerica a integralei pe un interval $l_k$ si la sfarsit, tot prin metoda dublarii recursive, se evalueaza suma finala si se obtine rezultatul dorit.

## CÂTEVA NOȚIUNI PRIVIND PARALELISMUL ÎN PROCESAREA IMAGINILOR

În sistemele grafice, rezultatele introducerii paralelismului sunt evidente. Îmbunătățirile aduse în asemenea sisteme sunt:

• mai multe coprocesoare care împart cu unitatea centrală de procesare acelaşi bus;

• mai multe procesoare de display cu memorie proprie;

• procesoare integrate care conțin suport hardware intern pentru mai multe operații grafice simultan.

De exemplu, procesorul Intel i860 este un microprocesor pe un singur cip cu suport integrat pentru grafică tridimensională. Instrucțiunile sale operează în paralel pe atâția pixeli câți pot fi împachetați într-un cuvânt de 64 biți. Pentru aplicații care utilizează, de exemplu, 8 biți pentru reprezentarea informației asociate cu un pixel, sunt posibile 8 operații grafice simultane. Instrucțiunile grafice paralele permise includ următoarele:

1. calculul în paralel al interpolărilor liniare;
2. calculul în paralel al comparărilor asociate cu algoritmul z-buffer de vizibilitate;
3. actualizarea, în paralel, condiționată, a pixelilor.

Etapele principale în transpunerea pe ecran a imaginilor tridimensionale a unui corp sunt următoarele:

1. traversarea bazei de date care descrie obiectul (prin primitive grafice ca puncte sau linii);

2. transformarea obiectului din sistemul de coordonate ale obiectului în sistemul de coordonate a lumii înconjurătoare;

3. acceptarea sau respingerea primitivelor pentru încadrarea în volumul de observare;

4. simularea unui model de iluminare a corpului;

5. trecerea obiectului în sistemul de coordonate normalizate;

6. aplicarea transformării perspective asupra obiectului;

7. transformarea primitivelor în valori per pixeli.

În figura de mai jos se poate observa distribuirea pixelilor unui ecran la elementele de procesare: în blocuri contigue (a) şi în zone fragmentate (b).

In fiecare din aceste faze, modalitatea de paralelizare a proceselor este diferită. Se dau următoarele exemple:

1. împărțirea bazei de date între procesoare;
2. există mai multe variante:

(a) componentele individuale ale fiecărui punct al obiectului pot fi repartizate pe procesoare diferite care efectuează anumite transformări pe baza unor matrici;

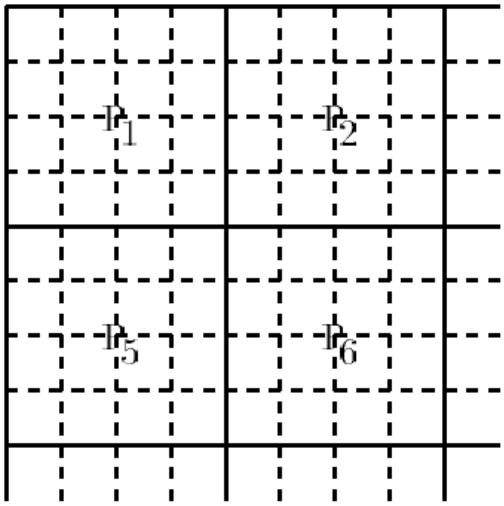
(b) dacă primitivele sunt uniforme, ca de exemplu un set de triunghiuri, toate vârfurile triunghiurilor pot fi transformate simultan de către trei procesoare distincte;

3. se poate asocia fiecărui plan ce delimitează volumul de observare un procesor care efectuează testele de acceptare/respingere referitoare la planul corespunzător;
4. în iluminarea obiectelor se recomandă utilizarea unui procesor specializat (hardware), în virgulă mobilă, care calculează culoarea unui punct pornind de la vectorul incident al luminii şi informațiile referitoare la suprafața pe care punctul se află;
5. se desfăşoară analog cu (3);
6. calculele se pretează la utilizarea unui procesor pipeline;
7. se partiționează pixelii ecranului între mai multe elemente de procesoare.

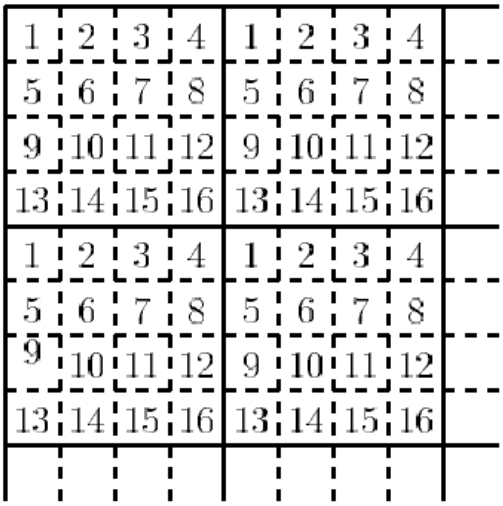
Există două strategii consacrate de partiționare a ecranului, după cum se poate observa în figura de mai jos:

• în blocuri contigue (a) când primitivele sunt procesate numai în acele porțiuni în care sunt vizibile (determinate geometric). Pot apărea probleme de eficiență prin încărcarea diferită a elementelor de procesare (porțiuni care nu conțin primitive ale obiectului față de porțiuni care conțin un număr mare de primitive ale obiectului);

• în zone fragmentate (b) prin care efortul de calcul este echilibrat (cea mai răspândită formă de partiționare).


1
2
5
6

(a)


1 2 3 4 1 2 3 4
5 6 7 8 5 6 7 8
9 10 11 12 9 10 11 12
13 14 15 16 13 14 15 16
1 2 3 4 1 2 3 4
5 6 7 8 5 6 7 8
9 10 11 12 9 10 11 12
13 14 15 16 13 14 15 16

(b)

# Capitolul 5

# SISTEME DISTRIBUITE

## DEFINIREA SISTEMELOR DISTRIBUITE

În literatura de specialitate pot fi regăsite numeroase şi diferite definiţii ale sistemelor informatice distribuite, nici una nefiind larg acceptată în rândul specialiştilor. Acest fapt poate fi explicat prin multitudinea tipurilor de sisteme distribuite implementate astăzi, într-o mare diversitate arhitecturală, tehnologică sau de altă natură, ceea ce face dificilă identificarea unor caracteristici generale comune întregii varietăţi de sisteme dezvoltate. Din acest motiv, misiunea proiectanţilor este dificilă datorită complexităţii sistemelor distribuite, ei trebuind să identifice toate combinaţiile fezabile din care să o aleagă pe cea optimă.

Pentru a desprinde elementele definitorii ale sistemelor distribuite, vom prezenta două definiţii simple, completate prin enumerarea caracteristicilor lor esenţiale şi câteva exemple.

O primă definiţie consideră sistemele distribuite ca "o colecţie de calculatoare independente care apar utilizatorilor acestora ca un singur sistem coerent". Această definiţie evidenţiază două aspecte esenţiale. Primul priveşte hardware-ul: **calculatoarele sunt autonome**. Cel de-al doilea vizează software-ul: **utilizatorii au impresia că lucrează cu un singur sistem**.

O altă definiţie descrie sistemele distribuite ca acele sisteme "în care componentele hardware şi software localizate într-o reţea comunică şi îşi coordonează acţiunile lor doar prin transmiterea de mesaje". Calculatoarele conectate prin intermediul reţelei pot fi separate de orice distanţă, putându-se afla pe continente diferite sau în aceeaşi clădire. Însă, cel mai important aspect care trebuie reţinut este faptul că **realizarea comunicării între componentele unui sistem distribuit se face numai prin intermediul schimbului de mesaje**.

Dincolo de aceste definiţii, problematica sistemelor distribuite poate fi clarificată prin prezentarea caracteristicilor lor esenţiale. Pe scurt, acestea sunt:

• diferenţele dintre variatele tipuri de calculatoare şi modul în care ele comunică sunt ascunse (transparente) pentru utilizator, la fel ca şi organizarea internă a sistemului distribuit;

• utilizatorii şi aplicaţiile pot interacţiona cu un sistem distribuit într-o manieră uniformă şi consistentă, indiferent de locul şi momentul în care are loc interacţiunea; execuţia concurentă a programelor reprezintă regula într-un sistem distribuit. Doi utilizatori îşi pot realiza sarcinile lor de lucru pe propriile calculatoare prin partajarea unor resurse, precum paginile web sau fişiere, atunci când este necesar;

• sistemele distribuite trebuie să fie scalabile adică, să poată fi uşor extinse. Această caracteristică este o consecinţă directă a independenţei calculatoarelor din sistem, dar şi a faptului că pentru utilizator organizarea internă este transparentă;

• un sistem distribuit trebuie să asigure independenţa faţă de eventualele căderi sau disfuncţionalităţi ale unor calculatoare sau aplicaţii din sistem, el trebuind să fie în continuare disponibil utilizatorilor. Este responsabilitatea proiectanţilor de a prevedea consecinţele eventualelor disfuncţionalităţi.

Conceptul de sistem distribuit este aplicat unei mari varietăţi de configuraţii şi aplicaţii. Totuşi, pornind de la cele două componente principale ale unui software –

prelucrările şi datele, pot fi identificate două tipuri de bază de sisteme distribuite: **sisteme cu prelucrări distribuite** şi **sisteme cu date distribuite**. Există mai multe variante de configurare a unui mediu cu prelucrări distribuite: aplicaţiile pot fi stocate într-o singură locaţie şi accesate de către oricare procesor conectat în sistem; o aplicaţie poate fi replicată pe mai multe locaţii din reţea; diferite aplicaţii pot fi rezidente pe diferite locaţii din reţea, însă ele sunt accesibile tuturor utilizatorilor din reţea. Distribuirea datelor presupune proiectarea unei baze de date distribuite în care datele sunt fragmentate şi dispersate pe diferite locaţii din reţea sau ele sunt replicate pe mai multe noduri din reţea în vederea uşurării accesului la date. O altă configuraţie de sistem distribuit poate rezulta prin combinarea celor două tipuri de bază. Oricum, asupra arhitecturilor distribuite şi asupra modului de distribuire a prelucrărilor şi a datelor vom reveni în capitolele următoare.

Pentru o mai bună înţelegere a sistemelor distribuite vom prezenta în continuare câteva exemple.

**Exemplul 1**. Să considerăm reţeaua unei companii care, în afara staţiilor de lucru ale utilizatorilor, conţine un grup de procesoare situate eventual într-o sală specială şi care nu sunt atribuite unui utilizator anume, dar care pot fi alocate dinamic în funcţie de nevoi. Un astfel de sistem ar putea avea un singur sistem de fişiere în care toate fişierele să fie accesibile de pe toate nodurile în acelaşi fel şi utilizând aceeaşi cale. Mai departe, atunci când un utilizator introduce o comandă, sistemul ar putea să caute cel mai bun loc pentru a o executa, fie pe staţia utilizatorului respectiv, fie pe o staţie mai liberă a altui utilizator, fie pe unul din procesoarele nealocate unui anumit utilizator. Atât timp cât sistemul apare utilizatorului ca un sistem cu un singur procesor care partajează timpul de acces la resursele sale, el este un sistem distribuit.

**Exemplul 2.** Luăm în considerare un sistem informatic pentru gestiunea comenzilor clienţilor bazat pe tehnologia workflow. Un astfel de sistem poate fi utilizat de oameni din diferite departamente şi, eventual, dispersaţi în teritoriu. De exemplu, angajaţii din departamentul de vânzări pot fi împrăştiaţi la nivelul unei regiuni sau a întregii ţări. Comenzile pot fi introduse în baza de date prin intermediul unui laptop conectat la sistem prin intermediul reţelei telefonice sau chiar al unui telefon celular. Odată introdusă, comanda este înaintată departamentului de producţie pentru întocmirea comenzii de livrare. Comanda de livrare este apoi trimisă la depozit şi la departamentul financiar pentru generarea facturii ce va fi ulterior înregistrată în contabilitate. Utilizatorii nu au habar de circuitul fizic al comenzii prin sistem; ei percep sistemul ca şi cum ar exista o bază de date centralizată.

Fără îndoială, Internetul reprezintă cel mai mare sistem distribuit din lume. El reprezintă cea mai vastă colecţie de calculatoare de tipuri diferite, interconectate între ele; programele care rulează pe calculatoarele conectate interacţionează prin schimbul de mesaje; modul de organizare a Internetului, localizarea resurselor hard şi soft (de exemplu paginile Web) sunt transparente utilizatorului; utilizatorii pot utiliza serviciile disponibile în acelaşi mod (precum Web-ul, transferul de fişiere, poşta electronică), indiferent de momentul şi locul în care s-ar afla; setul de servicii oferite poate fi extins prin adăugarea unui server sau a unui nou tip de serviciu; mai mulţi utilizatori pot accesa simultan aceeaşi pagină web; toleranţa la disfuncţionalităţi a constituit fundamentul Internetului.

## AVANTAJELE ŞI DEZAVANTAJELE SISTEMELOR DISTRIBUITE

Numeroasele progrese din domeniul tehnologiei informaţionale au creat premisele dezvoltării sistemelor distribuite. Însă, numai simplu fapt că este disponibilă nu justifică utilizarea unei tehnologii informaţionale. Probabil că motivaţia principală pentru utilizarea sistemelor distribuite o reprezintă dorinţa principală a utilizatorilor de a partaja resursele. Noţiunea de **resursă** este una abstractă, folosită pentru a descrie mulţimea lucrurilor care pot fi partajate într-o reţea de calculatoare. Ea face referire la componentele hardware, precum discurile şi imprimantele, dar şi la cele software, precum fişierele, bazele de date, obiectele de toate tipurile.

Partajarea resurselor nu este singurul avantaj al sistemelor distribuite, numărul lor mare făcând dificilă prezentarea lor exhaustivă. Mai mult, ele diferă de la o tehnică la alta (de exemplu distribuirea datelor şi distribuirea prelucrărilor). De aceea, asupra avantajelor sistemelor distribuite vom reveni în capitolele următoare, atunci când vom aborda mai detaliat diferitele probleme ale dezvoltării sistemelor distribuite.

Principalele avantajele generale care pot fi obţinute prin implementarea sistemelor distribuite sunt enumerate în tabelul de mai jos. Obţinerea acestor avantaje reprezintă o sarcină dificilă, deoarece ele depind de numeroşi factori.

| **Avantaje** | **Dezavantaje** |
|---|---|
| Creşterea disponibilităţii şi siguranţei resurselor | Complexitatea sistemelor distribuite |
| Reducerea costurilor de comunicaţie | Sporirea dificultăţilor în controlul resurselor informaţionale |
| Flexibilitatea dezvoltării sistemelor – creştere incrementală | Probleme legate de asigurarea consistenţei datelor |
| Alinierea cu structura organizatorică a firmei | Sporirea dificultăţilor în testarea şi detectarea erorilor |
| Obţinerea unor timpi de răspuns mai buni | |
| Independenţa faţă de tehnologiile unui singur furnizor | |

În continuare vom comenta pe scurt câteva dintre avantajele şi dezavantajele prezentate în tabel, efectuând o comparaţie cu sistemele centralizate, deoarece ele reprezintă soluţia alternativă la sistemele distribuite.

**Flexibilitatea dezvoltării sistemelor distribuite** este dată de faptul că o firmă aflată în plină dezvoltare (extindere) are posibilitatea de a adăuga incremental noi resurse (hard şi soft) în sistem, respectiv achiziţionarea, instalarea şi conectarea lor pe măsură ce ele sunt necesare. Flexibilitatea sistemelor centralizate este limitată de inabilitatea lor de a asigura creşterea incrementală. Dezvoltarea sau extinderea activităţii firmei determină supraîncărcarea sistemului informaţional existent şi, implicit, necesitatea înlocuirii acestuia cu altul mai performant (în cazul sistemelor distribuite nu se pune problema înlocuirii acestuia ci a extinderii lui, conservându-se astfel investiţiile anterioare). Chiar dacă s-ar pune problema planificării extinderii viitoare a firmei în vederea dezvoltării unui sistem informatic corespunzător, soluţia unui sistem centralizat tot nu ar fi satisfăcătoare deoarece ea ar fi prea scumpă, atât timp cât o bună parte din capacitatea de stocare şi prelucrare a sistemului nu va fi utilizată decât ulterior, pe măsura dezvoltării firmei, şi numai dacă previziunile se adeveresc.

Una dintre motivațiile importante ale dezvoltării sistemelor distribuite este legată de **reflectarea transformărilor din mediile de afaceri**. Companiile multinaționale au sisteme informatice pentru fiecare țară în care desfășoară afaceri, necesitatea integrării lor fiind evidentă. De asemenea, reflectarea structurii organizatorice a întreprinderii în structura bazei de date reprezintă un avantaj important; multe organizații sunt distribuite cel puțin la nivel logic (în câteva subunități, departamente, grupuri de lucru etc.) dar, adesea, şi la nivel fizic (uzine, fabrici, ateliere etc.), iar distribuirea datelor conform organizării din firma respectivă permite fiecărei unități organizatorice să stocheze şi să gestioneze independent datele care privesc operațiunile sale. Pe de altă parte, dezvoltarea afacerilor electronice şi a afacerilor mobile va potența extinderea utilizării sistemelor distribuite. În fapt, dezvoltarea afacerilor electronice şi a celor mobile nu ar fi posibilă fără utilizarea tehnologiei sistemelor distribuite.

**Creşterea disponibilității resurselor** reprezintă un alt avantaj major al sistemelor distribuite. Apariția unei disfuncționalități într-un sistem centralizat (căderea serverului sau a liniei de comunicație) determină blocarea întregului sistem informațional până la remedierea problemei ivite. În schimb, sistemele distribuite sunt proiectate să funcționeze şi în condițiile apariției unor disfuncționalități, care va afecta numai o parte a sistemului. Celelalte resurse rămân disponibile, ele putând chiar prelua sarcinile părții de sistem afectate, situație în care utilizatorul nu va fi conştient de disfuncționalitatea apărută. Un studiu al International DARTS relevă că pierderile medii pe un minut de nefuncționare a sistemului în cazul aplicațiilor ERP, a aplicațiilor de gestiune a relațiilor cu furnizorii şi a aplicațiilor de comerț electronic sunt de 7900$, 6600$, respectiv 7800$. În cazul altor aplicații, nivelul pierderilor pe un minut de nefuncționare a sistemului ar putea fi mult mai mari.

Sistemele distribuite permit **reducerea costurilor de comunicație şi depăşirea limitelor mediilor de comunicație**. Într-un sistem distribuit, majoritatea prelucrărilor pot fi realizate local, iar datele de interes local pot fi stocate şi gestionate local, ceea ce determină reducerea drastică a traficului în rețea. Cea mai mare problemă cu care se poate confrunta o bază de date centralizată, atunci când ea este accesată de la distanță, este legată de eventualitatea blocajelor rețelei de comunicație; nici supraîncărcarea serverului de numeroasele accese de la distanță nu trebuie neglijată.

Sistemele distribuite oferă timpi de răspuns mai buni la cererile utilizatorilor. Sistemele centralizate păcătuiesc adesea prin oferirea unor timpi de răspuns nesatisfăcători utilizatorilor, datorită volumului mare de date ce trebuie transmise prin rețea. Cele două topologii cu baza de instalare cea mai mare – Token Ring pe 16 Mb şi Ethernet la 10 Mb – nu permit obținerea unor timpi de răspuns acceptabili atunci când procesele de prelucrare solicită un volum mare de date.

În afara avantajelor prezentate, implementarea sistemelor distribuite au asociate şi unele dezavantaje ce trebuie luate în considerare la dezvoltarea lor.

Poate cea mai importantă piedică în extinderea utilizării sistemelor distribuite o reprezintă dificultatea dezvoltării lor generate de enorma complexitate a acestor sisteme. Principalele surse ale complexității sunt: distribuirea datelor şi/sau replicarea lor, distribuirea prelucrărilor, asigurarea diferitelor forme de transparență, asigurarea consistenței datelor. Un sistem cu baze de date distribuite care trebuie să ascundă natura distribuită a datelor față de utilizatori este fără îndoială mai complex decât un sistem cu baze de date centralizate. Bazele de date replicate adaugă cel puțin un nivel suplimentar

de complexitate. Dacă sistemul nu este bine proiectat, atunci el va furniza un nivel de performanță, disponibilitate şi siguranță inacceptabile.

## OBIECTIVE GENERALE PRIVIND PROIECTAREA SISTEMELOR DISTRIBUITE

Dincolo de avantajele oferite, prin proiectarea unui sistem distribuit trebuie urmărită atingerea unor obiective care să permită obținerea avantajelor propuse. În continuare vom descrie succint câteva dintre obiectivele care trebuie realizate în dezvoltarea sistemelor distribuite.

**Eterogenitatea.** Un sistem distribuit trebuie să permită utilizatorilor accesarea serviciilor şi execuția programelor pe platforme eterogene. Eterogenitatea priveşte rețelele, calculatoarele, sistemele de operare, limbajele de programare (mediile de dezvoltare a aplicațiilor) şi implementările diferiților dezvoltatori de aplicații.

Internetul reuneşte diferite tipuri de rețele, însă eterogenitatea lor este ascunsă de faptul că toate calculatoarele conectate utilizează protocoalele Internetului pentru a comunica între ele. Aceeaşi soluție este utilizată şi pentru mascarea eterogenității sistemelor de operare, fiecare trebuind să aibă implementate protocoalele Internetului. Diferite tipuri de calculatoare pot utiliza metode diferite de reprezentare a datelor în memoria calculatorului (de exemplu pentru tipul de date Integer), aspect ce trebuie să fie transparent unor aplicații ce rulează pe tipuri diferite de calculatoare pentru a putea schimba mesaje între ele. În mod asemănător, limbajele de programare pot utiliza diferite reprezentări ale caracterelor şi structurilor de date, precum tablourile de date sau înregistrările, diferențe ce trebuie rezolvate dacă programele scrise în limbaje de programare diferite trebuie să comunice între ele. În sfârşit, miza cea mai importantă pentru proiectanți o reprezintă dezvoltarea de programe care să poată comunica cu programele scrise de alți proiectanți, scop în care ei trebuie să adopte standarde comune. Această problemă vizează în primul rând structura datelor din mesaje.

Rezolvarea eterogenității în sistemele distribuite este asigurată prin **componenta middleware**. Termenul middleware este aplicat unui nivel intermediar al software-ului şi are rolul de a ascunde eterogenitatea rețelelor, a echipamentelor, a sistemelor de operare şi a limbajelor de programare. Cele mai cunoscute middleware sunt CORBA, Java RMI şi DCOM. În plus, middleware-ul furnizează un model uniform ce trebuie adoptat de programatorii de aplicații distribuite. Astfel de modele sunt: invocarea obiectelor de la distanță, notificarea evenimentelor de la distanță, accesul SQL de la distanță şi prelucrarea tranzacțiilor distribuite. De exemplu standardul CORBA furnizează un model pentru invocarea obiectelor de la distanță, care permite unui obiect dintr-un program ce rulează pe un calculator să invoce (apeleze) o metodă a unui obiect dintr-un program ce rulează pe alt calculator. Implementarea ei ascunde faptul că mesajele sunt transmise prin rețea.

O problemă interesantă o reprezintă **programele mobile** (mobile cod). Ele fac referire la programele care pot fi trimise de pe un calculator pe altul şi să fie executate la destinație (este cazul applet-urilor Java). Există numeroase situații în care un utilizator PC transmite un fişier executabil ataşat la un email, însă destinatarul nu este în măsură să-l execute, de exemplu pe o platformă Macintosh sau Linux. Acest eveniment neplăcut apare datorită faptului că limbajul maşină este diferit de la un tip de calculator la altul.

Această problemă poate fi rezolvată prin introducerea conceptului de **maşină virtuală**. Compilatorul unui anumit limbaj de programare va genera cod pentru o maşină virtuală în loc să genereze cod pentru limbajul maşină al unui anumit tip de calculator. De exemplu, compilatorul Java generează cod pentru maşina virtuală Java, acesta fiind implementat o singură data indiferent de tipul de calculator. Numai că soluția Java nu este aplicabilă programelor scrise în alte limbaje de programare.

## SISTEME DESCHISE

Un sistem distribuit deschis este acel sistem care oferă servicii în conformitate cu regulile care descriu sintaxa şi semantica serviciilor respective. De exemplu, în rețelele de calculatoare formatul, conținutul şi semnificația mesajelor transmise sunt stabilite prin intermediul unui set de reguli, formalizate sub forma protocoalelor de comunicație.

În sistemele distribuite, serviciile sunt specificate prin intermediul **interfețelor**, descrise adesea prin intermediul unui **limbaj de definire a interfețelor (Interface Definition Language – IDL)**. De regulă, specificațiile unei interfețe scrise într-un IDL privesc doar sintaxa serviciilor, adică numele funcției care este disponibilă împreună cu parametrii, valorile returnate, excepțiile care pot apare etc. Specificațiile care privesc semantica serviciilor (adică descrierea exactă a modului în care sunt realizate serviciile respective) sunt descrise prin intermediul limbajului natural. În fapt, ea se referă la specificațiile de proiectare detaliată ale serviciilor.

Atingerea acestui obiectiv nu este posibilă fără ca specificațiile şi documentația interfețelor componentelor software ale sistemului să fie cunoscute proiectanților şi programatorilor, adică să fie publice. Prin urmare, importanța întocmirii specificațiilor de proiectare este accentuată în cazul dezvoltării sistemelor distribuite. Ele au darul de a uşura sarcina proiectanților de sisteme distribuite în care multe componente sunt dezvoltate de persoane diferite. De exemplu, publicarea specificațiilor protocolului de comunicare al Internetului, sub forma unei serii de documente numite "Requests For Comments" (RFCs), a făcut posibilă dezvoltarea unui imens număr de aplicații Internet. De asemenea, modelul CORBA este publicat prin intermediul unei serii de documente tehnice, care includ specificațiile complete ale interfețelor serviciilor disponibile.

Specificațiile interfețelor trebuie să fie **complete** şi **neutre**. Prin specificații complete se înțelege ca tot ceea ce este necesar realizării unei implementări să fie făcute cunoscute. De asemenea, este foarte important ca specificațiile să nu prescrie modul în care trebuie realizată o implementare; ele trebuie să fie neutre. Completitudinea şi neutralitatea specificațiilor sunt importante pentru obținerea interoperabilității şi a portabilității. **Interoperabilitatea** reprezintă măsura în care două componente implementate de dezvoltatori diferiți pot co-exista şi lucra împreună prin apelarea serviciilor fiecăreia în maniera stabilită prin standardul comun. **Portabilitatea** se referă la faptul că o aplicație dezvoltată pentru sistemul distribuit A poate fi executată, fără nici o modificare, pe sistemul distribuit B, diferit de primul, dacă B implementează aceleaşi interfețe ca şi A.

Caracterul deschis al sistemelor distribuite este determinat în primul rând de gradul în care noi servicii privind resursele partajate pot fi adăugate şi utilizate de o varietate de programe. Altfel spus, caracterul deschis determină dacă sistemul distribuit poate fi extins sau reimplementat în diferite moduri. El poate fi extins la nivelul

hardware, prin adăugarea de calculatoare, sau la nivelul software, prin adăugarea unor noi servicii şi reimplementarea celor vechi.

**Securitatea.** Multe din resursele informaționale disponibile în sistemele distribuite au o valoare intrisecă pentru utilizatorii săi. De aceea, securitatea lor prezintă o importanță deosebită. Securitatea resurselor informaționale are trei componente:

- **confidențialitatea** – protecția împotriva divulgării neautorizate;
- **integritatea** – protecția împotriva modificării sau denaturării;
- **disponibilitatea** – protecția împotriva accesului neautorizat la resurse.

În general, cu cât numărul facilităților oferite utilizatorilor săi este mai mare, cu atât problema securității unui sistem distribuit este mai dificil de rezolvat. Riscurile de securitate într-un intranet sunt legate de accesul liber la toate resursele sistemului. Deşi se poate apela la utilizarea unui firewall pentru a forma o barieră în jurul sistemului care să restricționeze traficul la intrare şi la ieşire, el nu va asigura utilizarea corespunzătoare a resurselor de către utilizatorii din intranet.

Spre deosebire de sistemele centralizate, problema securității sistemelor distribuite este foarte amplă, ea neputând fi abordată în câteva pagini. Oricum, proiectanții de sisteme distribuite trebuie să fie conştienți de importanța ei şi să-i acorde atenția cuvenită.

**Scalabilitatea.** Problema scalabilității este tipică şi foarte importantă în dezvoltarea sistemelor distribuite. Un sistem este considerat scalabil dacă el rămâne eficace atunci când apare o creştere semnificativă a numărului de resurse şi de utilizatori. Internetul reprezintă cea mai bună ilustrare a scalabilității unui sistem, el funcționând bine în condițiile creşterii dramatice a numărului calculatoarelor conectate şi a serviciilor furnizate.

În cazul în care noi servicii sau utilizatori trebuie adăugați, principalele probleme derivă din limitele centralizării serviciilor, datelor şi a algoritmilor. Acest lucru se poate observa în tabelul de mai jos, care oferă exemple de limite ale scalabilității.

| **Concepte** | **Exemple** |
|---|---|
| Servicii centralizate | Un singur server Web pentru toți utilizatorii |
| Date centralizate | O singură bază de date pentru tranzacțiile unei bănci |
| Algoritmi centralizați | Rutarea bazată pe informații complete |

În cazul multor servicii centralizate (adică ele sunt implementate pe un singur server), creşterea numărului de utilizatori poate duce la apariția aşa-ziselor "gâtuiri". Chiar dacă ar dispune de capacitate de prelucrare şi stocare nelimitată, comunicarea cu acel server tot va limita creşterea, implicit şi scalabilitatea sistemului. Soluția alternativă este evidentă: distribuirea serviciilor respective pe mai multe servere. Numai că, în unele situații acest lucru nu este recomandat. Dacă avem un serviciu care gestionează informații confidențiale, precum conturile bancare, soluția cea mai bună constă în implementarea serviciului respectiv pe un singur server securizat într-o cameră specială şi protejat față de celelalte părți ale sistemului distribuit prin intermediul unor componente de rețea speciale.

Dezavantajele centralizării serviciilor se regăsesc şi în cazul centralizării datelor. Gestionarea datelor dintr-o bancă printr-o bază de date centralizată (adică implementată pe un singur server de baze de date) este aproape imposibilă datorită atât volumului mare de date ce trebuie stocat, cât şi mediului tranzacțional puternic (de ordinul sutelor

de tranzacţii pe secundă), ceea ce solicită din plin capacitatea de comunicare a serverului. În mod asemănător, cum ar fi putut funcţiona Internetul dacă DNS-ul (Domain Name System) ar fi fost implementat într-o singură tabelă localizată pe un singur server. După cum ştim, DNS-ul gestionează informaţii despre milioane de calculatoare din lume şi permite localizarea serverelor web. Dacă cererile de rezolvare a unui URL ar fi transmise unui singur server DNS, atunci nimeni nu ar mai fi putut utiliza web-ul astăzi.

Problemele sunt asemănătoare în cazul centralizării algoritmilor. De exemplu, în sistemele distribuite mari trebuie transmise un număr imens de mesaje care trebuie direcţionate pe mai multe linii de comunicaţie. Teoretic, soluţia optimă presupune colectarea informaţiilor despre încărcarea tuturor calculatoarelor şi a liniilor de comunicaţie, iar pe baza unui algoritm de calcul să se determine calea optimă de transmisie. Numai că, această soluţie ar duce la încărcarea suplimentară a calculatoarelor şi a liniilor de comunicaţie, de vreme ce ele trebuie să schimbe numeroase mesaje pentru a transmite informaţiile necesare stabilirii căii optime la un moment dat. De aceea, în practică se apelează la descentralizarea algoritmilor.

Până acum am discutat problema scalabilităţii din perspectiva creşterii dimensiunii sistemului prin adăugarea de resurse (hard sau soft) şi utilizatori. Ea este mult mai complexă dacă luăm în considerare alte două dimensiuni ale scalabilităţii: scalabilitatea geografică şi scalabilitatea administrativă. S*calabilitatea geografică* se referă la sistemele în care resursele şi utilizatorii sunt localizaţi la distanţe foarte mari; *scalabilitatea administrativă* presupune ca sistemul să fie uşor de gestionat chiar dacă el este răspândit pe mai multe organizaţii administrative independente.

Dacă până aici am prezentat problemele care se ivesc în legătură cu scalabilitatea sistemelor distribuite, în continuare vom prezenta pe scurt soluţiile de rezolvare a lor. În acest sens, sunt folosite în general două tehnici de scalare: distribuirea şi replicarea.

**Distribuirea** implică descompunerea unei componente în mai multe părţi mai mici şi răspândirea acestora în diferite locuri din sistem. Un exemplu elocvent în acest sens îl reprezintă DNS-ul din Internet, care este descompus şi organizat ierarhic în trei domenii, iar acestea la rândul lor sunt divizate în mai multe zone; numele din fiecare zonă sunt gestionate prin intermediul unui singur nume de server. În mod asemănător, o aplicaţie poate fi proiectată astfel încât realizarea unei funcţii să fie asigurată, de mai multe module de program ce pot fi răspândite pe mai multe servere sau între server şi clienţi.

**Replicarea** reprezintă o altă soluţie pentru rezolvarea problemei scalabilităţii sistemelor distribuite. Ea nu determină doar sporirea disponibilităţii, dar ajută şi la repartizarea încărcării între componentele sistemului în vederea îmbunătăţirii performanţelor. O formă specială a replicării o reprezintă **cashing-ul**. Distincţia dintre replicare şi cashing este dificilă sau chiar artificială. Ca şi în cazul replicării, cashing-ul presupune existenţa mai multor copii ale unei resurse plasate în apropierea clienţilor care o accesează. Spre deosebire de replicare, cashing-ul este o decizie pe care o ia clientul care accesează o resursă şi nu proprietarul resursei (respectiv serverul). Dincolo de avantajele celor două tehnici (replicarea şi cashing-ul), neluarea în considerare a altor aspecte la utilizarea lor poate determina o diminuare a performanţelor şi nu o îmbunătăţire a lor. Existenţa mai multor copii pentru o resursă poate duce la situaţia în care după modificarea unei copii, aceasta să fie diferită de celelalte. Aşadar, utilizarea

celor două tehnici pot duce la apariția problemelor de consistență, asupra cărora vom reveni cu detalii în capitolele următoare.

## TRATAREA DISFUNCȚIONALITĂȚILOR

Sistemele informatice înregistrează uneori căderi în funcționarea normală. În cazul disfuncționalităților hardware sau software, programele pot produce rezultate incorecte sau pot înceta să mai funcționeze înainte de a-şi fi terminat sarcina de realizat. Rezolvarea lor este dificilă, deoarece disfuncționalitățile în sistemele distribuite sunt parțiale. Aceasta înseamnă că unele componente pot să înceteze să mai funcționeze, în timp ce altele continuă să funcționeze. De aceea, rezolvarea disfuncționalităților presupune găsirea soluțiilor la următoarele probleme:

• **detectarea disfuncționalităților**. De exemplu, depistarea datelor alterate dintr-un mesaj sau fişier poate fi realizată prin intermediul sumelor de control. Există şi situații în care prezența unei disfuncționalități nu poate fi sesizată.

• **ascunderea disfuncționalităților**. Acest lucru poate fi realizat prin retransmiterea mesajelor în cazul în care ele nu au ajuns la destinație sau salvarea datelor pe un disc pereche astfel încât, dacă datele de pe primul disc sunt inconsistente, ele pot fi refăcute prin intermediul celui de-al doilea disc.

• **tolerarea disfuncționalităților**. În unele situații (precum sistemele distribuite mari) nu este indicată ascunderea disfuncționalităților. Soluția alternativă constă în proiectarea aplicațiilor client astfel încât să le tolereze, ceea ce implică tolerarea lor şi din partea utilizatorilor. De exemplu, atunci când un browser nu poate contacta un server de web, el nu-l va face pe utilizator să aştepte pentru a încerca de mai multe ori să stabilească legătura; el îl va informa pe utilizator de problema ivită şi rămâne la latitudinea acestuia să încerce din nou mai târziu, el având posibilitatea să continue cu o altă activitate.

• **refacerea în urma căderilor din sistem**. Rezolvarea acestei probleme presupune proiectarea aplicațiilor astfel încât să mențină consistența datelor la căderea unui server.

**Transparența.** Transparența reprezintă unul dintre cele mai importante obiective urmărite la dezvoltarea sistemelor distribuite şi care are influențe majore asupra activității de proiectare. Acest concept presupune ascunderea față de utilizatori, programatori şi aplicații a faptului că procesele şi resursele din sistem sunt distribuite fizic pe mai multe calculatoare. Sistemul trebuie perceput ca un întreg şi nu ca o colecție de componente independente.

Modelul de referință al ISO (International Standards Organization) pentru sistemele deschise cu prelucrări distribuite prezintă 8 forme ale transparenței, prezentate în tabelul de mai jos.

| **Forme ale transparenței** | **Descriere** |
|---|---|
| Acces | Ascunde diferențele în reprezentarea datelor. De asemenea, permite accesarea resurselor locale şi a celor aflate la distanță utilizând operațiuni identice |
| Localizare | Ascunde localizarea resurselor în sistem |
| Migrare | Ascunde faptul că resursele au fost mutate la o altă locație din sistem. |

| | |
|---|---|
| Re-localizării | Ascunde faptul ca resursele pot fi mutate la o altă locație din sistem chiar în timpul utilizării lor. |
| Replicare | Ascunde existența unei copii pentru o resursă |
| Concurența | Permite utilizarea concurentă a resurselor partajate de către mai mulți utilizatori sau procese (programe), fără să existe interferențe între aceştia. |
| Disfuncționalități | Ascunderea disfuncționalităților, cu posibilitatea ca utilizatorii să-şi poată termina sarcinile de lucru. |
| Persistența | Ascunde faptul că o resursă software este în memoria calculatorului sau pe disc. |

Cel mai bun exemplu pentru ilustrarea transparenței accesului îl reprezintă instrumentul grafic Windows Explorer. Acesta, indiferent de faptul că un folder este local sau se află pe un alt calculator, va folosi acelaşi simbol pentru reprezentarea lor, ele putând fi accesate în acelaşi mod, adică prin selectarea folderului dorit. Lipsa transparenței accesului este evidentă atunci când un sistem distribuit nu permite accesarea unui fişier aflat pe un alt calculator decât prin intermediul programului ftp. De asemenea, modul de reprezentare a datelor diferă de la un tip de calculator la altul, de la un sistem de operare la altul, de la un limbaj de programare la altul. Aceste diferențe trebuie să fie ascunse utilizatorilor şi aplicațiilor dintr-un sistem distribuit.

**Transparența localizării** se referă la faptul că utilizatorul nu ştie unde este plasată fizic o anumită resursă din sistem, cu toate că el poate să o acceseze. Acest tip de transparență poate fi obținută prin atribuirea de nume logice resurselor din sistem. Utilizarea URL-urilor în web constituie un astfel de exemplu. Partea din URL care identifică numele de domeniu al unui server web se referă la numele calculatorului şi nu la adresa sa IP.

Transparența accesului şi cea a localizării mai sunt cunoscute împreună ca **transparența rețelei**.

Atunci când într-un sistem distribuit resursele pot fi mutate fără a fi afectat modul în care ele sunt accesate se spune că asigură **transparența migrării**. Chiar dacă un program sau o tabelă dintr-o bază de date sunt mutate în altă parte a sistemului, utilizatorul nu va sesiza acest lucru la accesare, iar aplicația nu va trebui modificată pentru a reflecta noua localizare. Importanța acestui tip de transparență şi căile de realizare a ei în bazele de date distribuite vor fi discutate pe larg într-un capitol viitor. Şi mai puternică este **transparența re-localizării**, care presupune mutarea unor resurse chiar în timpul accesării lor fără ca utilizatorii sau aplicațiile să fie nevoite să ia în considerare această operațiune. Acest tip de transparență apare, de exemplu, atunci când un utilizator mobil continuă să utilizeze laptopul sau conectat printr-o rețea fără fir pe durata deplasării sale de la un loc la altul fără să fie deconectat fie şi temporar. Acest tip de transparență este întâlnit mai ales în sistemele distribuite mobile.

Replicarea joacă un rol important în sistemele distribuite, resursele putând fi replicate în vederea creşterii disponibilității şi a performanțelor prin plasarea unei copii cât mai aproape de locul din care ele sunt accesate. Replicarea poate fi utilizată împreună cu tehnologia bazelor de date distribuite pentru a îmbunătăți performanțele accesării datelor. În sfârşit, **transparența replicării** presupune ascunderea faptului că pentru o resursă există mai multe copii în cadrul sistemului. Pentru a ascunde acest lucru față de utilizatori, va trebui ca toate copiile să aibă acelaşi nume. De aceea, pentru

ca un sistem să ofere transparența replicării, el trebuie să asigure şi transparența localizării. Altfel, ar fi imposibilă referirea copiilor plasate pe locații diferite.

Unul dintre avantajele sistemelor distribuite îl reprezintă posibilitatea partajării resurselor. De exemplu, doi utilizatori pot accesa simultan aceeaşi tabelă a bazei de date, ceea ce înseamnă că ei partajează tabela respectivă. Numai că, este foarte important ca fiecare utilizator să nu sesizeze faptul că un alt utilizator accesează aceeaşi resursă odată cu el. Acest fenomen este numit **transparența concurenței.** Revenind la exemplul nostru, vom vedea mai târziu că rezolvarea acestei probleme nu este atât de simplă, deoarece cei doi utilizatori pot lăsa datele din baza de date într-o stare inconsitență (presupunem că este vorba despre tabela de stocuri şi că cei doi utilizatori încearcă să vândă acelaşi produs simultan). De aceea, anticipând nițel, se apelează la mecanismul de blocare, ceea ce contravine acestui tip de transparență.

**Transparența disfuncționalităților** presupune ca utilizatorii să nu observe că anumite resurse au încetat să mai funcționeze sau nu mai funcționează normal. Mai mult, sistemul va asigura ulterior refacerea situației datorate acelei disfuncționalități. Asigurarea acestui tip de transparență este cel mai dificil, datorită inabilității sistemului de a distinge între o resursă "moartă" şi una care funcționează, dar foarte încet. De exemplu, la accesarea unui server web foarte solicitat, este posibil ca browserul să raporteze că pagina respectivă nu este disponibilă.

Ultimul tip de transparență asociat sistemelor distribuite se referă la **transparența persistenței**, şi se referă la ascunderea faptului că o componentă software se află în memoria volatilă sau pe disc. Acest tip de transparență este important în cazul serverelor de baze de date. A se revedea în acest sens zona SQL Share şi procesul de scriere din Oracle Server.

În final trebuie spus că realizarea transparenței distribuirii reprezintă un obiectiv important în proiectarea şi implementarea sistemelor distribuite însă, uneori este greu de obținut, iar în alte situații trebuie luate în calcul şi alte aspecte precum performanța sistemului.

Prezentarea succintă a celor mai importante obiective urmărite la proiectarea şi implementarea sistemelor distribuite, care se adaugă la cele care privesc în general sistemele informatice, demonstrează complexitatea activității de dezvoltare a sistemelor distribuite.

## ARHITECTURA SISTEMELOR DISTRIBUITE

Sistemele distribuite implementate până în prezent evidențiază o varietate arhitecturală mare. Cu toate acestea, ele au în comun o serie de caracteristici şi împărtăşesc unele probleme comune în dezvoltarea lor. Caracteristicile comune şi aspectele de proiectare a sistemelor distribuite pot fi prezentate sub forma unor modele descriptive. Fiecare astfel de model va reprezenta o descriere abstractă, simplificată dar consistentă a aspectelor relevante ale proiectării sistemelor distribuite.

**Definirea arhitecturii sistemelor distribuite.** O definiție standard, universal acceptată, pentru arhitectura sistemului informatic nu există, majoritatea opiniilor exprimate punând în centrul atenției conceptele de **componentă** şi **conexiune**. Una din definițiile mai recente consideră arhitectura programelor ca fiind „structura sau structurile care privesc componentele programului, proprietățile externe ale acestor componente, precum şi relațiile dintre ele".

În funcție de semnificația noțiunii de componentă, arhitectura sistemelor informatice poate fi definită într-un sens restrâns şi într-un sens mai larg. Proiectarea arhitecturii unui program poate viza, în sens restrâns, componentele programului, respectiv modulele acestuia, însă ea poate fi extinsă prin includerea bazei de date şi a componentei middleware care permite configurarea comunicării într-un sistem client/server.

Proprietățile acestor componente sunt acele caracteristici care permit înțelegerea modului în care ele interacționează, respectiv modul de apelare a unui modul din alt modul sau mecanismul de accesare a bazei de date de către modulele programului. Proiectarea arhitecturală a programului nu ia în considerare proprietățile interne ale componentelor, cum ar fi detaliile unui algoritm specifice unui modul.

Relațiile dintre componente se pot referi fie la apelarea unei proceduri, cu transmiterea eventuală a datelor necesare execuției procedurii respective, fie la protocolul de accesare a bazei de date de către procedurile de program.

Obiectivul general urmărit în cadrul proiectării arhitecturale vizează conceperea unei structuri a sistemului care să corespundă cerințelor prezente şi celor viitoare, astfel încât sistemul să fie sigur în funcționare, adaptabil, uşor de gestionat, eficient. O bună proiectare arhitecturală se va traduce într-un sistem uşor de implementat, testat şi modificat.

Multitudinea sistemelor informatice distribuite implementate până în prezent relevă o varietate mare a arhitecturilor, dar care pot totuşi fi încadrate în câteva modele arhitecturale. Un model arhitectural defineşte modul în care interacționează între ele componentele unui sistem, precum şi localizarea (maparea) lor într-o rețea de calculatoare. Modelul arhitectural al unui sistem distribuit are rolul de a simplifica şi abstractiza (în sensul de a evidenția caracteristicile esențiale ale sistemului) funcțiile componentelor sistemului. Apoi, el ia în considerare:

• plasarea componentelor în cadrul rețelei – căutând să definească modelele corespunzătoare de distribuire a datelor şi a prelucrărilor;

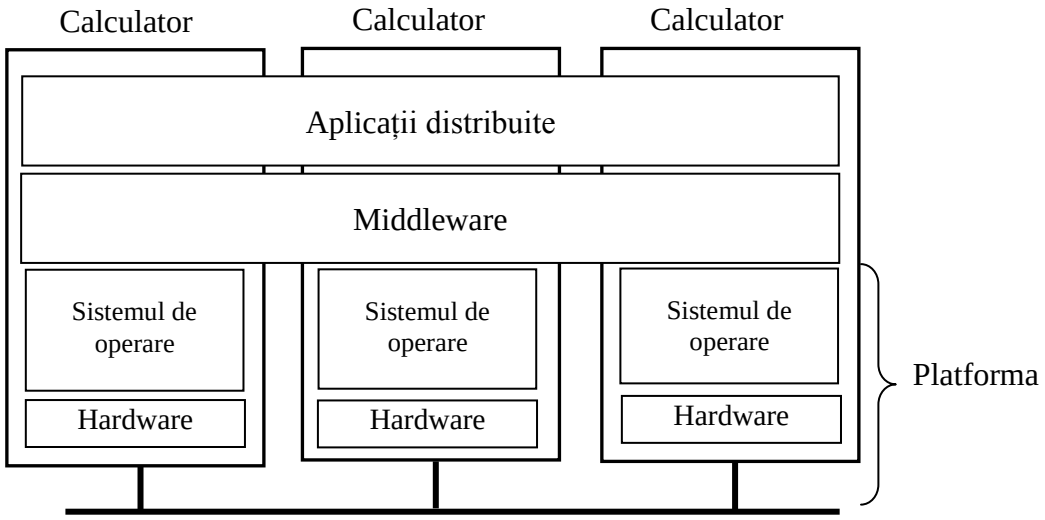
• interacțiunile dintre componente – adică, rolurile lor funcționale şi modelele de comunicare dintre ele.

Modelele de alocare a sarcinilor de lucru într-un sistem distribuit se reflectă direct asupra performanțelor şi eficacitatea sistemului rezultat. Localizarea componentelor unui sistem distribuit este determinată de aspectele de performanță, siguranță în funcționare, securitate şi costurile implicate.

## ARHITECTURA SOFTWARE

Într-un sistem distribuit hardware-ul este important însă software-ul reprezintă elementul determinant; de componenta software depinde cum va arăta un sistem distribuit. De aceea, discuția privind arhitectura sistemelor distribuite se va axa pe arhitectura software.

Inițial, prin **arhitectura software** se făcea referire la structurarea software-ului pe niveluri sau module, cea mai cunoscută fiind structura ierarhică pe module. Recent, acelaşi termen este descris în termenii serviciilor oferite şi solicitate între procesele localizate pe acelaşi calculator sau pe calculatoare diferite. Prin urmare, noua orientare, către procese şi servicii, poate fi exprimată prin intermediul **nivelurilor de servicii**. Ea este prezentată schematic în figura de mai jos.




După cum se poate observa, structura generală a unui sistem distribuit presupune trei niveluri (straturi): platforma, middleware şi programele de aplicații distribuite. Fiecare nivel oferă servicii nivelului superior. Astfel, aplicațiile distribuite apelează la serviciile oferite de componenta middleware care, la rândul său, beneficiază de serviciile oferite de sistemele de operare.

## PLATFORMELE HARDWARE ŞI SOFTWARE ÎN SISTEMELE DISTRIBUITE

Componenta hardware şi nivelul cel mai de jos al software-ului sunt adesea referite împreună prin termenul **platformă**. În practică ele pot fi referite separat prin platforma hardware şi platforma software. Acest nivel oferă servicii nivelurilor situate deasupra sa, servicii care sunt implementate în mod independent pe fiecare calculator. El oferă interfața de programare nivelului care facilitează comunicarea şi coordonarea dintre procese. Printre cele mai cunoscute exemple de platforme se regăsesc: Intel x86/Windows, Sun SPARC/SunOS, PowerPC/MacOS, Intel x86/Linux.

Referindu-ne la arhitectura hardware, ea specifică modul în care sunt conectate calculatoarele, mai concret procesoarele. Orice sistem distribuit presupune existența a multiple procesoare, dar care pot fi organizate în câteva moduri diferite în ce priveşte interconectarea şi comunicarea dintre ele. Deşi în literatura de specialitate au fost prezentate numeroase scheme de clasificare a sistemelor bazate pe multiple procesoare, nici una nu a primit o recunoaştere largă. În continuare vom face o succintă prezentare a câtorva clasificări.

Din punctul de vedere al partajării sau nu a memoriei, există două tipuri de sisteme: **multi-procesoare**, respectiv cele în care mai multe procesoare partajează memoria, şi **multi-calculatoare**, respectiv cele care care nu o partajează. În cazul sistemelor multi-procesoare, toate procesoarele partajează o singură zonă fizică de memorie. Astfel, dacă oricare procesor scrie valoarea 44 la adresa 1000, atunci ulterior oricare alt procesor care va citi valoarea conținută la adresa respectivă va prelua valoarea 44. În contrast, într-un sistem multi-calculatoare, fiecare procesor dispune de propria memorie. Dacă un procesor va scrie valoarea 44 la adresa 1000 a propriei memorii, ulterior un alt procesor care va citi valoarea conținută la adresa 1000 va obține

probabil o altă valoare. Cel mai comun exemplu de sistem multi-calculatoare îl reprezintă o colecție de calculatoare conectate la rețea.

O altă clasificare grupează sistemele distribuite în **sisteme eterogene** şi sisteme omogene. Această clasificare este aplicată doar în cazul sistemelor multi-calculatoare. Intr-un sistem omogen, toate procesoarele sunt la fel şi, în general, accesează zone fizice de memorie diferite dar de aceeaşi capacitate, iar în cadrul rețelei se utilizează aceeaşi tehnologie. De regulă, acest tip de sisteme sunt utilizate mai mult ca sisteme paralele (adică lucrează pentru aceeaşi problemă). În schimb, sistemele eterogene pot conține calculatoare de tipuri diferite, interconectate prin rețele de diferite tipuri. De exemplu, un sistem distribuit poate fi construit pe baza unei colecții de rețele locale care utilizează tehnologii diferite, ele putând fi interconectate printr-un backbone bazat pe tehnologia FDDI.

Atunci când vorbim despre platforma software, cel mai adesea se face referire la sistemul de operare. De fapt, sistemele distribuite se aseamănă în bună măsură cu sistemele de operare tradiționale. Ele gestionează resursele hardware permițând mai multor utilizatori şi aplicații să le partajeze. Prin resurse hardware se face referire la procesoare, memorie, echipamente periferice şi rețea. De asemenea, sistemele distribuite ascund complexitatea şi eterogenitatea resurselor hardware.

Sistemele de operare pentru sistemele distribuite pot fi împărțite în două categorii: **sisteme strâns-cuplate (tightly-coupled)** şi **sisteme slab-cuplate (loosely-coupled)**. În cazul sistemelor strâns-cuplate, sistemul de operare încearcă să mențină o singură imagine, globală, asupra resurselor hardware, în timp ce sistemele slab-cuplate pot fi văzute ca o colecție de calculatoare, fiecare cu propriul sistem de operare, dar care colaborează.

Distincția între sistemele strâns-cuplate şi cele slab-cuplate este legată de clasificările prezentate anterior pentru componenta hardware. Astfel, sistemele strâns-cuplate, referite şi ca **sisteme de operare distribuite**, sunt utilizate în sistemele multi-procesoare şi sistemele omogene. Principala sarcină a unui sistem distribuit rezidă în ascunderea complexității gestiunii resurselor hardware astfel încât ele să poată fi partajate de multiple procese. În schimb, sistemele slab-cuplate, referite adesea ca **sisteme de operare de rețea (NOS – Network Operating System)**, sunt utilizate în cazul sistemelor eterogene. Distincția dintre un NOS şi un sistem de operare tradițional constă în faptul că, pe lângă gestiunea resurselor, el asigură disponibilitatea serviciilor locale clienților aflați la distanță.

Majoritatea sistemelor distribuite sunt eterogene, deci utilizează un NOS. În această situație, sunt necesare unele servicii suplimentare celor oferite de NOS, care să asigure o mai bună transparență a naturii distribuite a sistemului. Toate aceste servicii sunt grupate pe un nivel intermediar numit **middleware**. El reprezintă inima sistemelor distribuite moderne. Dacă acesta ar lipsi, aplicațiile distribuite ar trebui să apeleze direct la serviciile oferite de sistemele de operare pentru realizarea diferitelor sarcini (cum ar fi transmiterea mesajelor), ceea ce ar presupune un efort suplimentar considerabil din partea programatorilor, întrucât fiecare sistem de operare are implementări diferite ale aceluiaşi serviciu. În tabelul următor este prezentată o comparație între sistemele distribuite (DOS), sistemele de operare pentru rețea (NOS) şi middleware.

| **Sistemul** | **Descriere** | **Obiectivul principal** |
|---|---|---|
| DOS | Sisteme de operare strâns-cuplate, utilizate în sistemele multi-procesoare şi sistemele | Ascunde complexitatea gestiunii resurselor |

| | | |
|---|---|---|
| | multi-calculatoare omogene | hardware |
| NOS | Sisteme de operare slab-cuplate, utilizate în sistemele multi-calculatoare eterogene (LAN şi WAN) | Oferă servicii locale clienților aflați la distanță |
| Middleware | Strat software adițional situat deasupra NOS-ului şi furnizează servicii generale | Furnizează transparența distribuirii |

Ultimul nivel al arhitecturii software conține aplicațiile specifice diferitelor domenii, care apelează la serviciile oferite de middleware. Utilizatorii interacționează cu sistemul distribuit prin intermediul acestor programe.

## NIVELUL MIDDLEWARE

O definiție mai formală, consideră middleware-ul ca un nivel al software-ului al cărui scop constă în mascarea eterogenității (platformei, s.n.) şi furnizarea unui model de programare comod dezvoltatorilor de aplicații. El este format din procese sau obiecte ce se regăsesc pe un grup de calculatoare, şi care interacționează între ele pentru a asigura implementarea comunicării şi partajării resurselor în aplicațiile distribuite.

Nivelul middleware sprijină comunicarea dintre programele de aplicații prin intermediul unor „abstractizări" precum invocarea metodelor de la distanță, comunicarea în cadrul unui grup de procese, notificarea evenimentelor, replicarea datelor partajate şi transmisia în timp real a datelor mutimedia.

Unele aplicații distribuite apelează direct la interfața de programare furnizată de sistemul de operare al rețelei, ignorând nivelul middleware. Avantajul oferit de nivelul middleware constă în ascunderea eterogenității platformelor pe care este implementat sistemul distribuit. De aceea, majoritatea sistemelor middleware oferă o colecție de servicii mai mult sau mai puțin completă, descurajând utilizarea altor interfețe decât a celor către propriile servicii.

Pentru a simplifica dezvoltarea şi integrarea aplicațiilor distribuite, majoritatea soluțiilor middleware se bazează pe un anumit **model**, care descrie aspectele privind distribuirea şi comunicarea. Cele mai utilizate astfel de modele sunt: apelarea procedurilor de la distanță, distribuirea obiectelor şi distribuirea documentelor.

**Apelarea procedurilor de la distanță – RPC (Remote Procedure Calls)**. Unul dintre primele modele middleware are la bază mecanismul RPC. În acest model, accentul este pus pe ascunderea particularităților comunicației în rețea astfel încât să permită unui proces să apeleze o procedură localizată pe un alt calculator. La apelarea unei astfel de proceduri, parametrii sunt transmişi în mod transparent calculatorului pe care este localizată procedura respectivă şi pe care ea va fi executată; rezultatele execuției sunt transmise înapoi procesului (procedurii) apelant(e). În acest mod, se va crea impresia că procedura apelată este executată local; procesul apelant nu are habar că este vorba de o comunicare în rețea, cu excepția eventualei întârzieri cu care primeşte rezultatele. O soluție middleware care se bazează pe acest model este Sun RPC.

**Distribuirea obiectelor**. Lansarea modei „orientate obiect" a avut efecte şi asupra soluțiilor middleware. Atât timp cât o procedură poate fi apelată de la distanță, s-a pus problema posibilității invocării obiectelor rezidente pe alte calculatoare într-o manieră transparentă. Astfel, a apărut un nou model, care stă la baza multor soluții middleware. În categoria soluțiilor middleware bazate pe distribuirea obiectelor se încadrează **CORBA (Common Object Request Broker Architecture)** al OMG

(Object Management Group), **Java RMI (Java Remote Object Invocation)**, **DCOM (Distributed Component Object Model)** al Microsoft şi **RM-ODP (Reference Model for Open Distributed Processing)** al ISO/ITU-T. Esența modelului bazat pe distribuirea obiectelor constă în faptul că fiecare obiect implementează o interfață care ascunde detaliile interne ale obiectului față de utilizatorii săi (a se înțelege de fapt programatori). Singurul lucru pe care un proces îl poate vedea la un obiect este interfața sa. O interfață constă în metodele pe care obiectul le implementează.

De regulă obiectele distribuite sunt implementate astfel încât fiecare obiect să fie localizat pe un singur calculator şi, în plus, interfața sa să fie disponibilă (vizibilă) şi pe alte calculatoare. Invocarea unei metode de către un proces este transformată într-un mesaj care va fi transmis obiectului în cauză (localizat pe alt calculator decât cel de pe care este inițiat procesul); obiectul va executa metoda cerută şi va transmite înapoi rezultatele, tot sub formă de mesaje; mesajul de răspuns este transformat în valoare, ce va fi preluată şi prelucrată corespunzător de procesul invocant. Ca şi în cazul mecanismului RPC, procesul apelant nu va fi conştient de comunicația care a avut loc în rețea.

**Distribuirea documentelor**. Succesul Web-ului se datorează în bună măsură simplității şi eficacității modelului middleware bazat pe distribuirea documentelor. În acest model, informațiile sunt organizate sub formă de documente (care conțin nu doar date de tip text, dar şi video, audio etc), fiecare document fiind rezident pe un anumit calculator, localizarea sa fiind transparentă. Documentele pot conține link-uri către alte documente, iar prin intermediul unui astfel de link documentul la care face referire poate fi descărcat de pe calculatorul pe care este rezident şi afişat pe ecranul utilizatorului.

După cum spuneam anterior, middleware-ul pune la dispoziție o serie de servicii care pot fi utilizate de programele de aplicații. De exemplu, standardul CORBA (Common Object Request Broker Architecture) oferă o varietate de servicii prin intermediul cărora furnizează aplicațiilor o serie de facilități, precum: atribuirea numelor, securitate, tranzacții, persistența, notificarea evenimentelor. Serviciile cele mai comune oferite de majoritatea soluțiilor middleware sunt:

• **Transparența accesului**. Toate soluțiile middleware acoperă această cerință. Astfel de servicii oferă facilități de comunicare de nivel înalt care ascund modul în care are loc transmiterea „low-level" a mesajelor prin rețelele de calculatoare. În acest fel, interfața de programare a nivelului de transport specifică diferitelor sisteme de operare de rețea este înlocuită complet (de exemplu, nivelul transport în sistemul de operare Windows NT este implementat prin protocolul TCP). Modul în care este asigurat suportul comunicării diferă foarte mult în funcție de modelul de distribuire pe care o soluție middleware o oferă utilizatorilor şi aplicațiilor. După cum am văzut anterior, apelarea procedurilor de la distanță sau invocvarea obiectelor distribuite reprezintă astfel de modele. În plus, multe din soluțiile middleware oferă facilități nu doar pentru transmiterea mesajelor, ci şi pentru transparența accesării datelor aflate la distanță, cum ar fi bazele de date distribuite. Un alt exemplu de comunicare de nivel înalt îl reprezintă încărcarea documentelor de pe web.

• **Utilizarea numelor (naming)**. În sistemele distribuite, numele sunt utilizate pentru referirea unei mari varietăți de resurse, precum calculatoare, servicii, obiecte şi fişiere aflate la distanță, utilizatori. Utilizarea numelor facilitează partajarea şi regăsirea acestor entități. De exemplu, un URL reprezintă numele atribuit unei pagini web şi care va fi folosit pentru accesarea ei. Procesele nu pot partaja o anumită resursă gestionată de

un calculator decât dacă îi este asociat un nume. De asemenea, utilizatorii nu vor putea comunica între ei în cadrul unui sistem distribuit decât dacă ei pot fi referiți printr-un nume (de exemplu, adresele email). Acest serviciu este comparabil cu o carte de telefoane sau arhicunoscutele pagini aurii. O limită specifică utilizării numelor este legată de faptul că localizarea entității care este referită prin nume trebuie să fie fixă. Această ipoteză stă la baza conceperii web-ului, de exemplu. Fiecare document are atribuit un URL, acesta conținând şi numele serverului pe care este stocat documentul respectiv. Dacă se doreşte mutarea documentului pe un alt server, atunci numele (URL-ul) nu mai este valabil.

• **Persistența**. Multe din sistemele middleware oferă facilități de stocare. În forma cea mai simplă, persistența este asigurată prin intermediul unui sistem de fişiere distribuite. Soluțiile midleware mai avansate utilizează bazele de date sau oferă aplicațiilor facilități de conectare la baze de date.

• **Tranzacții distribuite.** Aceste servicii sunt utile în sistemele în care stocarea datelor joacă un rol important. O tranzacție reprezintă o operațiune atomică efectuată asupra unei baze de date (de exemplu, adăugarea unei noi facturi). Tranzacțiile distribuite operează asupra bazelor de date răspândite pe mai multe servere. De aceea, în cazul lor sunt necesare unele servicii suplimentare, cum ar fi ascunderea erorilor ivite în validarea sau anularea unei tranzacții. Asupra mecanismului tranzacțional, a tranzacțiilor distribuite şi a altor aspecte privind bazele de date distribuite vom reveni pe larg în capitolul 3.

• **Securitatea**. Desigur că sistemele de operare oferă serviciile necesare asigurării securității sistemului, la care aplicațiile pot apela. Spre deosebire de acestea, serviciile oferite de middleware sunt universale, adică pot fi utilizate la nivelul întregii rețele de calculatoare, ceea ce nu este valabil în cazul celor oferite de sistemul de operare care pot fi utilizate doar pe calculatoarele respective şi nu la nivelul întregii rețele. Prin urmare, serviciile de securitate sunt implementate din nou în nivelul middleware (deasupra celor oferite de sistemele de operare).

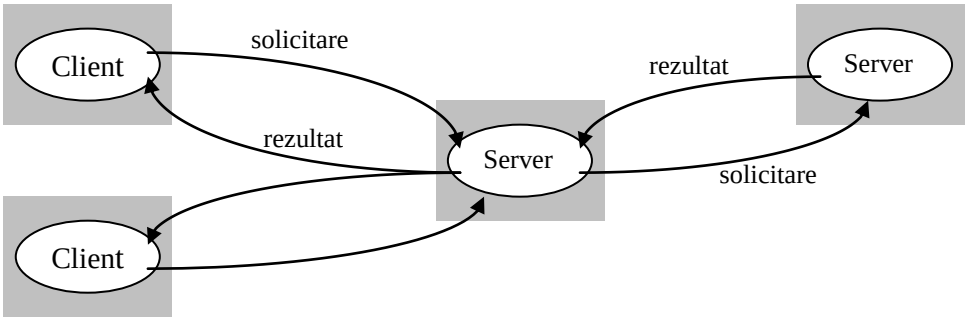
Multe din sistemele distribuite moderne sunt construite ca middleware pentru o serie de sisteme de operare. În acest fel, aplicațiile construite pentru un astfel de sistem distribuit vor fi independente de sistemul de operare. Totuşi extinderea pe scară largă a sistemelor distribuite este încetinită tocmai de soluțiile middleware, deoarece independența față de sistemul de operare a fost înlocuită cu o dependență puternică față de o anumită soluție middleware. Prin urmare este afectată una dintre caracteristicile esențiale ale sistemelor distribuite, prezentate în capitolul întâi, şi anume caracterul lor deschis.

Explicațiile rezidă în existența mai multor standarde dezvoltate de diferite organizații ca soluții middleware. De cele mai multe ori, aceste standarde sunt incompatibile între ele. Mai mult, produsele diferiților producători rareori interacționează corespunzător între ele, chiar dacă au implementat acelaşi standard. O astfel de situație poate apare datorită incompletitudinii definițiilor interfeței, care obligă programatorii să-şi creeze propriile interfețe (sau definiții). În consecință, aplicațiile scrise de ei ar putea să nu fie portabile, dacă două echipe dezvoltă propriile sisteme middleware, chiar dacă ambele echipe aderă la acelaşi standard (incomplet).

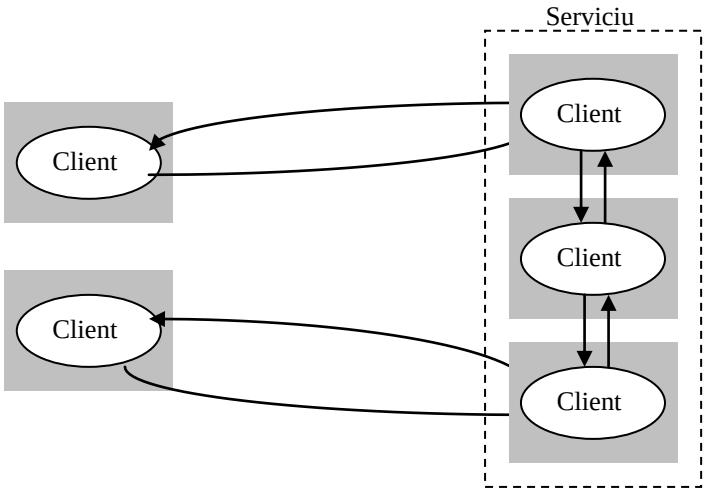
## MODELE ARHITECTURALE PENTRU SISTEMELE DISTRIBUITE

După cum arătam în primul paragraf, una activitățile specifice dezvoltării sistemelor distribuite constă în proiectarea arhitecturii sistemului, respectiv diviziunea responsabilităților între componentele sistemului şi plasarea lor pe calculatoarele din rețea. În acest sens, există mai multe modele arhitecturale. Asupra lor ne vom opri în continuare.

**Modelul client/server**. Această arhitectură este de departe cea mai cunoscută şi mai utilizată la dezvoltarea sistemelor distribuite, fiind prezentată schematic în figura de mai jos. În fapt, ea presupune împărțirea sarcinilor aplicației în procese client şi procese server care interacționează între ele prin schimbul de mesaje în vederea realizării unei activități. Acest model va fi discutat pe larg în paragraful următor.



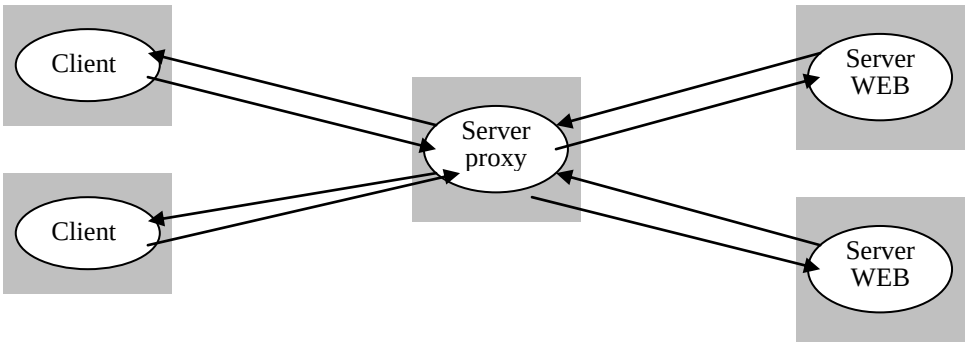

**Servicii furnizate de mai multe servere**. Conform acestei arhitecturi (vezi figura următoare), serviciile pot fi implementate sub forma mai multor procese server rezidente pe diferite calculatoare, care vor interacționa în funcție de necesități în vederea furnizării serviciului cerut de un proces client. Setul de obiecte care stă la baza serviciului respectiv poate fi partiționat şi distribuit pe mai multe servere. De asemenea, este posibil ca mai multe servere să întrețină copii ale obiectelor respective (este vorba despre replicare), cu scopul îmbunătățirii toleranței la erori, a perfomanțelor de accesare şi a disponibilității. De exemplu, serviciul web furnizat de *altavista.digital.com* este partiționat pe mai multe servere care conțin replici ale bazei de date.



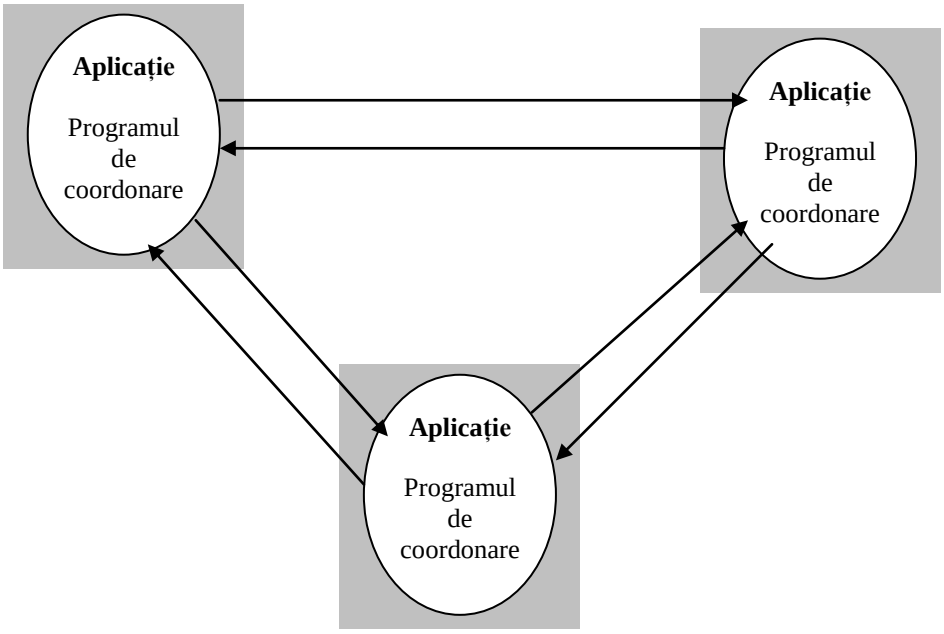

**Servere proxy şi tehnica de caching**. Cache reprezintă tehnica de stocare a obiectelor de date recent utilizate mai aproape de locul de utilizare. Atunci când un obiect este recepționat de un calculator, el va fi adăugat în zona de stocare cache,

înlocuind eventual alte obiecte care există deja în cache. La solicitatarea unui obiect de către un proces client, serviciul de caching va căuta mai întâi în cache pentru a pune la dispoziție obiectul solicitat, numai dacă există o copie actualizată a acestuia; altfel, o copie actualizată va fi încărcată de pe server. Zonele cache pot fi dispuse pe fiecare client sau ele pot fi localizate pe un server proxy partajat de mai mulți clienți.

Tehnica aceasta este utilizată pe scară largă în practică. Browserele Web întreţin pe fiecare client un cache cu cele mai recente pagini Web vizitate şi alte resurse Web. Ele utilizează o cerere HTTP specială pentru a verifica dacă paginile din cache sunt corespunzătoare cu cele originale de pe server înainte de a le afişa (este vorba de actualizarea lor). Serverele proxy Web (vezi figura următoare) oferă clienților o zonă de stocare cache partajabilă ce conține resursele Web ale unui singur site sau a mai multor site-uri. În acest mod, se obține o creştere a disponibilității şi performanțelor serviciilor Web prin reducerea încărcării rețelei şi a serverului Web.




**Procese perechi**. În această arhitectură toate procesele joacă roluri similare, interacționând în mod colaborativ ca perechi în vederea realizării unei activități sau prelucrări distribuite, fără a se face distincția între client şi server. Codul corespunzător proceselor perechi va avea rolul de a menține consistența resurselor de la nivelul aplicației şi de a sincroniza acțiunile de la nivelul aplicației dacă este necesar. În figura de mai jos este prezentată o astfel de arhitectură, formată din trei procese pereche, însă pot exista *n* procese care să interacționeze între ele.



Eliminarea proceselor server determină reducerea întârzierilor aferente comunicării inter-procese pentru accesarea obiectelor locale. De exemplu, o aplicație poate fi concepută astfel încât să permită utilizatorilor să afişeze şi să modifice interactiv o schiță (de exemplu schița unui proiect pentru un autoturism realizată cu un program special, de exemplu AUTOCAD) care este partajată. Aplicația poate fi implementată sub forma unor procese aplicație plasate pe fiecare nod care se va baza pe straturile middleware pentru a realiza notificarea evenimentelor şi comunicarea în cadrul grupului pentru a înştiința toate procesele aplicației despre eventuala modificare a schiței. Acest model oferă o comunicare interactivă mai bună (cu timpi de răspuns mai buni) pentru utilizatorii unui obiect distribuit partajat decât în cazul unei arhitecturi bazate pe server.

## MODELUL CLIENT/SERVER

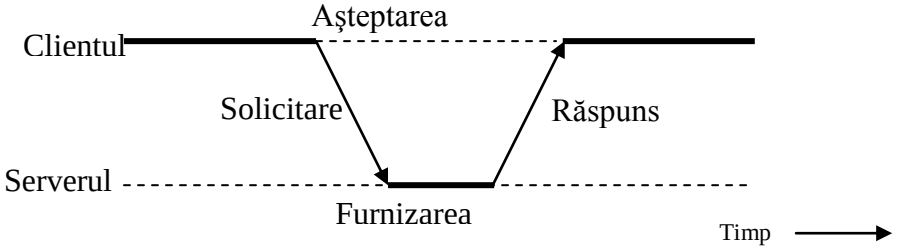
În general, puține sunt problemele legate de dezvoltarea sistemelor distribuite în care se înregistrează un consens în rândul specialiştilor. Un aspect asupra căruia se înregistrează un larg consens în rândul cercetătorilor şi practicienilor priveşte organizarea componentelor unui sistem distribuit prin folosirea termenilor **client**, care solicită servicii unui **server**, astfel încât să faciliteze înțelegerea şi stăpânirea complexității sistemelor distribuite. Aşadar, paradigma client/server reprezintă modelul arhitectural cel mai utilizat la dezvoltarea sistemelor distribuite.

## DEFINIREA MODELULUI CLIENT/SERVER

Ideea subiacentă conceptului client/server este **serviciul**. O aplicație informatică distribuită dezvoltată după modelul client/server este descompusă în două grupuri de procese: consumatorii de servicii, numiți **client** şi furnizorii de servicii, numiți **server**, care comunică între ele prin schimbul de mesaje de tip solicitare-răspuns. De exemplu, un server poate fi conceput pentru a oferi un serviciu de baze de date clienților săi. Serverul este funcțional independent de client, iar relația între client şi server este de colaborare (cooperare). Ea se diferențiază radical de aplicațiile centralizate, în care relația este de tip "stăpân-sclav" (master-slave).

În modelul client/server, clientul solicită serverului execuția unui serviciu prin transmiterea unui mesaj. La rândul său, serverul va transmite clientului rezultatul solicitării sale. Diferitele funcții ale aplicației informatice sunt regrupate sub forma programelor client şi server, fiecare cu roluri bine definite. Pentru utilizator totul este transparent, el comunicând cu programul client; schimbul de mesaje realizat între programele client şi server îi sunt transparente, el percepând aplicația ca un ansmablu executat doar pe postul său de lucru.

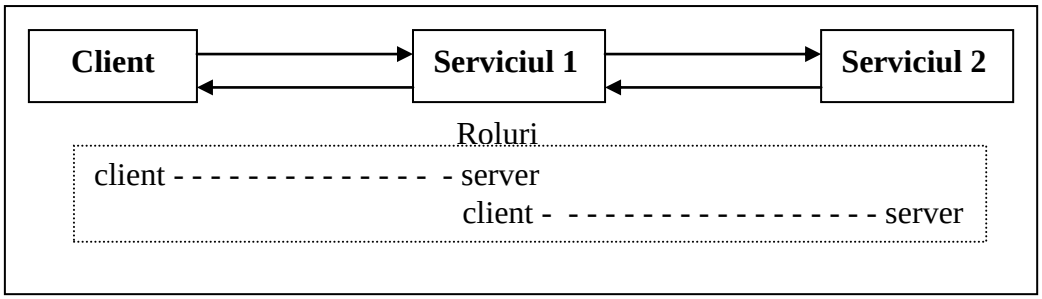
Figura următoare prezintă modelul general al interacțiunii dintre client şi server.



Arhitectura client/server poate fi definită ca un model de dezvoltare a aplicațiilor conform căruia sistemul informațional este descompus într-un mare număr de funcții server, executate pe una sau mai multe platforme hardware, care furnizează servicii comune unui mare număr de funcții client, executate pe una sau mai multe platforme hardware diferite dar interconectate, şi care realizează sarcini bine definite în legătură cu serviciile furnizate de server.

Spre deosebire de cele prezentate până acum, mai pot fi identificate două situații distincte:

• un server poate apela la serviciile furnizate de către un alt server, obținându-se o relație client server pe mai multe straturi. În figura de mai jos este prezentată o astfel de relație pe două straturi.




• programele client şi server se pot găsi pe acelaşi calculator, un exemplu în acest sens constituindu-l schimburile inter-aplicații de tip DDE (Dynamic Data Exchange).

Din cele prezentate până aici se poate clarifica relația dintre sistemele distribuite şi sistemele client/server. Astfel, într-un sistem client/server nu este obligatoriu ca cele două grupe de funcții (client şi server) să fie localizate pe calculatoare diferite, ele putând fi rezidente pe acelaşi calculator; de cele mai multe ori arhitectura client/server este implementată într-un sistem distribuit. Pe de altă parte, un sistem distribuit nu implică neapărat arhitectura client/server. Arhitectura cvasi-utilizată la dezvoltarea sistemelor distribuite este reprezentată de modelul client/server însă, nu este singura alternativă. Aşadar, deşi diferite conceptual, de cele mai multe ori în practica dezvoltării sistemelor informaționale se poate pune semnul de egalitate între sistemele distribuite şi sistemele client/server. De aceea, pe parcursul cursului cele două noțiuni vor fi utilizate interschimbabil cu acelaşi sens.

## ARHITECTURI CLIENT/SERVER MULTISTRAT

Modelul client/server a constituit subiectul multor dezbateri şi controverse. Problema principală este legată de distincția clară dintre client şi server. Proiectarea sistemelor client/server presupune conceperea arhitecturii aplicațiilor pe straturi bine definite. O astfel de abordare permite proiectarea independentă a straturilor, singura grijă constând în definirea clară şi proiectarea atentă a interfețelor, urmărindu-se ca:

• fiecare strat să aibă un domeniu bine definit, în sensul definirii foarte clare a sarcinilor şi responsabilităților fiecărui strat;

• fiecare strat trebuie să îndeplinească o sarcină specifică; dacă, de exemplu, unul din straturi este responsabil cu interacțiunea cu utilizatorul, atunci numai acel strat

va comunica cu utilizatorul, celelalte straturi realizând acest lucru prin intermediul acestui strat dacă au nevoie de informații de la utilizator.

• stabilirea unor protocoale bine definite pentru interacțiunea dintre straturi, interacțiune care să se realizeze numai prin intermediul acestor protocoale.

O primă încercare în acest sens a constituit-o împărțirea aplicațiilor pe două straturi, rezultând a**rhitectura cu două straturi**. Această arhitectură presupune descompunerea aplicației în următoarele două straturi:

• **stratul corespunzător aplicației**, în care se include interfața grafică cu utilizatorul, respectiv logica prezentării, şi implementarea regulilor de afaceri (business rules), respectiv logica aplicației. Tot acest strat poate coordona şi logica tranzacției, care garantează că actualizările în baza de date specifice unei tranzacții sunt terminate complet (validate sau anulate).

• **stratul corespunzător bazei de date**, care este responsabil de menținerea integrității bazei de date. În acest strat poate fi implementată întreaga logică a tranzacției sau o parte a ei.

Distincția dintre cele două straturi nu este întotdeauna bine definită deoarece logica tranzacției este adesea implementată pe server BD, sub forma procedurilor stocate, iar regulile afacerilor, parte a logicii aplicației sunt de asemenea implementate pe server, sub forma trigger-elor. În plus, sunt întâmpinate greutăți considerabile în dezvoltarea sistemului informațional pe baza creşterii accentuate a numărului de aplicații, a numărului şi tipului serverelor de baze de date. Această deficiență poate fi rezolvată prin introducerea unui nivel suplimentar, care să trateze regulile afacerii, rezultând o arhitectură cu trei straturi.

**Arhitectura cu trei straturi** presupune împărțirea aplicației în următoarele straturi:

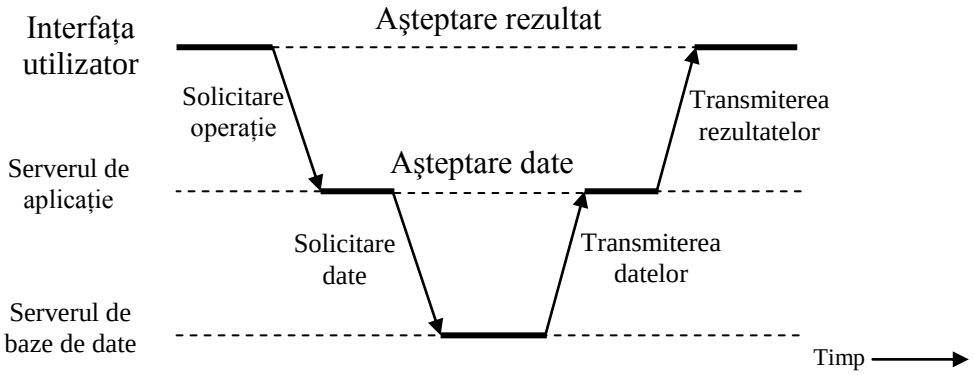
• **gestiunea interfației utilizator (gestiunea prezentării)** – priveşte dialogul între utilizatori şi aplicație, incluzând aici logica de prezentare a informației (ansamblul prelucrărilor efectuate asupra datelor necesare afişarii lor);

• **logica aplicației** - cuprinde ansamblul operațiilor de prelucrare specifice aplicației şi înlănțuirea lor logică;

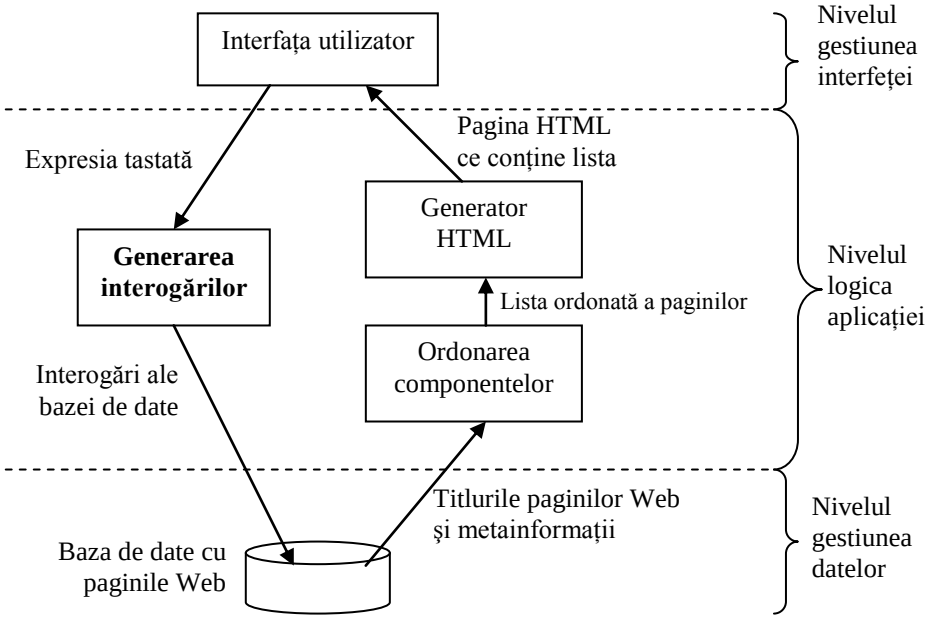
• **gestiunea datelor** – rezolvă cererile de date, asigură integritatea datelor, emiterea anumitor mesaje de alertare, precum şi gestiunea fizică a datelor (adăugări, modificări, ştergeri).

În esență, arhitectura pe trei straturi diferă de cea pe două straturi prin separarea logicii afacerii într-un strat distinct, localizat de regulă pe un server de aplicații care comunică strâns cu serverul de baze de date. Introducerea unui strat intermediar permite definirea şi implementarea regulilor afacerii independent de logica prezentării interfeței GUI şi a regulilor de proiectare a bazei de date. Acest avantaj devine evident în condițiile în care regulile afacerii sunt supuse mai des modificărilor, facilitând astfel reimplementarea lor.

Dacă cele trei straturi vor fi implementate pe calculatoare diferite, atunci vom avea situația în care un server va juca şi rolul de client. O arhitectură pe trei straturi este prezentată în figura de mai jos. Se observă că programele care formează stratul logicii afacerii sunt rezidente pe un server separat.




Un exemplu tipic de arhitectură pe trei straturi îl reprezintă modul de funcționare al unui motor de căutare pe Internet, prezentat în figura următoare.




Interfața (partea de front-end) permite utilizatorului să introducă expresia după care dorește să se efectueze căutarea şi, ulterior, va afişa o listă cu titlurile de pagini Web care corespund expresiei introduse. Partea de back-end va consta dintr-o imensă bază de date cu informații despre paginile Web. Între cele două niveluri se află "inima" motorului de căutare, respectiv partea de logică a programului, care transformă expresia introdusă de utilizator prin intermediul interfeței în una sau mai multe interogări ale bazei de date, după care va ordona rezultatele interogărilor într-o listă, şi pe care o va transforma într-o serie de pagini HTML.

Un alt exemplu de arhitectură pe trei straturi, este cel al unui sistem de asistare a deciziei pentru gestiunea portofoliului de acțiuni. Acest sistem poate fi de asemenea împărțit pe trei niveluri: partea front-end va implementa interfața utilizator, partea back-end va asigura accesarea bazei de date cu date financiare, iar partea de mijloc va conține programele de analiză financiară. Analiza datelor financiare poate implica tehnici şi metode sofisticate, de la cele statistice la cele de inteligență artificială, motiv pentru care logica aplicației ar putea fi implementată pe un server special, capabil să execute operațiuni de calcul complexe.

Printre avantajele unei arhitecturi client/server distribuită pe trei straturi enumerăm:

• **Reutilizare**. Componentele dezvoltate pot fi construite astfel încât funcționalitatea lor să fie partajate între mai multe aplicații;

• **Performanță**. Aplicațiile rulează într-un strat dedicat, bazat eventual pe resurse proprii şi tehnologii ale căror scop esențial este atingerea unei viteze de execuție superioare şi a unei scalabilități superioare.

• **Mentenanță**. Întreținerea şi reinstalarea aplicațiilor sau a unor părți ale acesteia, în cazul schimbării regulilor afacerii, este mult simplificată prin administrarea separată, centralizată, a componentelor lor.

• **Suport multi-limbaj**. Aplicațiile dezvoltate pe componente pot fi scrise în mai multe limbaje de programare (VB, C++, C#, Java) şi pot fi făcute interoperabile, chiar dacă provin de pe platforme diferite (.NET, J2EE).

• **Scalabilitate şi echilibrarea solicitării resurselor**. Componentele pot fi distribuite pe mai multe servere, ceea ce permite ridicarea pragului de scalabilitate în condițiile păstrării parametrilor de performanță şi disponibilitate a aplicațiilor.

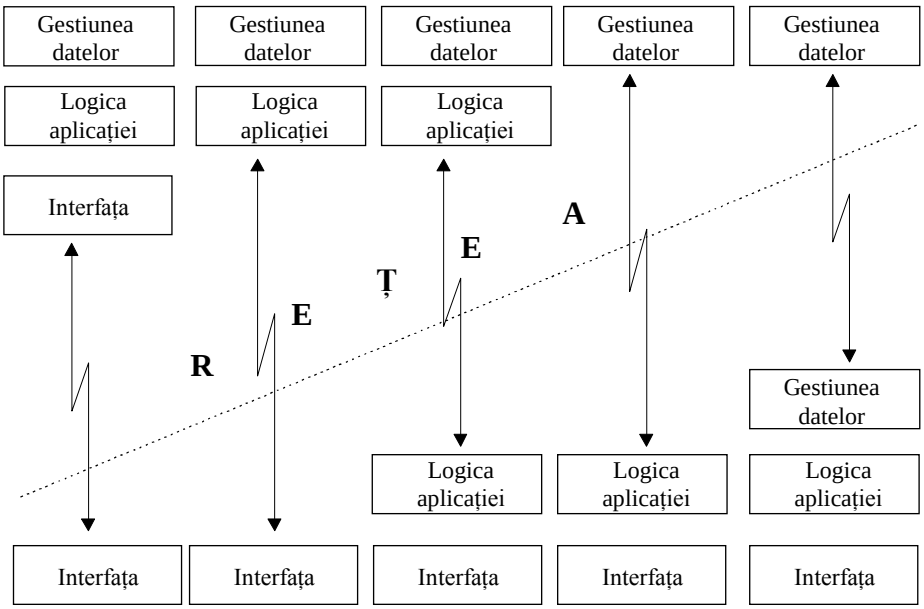
• **Eficientizarea accesului la date**. Serverele de baze de date nu vor mai fi solicitate de un număr mare de cereri de acces, gestiunea cererilor clienților revenind serverelor de aplicații (deci stratului intermediar). În acest mod, clienții nu mai sunt nevoiți să se conecteze direct la baza de date şi, prin urmare, nu vor mai avea nevoie de drivere specifice (ca în cazul arhitecturii pe două straturi).

• **Îmbunătățirea securității**. Componentele din stratul intermediar pot fi gestionate din punctul de vedere al securității printr-o infrastructură centralizată comună, determinând simplificarea şi reducerea costurilor de administrare.

În prezent se manifestă tendința dezvoltării aplicațiilor multistrat, în care pot exista mai mult de trei straturi, atât din punct de vedere logic, cât şi fizic. Acest lucru este posibil datorită apariției unei noi paradigme în dezvoltarea sistemelor informaționale, referită prin sintagma **orientată pe componente**. Această nouă abordare, coroborată cu libertatea în distribuirea componentelor datorită apariției unor protocoale (API) de comunicare specifice, determină orientarea către dezvoltarea de aplicații client/server multistrat.

## CLASIFICAREA MODELELOR ARHITECTURALE CLIENT/SERVER

Proiectarea sistemelor informatice conform tehnologiei client/server trebuie să ia în considerare diferitele tipuri de sisteme client/server. O clasificare a modelelor client/server a fost propusă de Gartner Group, pornind de la cele trei părți funcționale componente ale unei aplicații. Figura de mai jos arată cele cinci tipuri de arhitecturi regrupate sub numele client-server, fiecare tip fiind prezentat în continuare.




**Interfața (prezentarea) distribuită** urmăreşte dispunerea de facilități grafice pe postul client, motiv pentru care acest model mai este referit şi prin "cosmetica aplicației". Concret, acest model presupune adăugarea unei interfețe grafice evoluate la o aplicație centralizată, rezidentă pe un mainframe sau minicalculator, în vederea înlăturării dezavantajelor asociate interfețelor la modul caracter specifice platformelor mari. Aplicația centrală nu este modificată ci numai partea de interfață a aplicației, adică acea parte care selectează datele provenite de la calculatorul central şi le afişează într-un mediu grafic specific microcalculatoarelor.

Operația de transformare a interfeței se poate face de-o manieră mai simplă, fără alterarea succesiunii ecranelor specifice aplicației originale (unui ecran în modul caracter îi corespunde un ecran în modul grafic) fie de-o manieră mai "radicală", în sensul modificării succesiunii dialogurilor şi ecranelor utilizator specifice aplicației originale, însă tot fără a efectua vreo modificare în programele aplicației (unui ecran în modul caracter îi poate corespunde mai multe ecrane în modul grafic, precum şi invers). Deşi aduce unele îmbunătățiri aplicației, modelul interfață distribuită poate fi cu greu încadrat în arhitectura client/server, întru-cât partea server a aplicației rămâne semnificativă, manifestându-se o relație de tip master-slave. Şi totuşi, acest model oferă unele avantaje legate de:

• ameliorarea calității interfeței aplicației;

• conservarea investițiilor anterioare efectuate pentru realizarea aplicației, deoarece programele aplicației nu sunt modificate;

• oferă o soluție intermediară în vederea trecerii la arhitectura client/server.

Deşi o asemenea rezolvare răspunde cerințelor utilizatorilor în materie de interfață grafică, ea nu poate reprezenta decât o soluție temporară, deoarece:

• nu rezolvă problemele de comunicare a datelor, generate de traficul intens de date din rețea (la datele aplicației care tranzitează rețeaua între client şi server se adaugă informațiile tehnice legate de poziția câmpurilor în ecran, controale etc.)

• nu oferă deloc (sau prea puțin) performanțe noi, serverul asigurând în continuare toate prelucrările aplicației.

**Interfața (prezentarea) izolată (deportată)** este ea în care gestiunea interfeței utilizator a aplicației este rezidentă pe platforma client, platformă ce asigură în întregime gestionarea dialogului. Terminalele X reprezintă un exemplu de implementare a acestui model, ele oferind o mare portabilitate a aplicațiilor şi o independență totală față de producători de hard şi soft, putând lucra uşor cu platforme hard şi software eterogene.

Acest model asigură următoarele avantaje:

• îmbunătățește calitatea interfeței aplicației;

• conservă investițiile anterioare efectuate pentru realizarea aplicației prin separarea strictă a interfeței de prelucrările aplicației (logica aplicației);

• determină reducerea substanțială a costurilor, datorită prețului redus al terminalelor X şi uşurința întreținerii acestora.

Totuşi, ca şi în primul caz, nu oferă performanțe deosebite deoarece serverul asigură ansamblul prelucrărilor ceea ce presupune o mare încărcare a rețelei. În plus, terminalele X sunt utilizate doar în mediile UNIX.

**Prelucrări distribuite**, model care presupune repartizarea prelucrărilor aplicației între client şi server. Pe platforma client se regăseşte logica funcțională de bază care apelează serverul pentru executarea unor servicii externe prin lansarea unor cereri ce vor activa prelucrările localizate pe server, numite şi proceduri. Apelarea procedurilor de pe server de către clienți se poate face prin intermediul mecanismului RPC (Remote Program Call). Acest mecanism permite, de exemplu, apelarea de către aplicația client a procedurilor rezidente pe server care, la rândul lor, pot conține una sau mai multe cereri SQL.

Adoptarea acestui model necesită stabilirea unor criterii clare de repartizare a prelucrărilor, ceea ce complică procesul de proiectare a sistemelor informatice. Această problemă constituie una din dificultățile dezvoltării de aplicații conforme acestui model, deoarece rezolvarea ei necesită o bună cunoaştere a echipamentelor şi programelor pe care va fi implementată aplicația, precum şi o mare experiență în dezvoltarea unor astfel de aplicații. În general, se consideră că, cu cât numărul de cereri de accesare a datelor specifice unei proceduri este mai mare şi cu cât procedura este mai complexă, cu atât mai mult se justifică localizarea acelei proceduri pe server. Aceasta deoarece prelucrările sunt mai aproape de locul fizic de stocare a datelor, iar prin rețea vor tranzita numai cererile de apel de la distanță a procedurilor, nu şi datele.

Avantajele principale ale modelului bazat pe prelucrări distribuite constau în reducerea traficului prin rețea şi o repartizare echilibrată a prelucrărilor între client şi server. În schimb, după cum am mai spus, dezvoltarea unor asemenea aplicații este dificilă datorită cunoştințelor numeroase şi a experienței solicitate.

Un exemplu de adoptare a acestui model îl reprezintă aplicațiile care dispun de formulare pentru culegerea datelor şi care implementează diferite operațiuni de prelucrare şi verificare a datelor în cadrul formularelor, pentru a fi transmise datele către server într-o formă consistentă. În acest fel, dialogul interactiv dintre utilizator şi aplicație (în cazul apariției unor erori de culegere, de exemplu) este localizat pe platforma client, reducând astfel costurile şi întârzierile specifice comunicațiilor.

**Gestiunea izolată (deportată) a datelor**, în care platforma client asigură atât gestionarea dialogului cât şi logica aplicației, iar serverul asigură doar gestionarea datelor. În acest caz se realizează o repartizare clară a funcțiilor între client şi server şi se asigură o securitate sporită a datelor. Aplicația client transmite cererile sale de date

serverului, iar acesta din urmă transmite înapoi datele cerute. Toate prelucrările asupra datelor, specifice aplicației, sunt efectuate pe platforma client.

Dezvoltarea aplicațiilor conform acestui model este facilitată nu numai de repartizarea clară a funcțiilor între client şi server ci şi de oferta bogată de produse mature, cum ar fi SGBDR-urile. Astăzi, SGBDR-urile asigură şi controlul integrității datelor din BD, ceea ce reprezintă o facilitate foarte importantă într-un mediu client/server în care mai multe aplicații client pot modifica aceste date. Localizarea controlului integrității datelor în acelaşi loc în care se află datele (pe server) permite consultarea şi actualizarea datelor de către oricare din aplicațiile client în deplină siguranță, precum şi reducerea traficului de rețea (cererile privind controlul integrității numai tranzitează rețeaua, ca în cazul în care controlul integrității ar fi localizat pe platforma client). Introducerea trigger-elor în SGBDR-uri facilitează controlul integrității BD şi gestiunea datelor independent de aplicațiile client.

Modelul gestiunea izolată a datelor se diferențiază de sistemele bazate pe simpla partajare a fişierelor de date, în care pe server sunt stocate numai datele în timp ce serviciile de gestionare a datelor sunt rezidente pe client. Desigur că, în acest caz, traficul în rețea este mult mai mare.

Din cele prezentate anterior se pot desprinde următoarele avantaje ale acestui model:

• este mai uşor de înțeles deoarece funcțiile aplicației sunt clar repartizate între client şi server;

• garantează o securitate şi consistența mai bună a datelor;

• există o ofertă variată de produse bine maturizate.

Între dezavantajele asociate pot fi enumerate:

• nu este adaptat mediilor tranzacționale intensive; deşi SGBDR-urile asigură accesul concurent la date, ele nu suportă decât un număr limitat de utilizatori (câteva sute), caz în care se face apel la maşinile tranzacționale, care au rolul de server frontal al SGBDR.

• traficul în rețea este mai mare decât în cazul modelului bazat pe distribuirea prelucrărilor.

**Gestiunea distribuită a datelor** presupune repartizarea datelor între client şi unul sau mai multe servere. Datele repartizate vor fi gestionate ca un ansamblu logic, fiind numai fizic distribuite. Postul client devine el însuşi server de date şi se creează legături de tip server-server care, de cele mai multe ori presupune o gestiune a datelor într-un mediu eterogen (calculatoare, sisteme de operare, rețele sau SGBD-uri diferite).

Acest model reprezintă în teorie modelul ideal de distribuire deoarece permite combinarea datelor într-o manieră avantajoasă atât pentru unitate (coerența sistemului prin globalizarea resurselor eterogene) cât şi pentru utilizatori (sunt mai aproape de date, iar prelucrările datelor sunt mai rapide). Cu toate că asigură coerența globală a sistemului, în condițiile existenței unor resurse eterogene, şi oferă performanțe sporite, implementarea acestui model este deosebit de complexă, fie şi numai pentru că o cerere SQL trebuie analizată şi rezolvată la nivel global, iar pentru consistența datelor trebuie implementate mecanisme în două sau mai multe faze. La această complexitate se adaugă şi oferta (încă) limitată privind arsenalul de produse necesare pentru implementarea unui asemenea model.

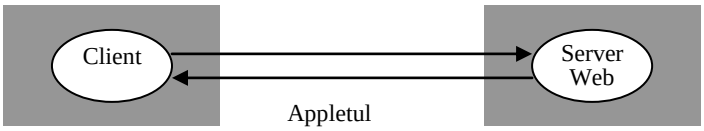
În practică, arhitectura client/server a unei aplicații poate combina mai multe din cele cinci modele prezentate anterior. O asemenea arhitectură poate rezulta prin distribuirea atât a datelor cât şi a prelucrărilor.

## ALTE MODELE CLIENT/SERVER

Dincolo de variantele clasice ale modelului client/server, prezentate anterior, mai pot fi imaginate altele, rezultate prin combinarea facilităţilor tehnologice existente cu cerinţele utilizatorilor şi obiectivele urmărite în dezvoltarea sistemelor distribuite. Câteva dintre aceste variante vor fi prezentate în continuare.

**Programe mobile (mobile cod)**. Applet-urile reprezintă cele mai cunoscute şi utilizate programe mobile. Prin intermediul unui browser (Internet Explorer), un utilizator poate selecta un link către un applet al cărui cod este stocat pe un server web; codul (programul) este încărcat (download) pe calculatorul utilizatorului unde va fi şi executat (vezi figura de mai jos, care prezintă schematic modul de utilizarea a applet-urilor web). Avantajul executării locale a programului constă în îmbunătăţirea comunicării interactive (reducerea timpului de răspuns), atât timp cât ea nu mai înregistrează întârzierile inerente comunicării în reţea.

a) clientul solicită încărcarea applet-ului




b) clientul interacţionează cu applet-ul


Accesarea unui serviciu presupune lansarea în execuţie a unui program care va invoca operaţiile acestuia (ale serviciului). Chiar dacă multe din servicii sunt standardizate pentru a putea fi accesate prin intermediul unor aplicaţii arhicunoscute (web-ul este un exemplu în acest sens), totuşi unele site-uri web utilizează anumite funcţiuni care nu sunt regăsite în browser-ele standard. În această situaţie este necesară încărcarea unor programe adiţionale. De exemplu, un astfel de program poate asigura comunicarea cu serverul atunci când o aplicaţie solicită punerea la curent a utilizatorului cu modificările informaţiilor stocate pe server, imediat ce ele apar. Acest lucru nu poate fi realizat prin intermediul interacţiunilor obişnuite cu server-ul web, care sunt iniţiate de client (clientul nu ar şti când să iniţieze comunicarea). Ar trebui ca interacţiunea dintre client şi server să poată fi iniţiată de server (atunci când au loc modificări ale informaţiilor), problemă ce poate fi rezolvată tocmai prin încărcarea unui program adiţional pe platforma client.

Un exemplu concret: un broker ar putea oferi un serviciu personalizat pentru a înştiinţa clienţii despre schimbarea preţurilor acţiunilor pe piaţă; pentru a beneficia de acest serviciu, fiecare client va trebui să încarce un applet special care să preia actualizările preţurilor de pe serverul brokeru-lui, să le afişeze şi, eventual, să execute cumpărări sau vânzări automate în funcţie de condiţiile stabilite de client şi care sunt stocate pe calculatorul său.

Totuşi, nu trebuie să ne entuziasmăm prea mult în ce priveşte utilizarea applet-urilor şi să nu ignorăm problemele de securitate care pot apare la calculatorul destinaţie. De aceea, browser-ele oferă applet-urilor un acces limitat la resursele locale (ale calculatorului destinaţie).

**Agenții mobili**. Un agent mobil reprezintă o aplicație (include atât programul cât şi datele) care „călătoreşte" în rețea de la un calculator la altul pentru a realiza o sarcină în numele cuiva, cum ar fi colectarea informațiilor şi, eventual, returnarea rezultatelor. În acest scop, un agent mobil poate face numeroase invocări ale resurselor locale ale fiecărui nod pe care îl vizitează (de exemplu accesarea unei baze de date locale). O astfel de problemă ar putea fi rezolvată de un client static, care ar putea invoca resursele necesare de la distanță, dar care ar presupune un volum mare de date de transferat (prin urmare costuri mari de comunicație şi întârzieri în realizarea sarcinii respective). În cazul arhitecturii bazate pe agenți mobili, invocarea resurselor de pe alte calculatoare (noduri) ale rețelei nu se face de la distanță, ci chiar invocarea este inițiată la distanță.

Agenții mobili pot fi utilizați la instalarea şi întreținerea aplicațiilor de pe calculatoarele unei organizații sau pentru compararea prețurilor diferiților producători pentru un anumit produs, prin vizitarea site-urilor fiecărui producător şi efectuarea unor operațiuni de interogare a bazei de date.

Problema securității este şi mai acută decât în cazul applet-urilor. Un agent mobil poate reprezenta o amenințare pentru reursele calculatoarelor pe care le vizitează (de exemplu baza de date). În cazul utilizării lor, calculatorul care îi primeşte în „vizită" trebuie să decidă asupra resurselor care le vor fi disponibile, în funcție de identitatea utilizatorului în numele căruia acționează agentul. Prin urmare, identitatea utilizatorului trebuie să fie inclusă, de o manieră securizată, în agent, alături de program şi datele sale.

**Network computers (NC).** Aceste tipuri de calculatoare au fost introduse ca răspuns la problema gestiunii fişierelor din aplicații şi a întreținerii diferitelor programe locale. Ele solicită un efort şi cunoştințe tehnice pe care de cele mai multe ori utilizatorii nu le au. De asemenea, ele sunt mai ieftine întrucât dispun de resurse mai reduse față de un PC; capacitatea procesorului şi a memoriei pot fi mai reduse. NC-urile încarcă sistemul de operare şi programele de aplicații necesare utilizatorilor de pe un server. În acest mod, aplicațiile vor rula local, însă vor fi gestionate şi întreținute centralizat, pe server. Un alt avantaj al utilizării NC-urilor constă în faptul că utilizatorul se va putea deplasa şi lucra pe orice calculator din rețea, atât timp cât programele de aplicații şi datele sunt rezidente pe server. În cazul în care NC-ul dispune de un hard disc, pe el va fi stocat doar un minim de aplicații, iar partea de disc rămasă disponibilă va fi utilizată ca loc de stocare de tip cache (în care vor fi memorate programele de aplicații şi datele care au fost accesate şi încărcate de pe server recent). În ciuda acestor avantaje, tehnologia NC nu a reprezentat un succes comercial.

**Client slăbănog (thin client)**. În această arhitectură, pe calculatorul utilizatorului este disponibilă o interfață bazată pe ferestre prin care acesta poate executa anumite programe de aplicații pe un alt calculator. Acest model oferă acelaşi avantaje ca în cazul NC-urilor, numai că el nu încarcă programele necesare de pe un server pentru a le executa local, ci le execută chiar pe serverul pe care programele sunt rezidente. Bineînțeles, serverul trebuie să fie un calculator cu capacitate mare de prelucrare şi să permită execuția simultană a numeroase aplicații. De regulă, el va avea mai multe procesoare şi va rula o versiune a sistemului de operare UNIX sau Windows NT. Acest tip de calculator poate fi utilizat în aplicațiile interactive precum CAD (Computer Aidded Design) sau prelucrarea imaginilor, atunci când întârzierile aferente comunicării prin rețea sunt compensate de nevoia de a lucra în echipă. Astfel de implementări sunt: sistemul X-11 pentru UNIX, sistemul Teleporting and Virtual Network Computer (VNC) dezvoltat la laboratoarele AT&T din Cambridge.

**Echipamentele mobile şi reţeaua spontană**. Lumea de astăzi este invadată de dispozitivele de calcul portabile, precum laptop-urile, PDA-urile, telefoanele mobile, camerele de luat vederi digitale etc. Multe dintre aceste echipamente pot fi conectate într-o reţea fără fir, iar prin integrarea lor într-un sistem distribuit se poate pune la dispoziţia utilizatorilor o putere de calcul mobilă. Modalitatea de distribuire care integrează echipamentele mobile şi cele ne-mobile într-o reţea dată poate fi referită prin termenul **reţea spontană**. El este utilizat pentru descrierea aplicaţiilor care implică conectarea dispozitivelor mobile şi a celor ne-mobile în reţea într-o manieră mai informală decât a fost posibil până în prezent. Este evident sprijinul pe care-l oferă în dezvoltarea afacerilor mobile.

# Capitolul 6

# SISTEME CU BAZE DE DATE DISTRIBUITE

Una dintre activitățile cele mai importante ale dezvoltării sistemelor informatice priveşte proiectarea bazei de date. Din punctul de vedere al partajării datelor, există trei alternative de proiectare: **sisteme informatice independente**, fiecare cu propria bază de date, care nu partajează date între ele; sisteme informatice care utilizează o **bază de date centralizată**; sisteme informatice cu **bază de date distribuită**.

Alegerea primei variante este justificată atunci când diferitele aplicații din sistem au cerințe reduse de partajare a datelor. Datele care trebuie totuşi partajate sunt transmise pe hârtie, sub forma rapoartelor, prin fax sau telefon, sau chiar prin email. Desigur că aceste metode de partajare a datelor sunt ineficace. În plus, astfel de aplicații independente sunt întâlnite din ce în ce mai rar astăzi, datorită accentului pus pe integrarea sistemelor informatice din ultimii ani.

Majoritatea sistemelor informatice dezvoltate în ultimii ani au la bază cea de-a doua opțiune. Însă, utilizarea bazelor de date centralizate implică unele deficiențe legate de costurile transmiterii datelor către/dinspre serverul central, timpii de răspuns nesatisfăcători, disponibilitatea limitată a resurselor, etc.

Primele două alternative pot fi văzute ca extremele spectrumului format de soluțiile de partajare a datelor. La mijloc se află soluția axată pe baze de date distribuite care, prin diferitele facilități pe care le oferă, este mai flexibilă şi permite combinarea avantajelor oferite de cele două soluții aflate la extremitatea spectrumului. După cum vom vedea ulterior, putem vorbi de un spectrum de soluții, deoarece tehnologia bazelor de date distribuite, prin facilitățile oferite, permite alegerea din mai multe variante a soluției celei mai potrivite cerințelor sistemului, de la o bază de date pur distribuită până la una complet replicată.

În acest capitol, vom aborda principalele aspecte ale dezvoltării sistemelor informatice cu baze de date distribuite. Astfel, ne vom ocupa de definirea bazelor distribuite, de avantajele pe care le oferă această tehnologie, de principiile care stau la baza distribuirii datelor, de problema mecanismului tranzacțional şi alte aspecte de dezvoltare a aplicațiilor în condițiile utilizării bazelor de date distribuite.

## DEFINIREA BAZELOR DE DATE DISTRIBUITE ŞI AVANTAJELE ACESTORA

O bază de date distribuită este definită ca o colecție de date integrate din punct de vedere logic dar distribuite fizic pe mai multe platforme conectate printr-o rețea, asigurându-se transparența localizării fizice a datelor pentru aplicațiile care le accesează. Aşadar, o bază de date distribuită poate fi considerată ca o bază de date virtuală.

Post defineşte o bază de date distribuită ca un sistem ce constă din multiple baze de date independente care operează pe două sau mai multe calculatoare conenctate în rețea şi care partajează date prin intermediul rețelei. Fiecare bază de date este gestionată de un SGBD independent, care este responsabil pentru menținerea integrității bazei de date. În situații extreme, bazele de date pot fi implementate pe platforme hardware eterogene, ce rulează sisteme de operare diferite şi care utilizează SGBD-uri de la furnizori diferiți. Un astfel de sistem este dificil de proiectat, implementat şi întreținut.

De regulă, sistemele cu baze de date distribuite utilizează acelaşi SGBD pe toate calculatoarele pe care sunt distribuite datele.

Distribuirea datelor poate fi justificată de obţinerea următoarelor avantaje:

• **Depăşirea limitărilor capacităţii de stocare.** Bazele de date voluminoase şi cu un mare număr de accesări pot depăşi capacitatea de stocare şi prelucrare ale serverului sau pot determina performanţe scăzute în accesarea datelor, dacă baza de date ar fi centralizată. Fragmentarea bazei de date în subseturi funcţionale şi stocarea acestora pe platforme diferite, dar care să reprezinte un ansamblu logic, duce la reducerea cerinţelor de prelucrare şi stocare a datelor pentru fiecare platformă pe care este distribuită baza de date.

• **Depăşirea limitărilor specifice mediilor de transmisie**. În cazul în care datele trebuie accesate de la mare distanţă apare problema limitărilor privind lărgimea de bandă a mediilor de transmisie în reţele. De exemplu, astăzi majoritatea LAN-urilor lucrează la 10 Mbps, cu soluţii implementate deja de 100 Mbps sau peste această limită în curând. În unele cazuri aceste performanţe nu sunt suficiente pentru traficul de date solicitat, iar dacă da, serviciile de comunicaţie sunt foarte scumpe. De aceea este mai eficientă distribuirea şi localizarea datelor cât mai aproape posibil de locul de accesare a datelor.

• **Disponibilitatea**. De cele mai multe ori o bază de date deserveşte mai multe aplicaţii. Dacă o bază de date centralizată este parţial distrusă sau inaccesibilă la un moment dat, atunci toate aplicaţiile care o accesează devin inoperabile. Replicarea datelor în conformitate cu cerinţele funcţionale ale aplicaţiilor protejează aplicaţiile împotriva eventualelor căderi ale bazei de date.

• **Reflectarea structurii organizaţionale în arhitectura sistemului**. În funcţie de modul de organizare al firmei este necesară distribuirea controlului şi gestiunii datelor, fiecare departament fiind responsabil pentru conţinutul schemei ce i-a fost alocată. Facilităţile de interogare distribuită permit, dacă este cazul, obţinerea unei imagini consolidate (globale) asupra datelor, ca şi cum ar fi o singură bază de date.

• **Combinarea surselor de date eterogene**. Este, poate, cea mai întâlnită situaţie care justifică distribuirea bazei de date. În acest caz, optarea pentru un sistem informaţional bazat pe distribuirea datelor nu este consecinţa unei decizii strategice ci o reacţie faţă de o situaţie existentă în cadrul firmei. În marile firme este foarte probabil să existe instalate şi utilizate două sau mai multe SGBD-uri diferite, situaţie în care este necesară combinarea datelor din aceste surse diferite astfel încât să ofere utilizatorilor (şi aplicaţiilor) imaginea unei singure baze de date.

• **Scalabilitatea bazelor de date distribuite**. În comparaţie cu bazele de date centralizate, bazele de date distribuite sunt foarte uşor de extins. De exemplu, dacă o companie utilizează o bază de date centralizată, eventuala extindere a activităţii sale într-o nouă regiune geografică, ceea ce solicită spaţiu de stocare şi capacitate de prelucrare suplimentare, ar putea determina înlocuirea serverului bazei de date. În cazul utilizării unei baze de date distribuite, eventuala extindere a activităţii ar putea fi acoperită prin adăugarea unui nou server de baze de date care să realizeze noile operaţiuni, iar echipamentele şi aplicaţiile existente vor rămâne funcţionale.

## OBIECTIVELE SPECIFICE BAZELOR DE DATE DISTRIBUITE

Dezvoltarea aplicaţiilor care utilizează baze de date distribuite reprezintă o sarcină dificilă (sau mai bine spus reprezenta). De aceea, numeroşi specialişti s-au

preocupat de formularea unor reguli care să simplifice dezvoltarea unor astfel de aplicații. În acest sens, C.J. Date a formulat un principiu fundamental pentru bazele de date distribuite: **o bază de date distribuită ar trebui să apară utilizatorilor exact ca o bază de date nedistribuită.** Cu alte cuvinte, utilizatorii dintr-un sistem distribuit ar trebui să vadă baza de date ca şi cum ea ar fi centralizată. Prin **utilizatori** facem referire atât la utilizatorii finali cât şi la dezvoltatorii de aplicații. Pornind de la acest principiu, trebuie făcută diferența dintre o bază de date distribuită şi accesarea de la distanță a mai multor baze de date.

Punerea în practică a acestui principiu solicită atingerea a 12 obiective specifice bazelor de date distribuite, formulate tot de C.J. Date, şi pe care le vom prezenta succint în continuare.

**1. Autonomia locală**. Conform acestui obiectiv, nodurile dintr-un sistem distribuit trebuie să fie autonome. Autonomia locală presupune ca toate operațiile efectuate pe un nod să fie controlate de către acel nod; funcționarea unui nod nu trebuie să depindă de alte noduri (dacă nodul Y încetează să funcționeze la un moment dat din varii motive, atunci funcționarea normală a nodului X nu trebuie să fie afectată). De asemenea, autonomia locală implică şi faptul că datele locale vor fi gestionate local de nodul pe care ele sunt rezidente, independent de faptul că datele respective sunt accesate de la distanță de alte noduri. Astfel, problemele de securitate, asigurarea integrității, stocarea datelor etc. rămân sub controlul nodului local.

În practică, realizarea totală a acestui obiectiv este imposibilă, existând anumite situații în care un nod cedează controlul asupra unor operațiuni unui alt nod. De aceea, acest obiectiv ar putea fi formulat astfel: **nodurile trebuie să fie cât mai autonome posibil**.

**2. Sistemul nu trebuie să se bazeze pe un nod central**. Autonomia locală presupune ca toate nodurile să interacționeze de la egal la egal. Prin urmare, nu trebuie să existe un nod central care să joace rolul de master şi care să gestioneze anumite servicii în mod centralizat, cum ar fi: prelucrarea centralizată a interogărilor, gestiunea centralizată a tranzacțiilor, atribuirea centralizată a numelor. În cazul unei asemenea situații întregul sistem distribuit ar deveni dependent de nodul respectiv, fiind mai vulnerabil. În fapt, acest obiectiv reprezintă un corolar al obiectivului prezentat anterior. Cu toate acestea ele trebuie considerate ca obiective distincte deoarece, chiar dacă autonomia locală completă nu poate fi obținută, acest obiectiv trebuie realizat în mod imperios.

**3. Funcționarea neîntreruptă**. Acest obiectiv este legat de două dintre avantajele generale majore ale sistemelor distribuite: **siguranța în funcționare**, adică probabilitatea ca sistemul să fie funcțional în orice moment, şi **disponibilitatea**, adică probabilitatea ca sistemul să funcționeze continuu o anumită perioadă de timp. Un sistem cu baze de date distribuite va continua să funcționeze în condițiile în care o componentă (de exemplu un nod) încetează să funcționeze.

**4. Independența (transparența) localizării.** Conform acestui obiectiv, utilizatorii nu trebuie să cunoască locul din sistem unde datele sunt stocate, ei putând să interacționeze cu baza de date ca şi cum datele ar fi stocate local (din punct de vedere logic), pe nodul la care sunt conectați. Transparența localizării este necesară pentru a simplifica accesul la baza de date distribuită. În plus, ea permite migrarea datelor de pe un nod pe altul fără a afecta programele sau alte activități de gestionare a datelor. Migrarea datelor este necesară atunci când se doreşte mutarea datelor pe alte noduri din rețea în vederea îmbunătățirii performanțelor sistemului distribuit.

În mediul ORACLE, mecanismul de transparență a localizării poate fi creat prin intermediul view-urilor, sinonimelor şi a procedurilor.

**5. Independența (transparența) fragmentării**. Un sistem care suportă fragmentarea datelor trebuie să asigure şi transparența fragmentării. Aceasta presupune ca utilizatorul să se comporte, cel puțin din punct de vedere logic ca şi cum datele nu ar fi fragmentate. Ea asigură, ca şi transparența localizării, simplificarea sarcinilor dezvoltatorilor de programe. În plus, transparența fragmentării trebuie să permită oricând recombinarea datelor fără să afecteze programele existente. Recombinarea datelor este necesară atunci când se schimbă cerințele de performanță ale sistemului.

Transparența fragmentării este realizată prin intermediul view-urilor puse la dispoziția utilizatorilor în care fragmentele de date sunt recombinate prin operații succesive de joncțiune şi reuniune. Sistemul va determina care fragmente trebuie accesate fizic pentru a răspunde la o anumită cerință a utilizatorilor. Problema actualizării datelor fragmentate şi distribuite fizic pe noduri diferite este similară cu cea a actualizării view-urilor bazate pe operații de joncțiune şi reuniune.

**6. Independența (transparența) replicării**. Replicarea datelor presupune existența uneia sau mai multor copii ale aceluiaşi fragment de date pe diferite noduri. Faptul că datele sunt replicate trebuie să fie transparent utilizatorului, acesta trebuind să se comporte, cel puțin din punct de vedere logic, ca şi cum datele nu ar fi replicate. Ca şi în cazul obiectivelor anterioare, transparența replicării simplifică sarcinile programatorilor şi ale altor utilizatori care accesează baza de date. Ei nu trebuie să se preocupe de faptul că actualizarea datelor de pe un nod trebuie propagată şi pe celelalte noduri unde sunt replicate datele actualizate, sau care nod trebuie accesat pentru a rezolva cerința unui utilizator. În plus, transparența replicării trebuie să permită ştergerea sau crearea unor noi replici ale datelor, ca răspuns la schimbarea cerințelor sistemului, fără să afecteze programele existente.

**7. Prelucrarea distribuită a interogărilor**. Acest obiectiv vizează în primul rând optimizarea interogărilor distribuite. Transparența localizării permite utilizatorilor să creeze şi să execute o interogare ca şi cum toate datele sunt stocate local. În fapt, pentru a genera rezultatul interogării, SGBD-ul va accesa date stocate pe diferite noduri. Pentru rezolvarea interogării pot exista mai multe variante de mutare a datelor de pe un nod pe altul în vederea limitării traficului în rețea şi diminuării timpului de răspuns. De aceea, în cazul interogărilor distribuite trebuie acordată o atenție deosebită optimizării acestora. Conform acestui obiectiv, o interogare distribuită nu trebuie executată ca o interogare locală aplicată asupra tuturor datelor după ce acestea au fost transferate de pe diferite noduri pe nodul de la care a fost inițiată interogarea; ea va fi descompusă în mai multe părți, fiecare fiind executată pe diferite noduri, în funcție de planul de execuție stabilit de mecanismul de optimizare. ORACLE oferă două metode de optimizare a interogărilor, aplicate în special în cazul interogărilor distribuite: metoda bazată pe costuri şi metoda bazată pe reguli.

**8. Gestiunea tranzacțiilor distribuite**. O tranzacție distribuită apare atunci când ea implică actualizarea datelor pe mai multe noduri. Mecanismul tranzacțional priveşte două aspecte majore ale actualizării datelor: controlul refacerii datelor şi controlul accesului concurențial. Ambele aspecte trebuie tratate în mod special în cazul tranzacțiilor distribuite. Astfel, garantarea atomicității unei tranzacții distribuite implică garantarea faptului că toate nodurile implicate vor realiza aceeaşi acțiune – validarea sau anularea tranzacției. Aceeaşi problemă apare şi în legătură cu durabilitatea, în sensul că dacă unul din nodurile implicate a validat sau anulat tranzacția respectivă, atunci

sistemul trebuie să garanteze că toate celelalte noduri vor realiza aceeaşi acţiune, chiar dacă funcţionarea unora dintre noduri este întreruptă temporar. Acest garanţii sunt oferite de sistem prin intermediul mecanismului Two Phase-Commit (2 PC). Asupra mecanismului tranzacţional şi a protocolului 2 PC vom reveni în acest capitol.

Controlul accesului concurent se bazează tot pe mecanismul de blocare, ca şi în cazul sistemelor nedistribuite.

**9. Independenţa hardware**. Acest obiectiv se referă la posibilitatea integrării datelor stocate pe calculatoare de diferite tipuri, toate acţionând ca parteneri în cadrul sistemului. Astfel, ar fi de dorit ca prin intermediul aceluiaşi SGBD să poată fi gestionate date localizate pe calculatoare de tipuri diferite.

**10. Independenţa sistemului de operare**. Acest obiectiv derivă din cel anterior şi presupune ca un SGBD să funcţioneze nu doar pe platforme hardware diferite, ci şi platforme software diferite. SGBD-ul va putea să gestioneze date stocate pe calculatoare diferite care rulează sisteme de operare diferite, precum UNIX, WINDOWS NT, MVS etc.

**11. Independenţa reţelei**. Dacă un sistem poate funcţiona pe platforme hardware şi software diferite, atunci este de dorit ca el să poată funcţiona şi în condiţiile existenţei în sistem a mai multor reţele de comunicaţie de diferite tipuri.

**12. Independenţa sistemului de gestiune a bazei de date**. Acest obiectiv se referă la sistemele eterogene în care datele din baza de date distribuită sunt gestionate de SGBD-uri diferite.

## CÂTEVA ELEMENTE PRIVIND PROIECTAREA BAZELOR DE DATE DISTRIBUITE

Proiectarea bazelor de date distribuite diferă în bună măsură de proiectarea bazelor de date centralizate. În fapt, proiectarea bazelor de date distribuite ridică pe lângă majoritatea problemelor care privesc proiectarea bazelor de date centralizate şi unele probleme specifice, deoarece la distribuirea datelor trebuie luate în considerare unele restricţii de proiectare, precum:

- asigurarea transparenţei localizării datelor pentru aplicaţii şi utilizatori;
- oferirea de performanţe pentru interogările distribuite, în condiţiile unor lărgimi de bandă date;
- gestiunea completă a tranzacţiilor, asigurarea consistenţei actualizărilor distribuite şi a controlului accesului concurent distribuit;
- identificarea administratorilor bazei de date distribuite în cadrul organizaţiei.

Aşadar, proiectarea unei baze de date distribuite presupune parcurgerea următoarelor etape de lucru:

- proiectarea schemei globale,
- proiectarea schemei fizice,
- proiectarea fragmentării şi
- proiectarea alocării.

Activităţile din primele două etape de lucru sunt asemănătoare cu cele care privesc proiectarea bazelor de date centralizate. Specific bazelor de date distribuite sunt problemele legate de fragmentarea şi alocarea datelor. În general, distribuirea datelor se poate realiza sub trei forme:

- **Actualizări distribuite**. Această formă de distribuire permite partiţionarea totală a datelor pe mai multe noduri, iar actualizările care implică mai multe noduri vor

fi distribuite nodurilor corespunzătoare pentru comiterea tranzacţiilor. Astfel se asigură o transparența deplină a localizării datelor pentru aplicațiile care actualizează datele. În acest caz vom avea o bază de date nereplicată (se mai spune bază de date distribuită pură), în sistemul distribuit existând o singură copie a bazei de date.

• **Interogarea distribuită**. Această formă de distribuire reprezintă o soluție mai puțin tehnică şi asigură aplicațiilor transparența localizării datelor, dar numai pentru interogarea lor. În acest caz, serverul de baze de date asigură doar păstrarea informațiilor relative la localizarea datelor, joncțiunea datelor distribuite şi oferirea datelor solicitate în forma dorită.

• **Replicarea datelor**. Această formă reprezintă cea mai simplă soluție de distribuire a datelor, motiv pentru care este şi cea mai des utilizată. Replicarea datelor este procesul prin care serverul de baze de date este responsabil pentru copierea automată a datelor selectate pe mai multe server (după cum este proiectat sistemul informațional), în cazul în care se modifică starea acelor date (sunt actualizate). Această operațiune de copiere se realizează,de regulă, în mod asincron, iar în cazul apariției unor probleme, serverul va încerca din nou până când va reuşi să finalizeze operația de replicare. Toate acestea sunt transparente pentru aplicație, replicarea fiind în fapt o funcție server-server. În acest caz vom avea o bază de date replicată, ea putând fi replicată total sau parțial.

## FRAGMENTAREA DATELOR

În general, fragmentarea presupune partiționarea schemei globale a bazei de date în mai multe părți disjuncte care, evident, se vor numi fragmente. Foarte importantă este **alegerea unității de distribuire**. În acest sens, cea mai simplă cale de distribuire constă în stocarea unei tabele sau a unui grup de tabele pe noduri diferite, caz în care unitatea de distribuire va fi tabela. În general, această soluție nu reprezintă cea mai bună cale de distribuire a datelor, deoarece majoritatea aplicațiilor accesează doar un subset al înregistrărilor sau doar anumite coloane ale unei tabele. De aceea, de regulă operațiunea de fragmentare se referă la partiționarea tabelelor. Fragmentarea unei tabelele presupune partiționarea ei într-un număr minim de fragmente disjuncte, astfel încât fiecare fragment să conțină suficiente informații care să permită reconstruirea tabelei inițiale (ca la normalizarea relației universale).

Distribuirea fragmentelor unei tabele prezintă unele avantaje. De exemplu, timpul de execuție a unei interogări asupra unei tabele de mari dimensiuni poate fi redus prin distribuirea execuției acelei interogări pe mai multe noduri, respectiv nodurile pe care se află fragmentele tabelei. În acest mod se introduce şi paralelismul în execuția interogării.

Fragmentarea poate fi realizată în două moduri de bază: **orizontală** şi **verticală.** Prin combinarea celor două metode se obține o a treia: fragmentarea **mixtă.**

**Fragmentarea orizontală** presupune partiționarea unei tabele în mai multe fragmente, iar fiecare fragment va conține un subset al înregistrărilor. Orice înregistrare se va regăsi cel puțin într-un fragment şi numai într-un singur fragment. Prin urmare un fragment va fi rezultatul unei operațiuni de selecție utilizând un predicat (o condiție pe care trebuie să o satisfacă toate înregistrările din fragment). Reconstrucția tabelei poate fi realizată prin operația de reuniune aplicată asupra tuturor fragmentelor.

**Fragmentarea verticală** a unei tabele presupune divizarea ei în mai multe fragmente, în care fiecare fragment va conține un subset al coloanelor din tabela

respectivă. Fiecare coloană trebuie inclusă în cel puțin un fragment, iar fiecare fragment va include coloanele care formează cheia candidat (toate fragmentele trebuie să aibă aceeaşi cheie candidat). Prin intermediul acestor coloane se va reconstrui tabela originală, prin operația de joncțiune naturală aplicată asupra tuturor fragmentelor. Deoarece coloanele care formează cheia candidat trebuie să fie incluse în toate fragmentele, fragmentarea verticală implică replicarea. Prin fragmentarea verticală se poate realiza replicarea şi a altor coloane decât cele care formează cheia candidat.

În modelul de aplicație prezentat la laborator nu întâlnim fragmentarea verticală. Ea apare atunci când aplicațiile de pe fiecare nod accesează doar un subset al coloanelor unei tabele. Acest gen de aplicații este întâlnit atunci când sistemul informatic (deci şi activitatea din firmă) este distribuit (descentralizată) pe criterii departamentale şi nu regionale.

**Fragmentarea mixtă** presupune aplicarea succesivă a fragmentării orizontale şi verticale, respectiv aplicarea fragmentării verticale la un fragment orizontal sau aplicarea fragmentării orizontale la un fragment vertical. Aceste operații pot fi repetate recursiv generând arbori de fragmentare de mare complexitate. Ordinea în care sunt aplicate fragmentările orizontale şi verticale este foarte importantă, deoarece poate afecta rezultatul final al fragmentării.

În practică, fragmentarea verticală este mai rar aplicată, deoarece distribuirea pe criterii funcționale a sistemului informatic este mai rar întâlnită. Cel mai adesea, distribuirea sistemului informatic se face pe criterii geografice.

## STRATEGIA ALOCĂRII DATELOR

Selectarea strategiei de alocare a datelor depinde de arhitectura sistemului şi de facilitățile oferite de SGBD-ul ales. Există patru abordări de bază:

• **centralizată,** în care toate datele vor fi localizate pe un singur nod. În acest caz apare problema spațiului de stocare limitat şi cea a disponibilității reduse a datelor. În schimb, implementarea acestei soluții este cea mai simplă.

• **partiționată**, care presupune partiționarea bazei de date în fragmente disjuncte şi alocarea fiecărui fragment pe un singur nod. Alegerea acestei soluții poate fi justificată atunci când dimensiunea bazei de date depăşeşte spațiul de stocare a unui singur nod sau performanțele de accesare a datelor sunt îmbunătățite prin creşterea numărului de accese locale.

• **replicarea completă a datelor**, care presupune existența unei copii complete a bazei de date pe toate nodurile din sistem. Replicarea completă a datelor este rar întâlnită în practică, ea fiind aplicată doar atunci când siguranța datelor este critică, dimensiunea bazei de date este redusă, iar ineficiența operațiilor de actualizare poate fi tolerată.

• **replicarea parțială a datelor**, presupune partiționarea bazei de date în fragmente critice şi necritice. Fragmentele necritice vor fi stocate pe un singur server, în timp ce fragmentele critice vor fi replicate pe mai multe noduri, în funcție de cerințele de disponibilitate şi performanță ale sistemului.

În general, prin alocarea datelor se urmăreşte minimizarea costului total, calculat prin însumarea costurilor de comunicație (aferente transmiterii mesajelor şi datelor), a costurilor de prelucrare (aferente utilizării procesorului şi operațiilor de intrare/ieşire) şi a costurilor cu stocarea datelor. De asemenea, se ia în considerare timpul de răspuns al

unei tranzacții, calculat prin însumarea următoarelor elemente: timpul de transmisie prin rețea, timpul de accesare locală a datelor şi timpul de prelucrare locală a datelor.

La proiectarea alocării datelor trebuie respectată o regulă generală: datele trebuie să fie plasate cât mai aproape de locul în care ele vor fi utilizate. În acest scop, pot fi utilizate mai multe metode, în funcție de rezultatul dorit, respectiv alocarea neredundantă sau alocarea redundantă a datelor. În practică mai cunoscute sunt trei metode de alocare:

• **Metoda de determinare a alocării neredundante**, numită şi metoda celei mai bune alegeri (the nonredundant best fit method), constă în evaluarea fiecărei alocări posibile şi alegerea unui singur nod, respectiv a nodului cu beneficiile cele mai mari. Beneficiile vor fi calculate prin luarea în considerare a tuturor operațiilor de interogare şi actualizare.

• **Metoda alocării redundante pe toate nodurile profitabile**, presupune selectarea tuturor nodurilor pentru care alocarea unui fragment va face ca beneficiile să fie mai mari decât costurile aferente. Această metodă va selecta toate nodurile profitabile. Beneficiul aferent unei copii suplimentare pentru fragmentul F la nodul N este calculat ca diferență între timpul de răspuns corespunzător interogării la distanță şi cel corespunzător interogării locale (adică pe nodul respectiv ar exista o copie a fragmentului de date), înmulțită cu frecvența interogărilor asupra fragmentului F inițiate la nodul N. Costul aferent unei copii suplimentare a fragmentului F pe nodul N este calculat prin însumarea timpului total de răspuns corespunzător tuturor operațiunilor de actualizare locală a datelor din fragmentul F inițiate de pe nodul N şi timpul total de răspuns corespunzător tuturor operațiunilor de actualizare la distanță a datelor din fragmentu F de pe nodul N inițiate de pe alte noduri.

• **Metoda alocării progresive a fragmentelor**, presupune construirea inițială a unei soluții neredundante şi apoi introducerea progresivă a copiilor replicate începând cu nodul cel mai profitabil. Mai întâi fiecare fragment va fi alocat pe un nod pe baza valorii maxime a profitului (diferența dintre beneficii şi costuri). Următoarea decizie de alocare va lua în considerare nodul la care a fost plasat în prima etapă un fragment şi valoarea maximă a profitului pentru nodurile rămase. Această procedură va continua succesiv, realizându-se o singură alocare în fiecare etapă, până când toate nodurile rămase sunt neprofitabile.

Ținând cont de considerațiile anterioare, proiectarea unei baze de date distribuite trebuie ia în considerare răspunsurile la întrebările prezentate în tabelul de mai jos. Alocarea redundantă (replicarea) a datelor este justificată doar în condițiile prezentate în ultima coloană din tabel.

| Întrebare | Distribuirea pură | Replicare |
|---|---|---|
| Care sunt cerințele privind: | | |
| • nivelul de consistență a datelor? | Mare | Redus – mediu |
| • costurile cu stocarea datelor? | Mediu-mare | Mic |
| • accesarea partajată a datelor? | Globale | Locale |
| • frecvența de actualizare a tabelelor? | Deseori | Rareori |
| • viteza de realizare a operațiilor de actualizare? | Mare<br>Mare | Mică<br>Mică |
| Care este importanța timpilor planificați de execuție a tranzacțiilor? | Bune-excelente | Slabe |

| | | |
|---|---|---|
| Cum sunt facilitățile pentru accesul concurent şi blocare oferite de SGBD-ul ales?<br>Pot fi evitate accesele partajate? | Nu | Da |

## GESTIUNEA TRANZACȚIILOR ÎN BAZELE DE DATE DISTRIBUITE

**Gestiunea tranzacţiilor** se referă la problematica menținerii bazei de date într-o stare consistentă în condițiile în care accesul la date se face în regim concurent sau în condițiile apariției unor defecte. Prin urmare, mecanismul tranzacțional trebuie să rezolve următoarele două probleme:

• **Controlul concurenței**, adică sincronizarea acceselor astfel încât să fie menținută integritatea bazei de date. De regulă, această problemă este rezolvată prin intermediul mecanismelor de blocare.

• **Rezistența la defecte**, care se referă la tehnicile prin care se asigură atât toleranța sistemului față de apariția unor defecte, cât şi capacitatea de recuperare a acestuia, adică posibilitatea de revenire la o stare consistentă în urma apariției unui defect care a determinat rămânerea bazei de date într-o stare de inconsistență.

O bază de date este într-o stare consistentă dacă datele respectă toate restricțiile de integritate definite asupra lor (restricții privind cheia, restricții de integritate referențială, restricții specifice domeniului problemei). Trecerea bazei de date dintr-o stare în alta are loc atunci când se realizează operațiuni de actualizare a datelor. Evident, orice actualizare asupra bazei de date trebuie să o lase într-o stare consistentă.

## DEFINIȚIA ŞI PROPRIETĂȚILE CONCEPTULUI DE TRANZACȚIE

Prin noțiunea de **tranzacție** se înțelege un ansamblu de operațiuni care sunt executate împreună asupra unei baze de date în vederea realizării unei activități. O tranzacție reprezintă o **unitate logică de prelucrare** care asigură consistența bazei de date. Consistența bazei de date este asigurată independent de faptul că tranzacția a fost executată în mod concurent cu alte tranzacții sau că au apărut disfuncționalități în timpul execuției tranzacției.

O tranzacție simplă poate fi adăugarea unui nou client în baza de date, iar o tranzacție mai complexă poate fi încasarea unei facturi (presupune adăugarea unei înregistrări în tabela de încasări, actualizarea contului prin care s-a efectuat încasarea, precum şi actualizarea soldului clientului respectiv). Pe timpul execuției unei tranzacții, baza de date poate fi într-o stare inconsistentă, însă ea trebuie să fie într-o stare consistentă atât înainte, cât şi după execuția tranzacției.

Pentru ca o tranzacție să garanteze consistența unei baze de date, ea trebuie să satisfacă 4 condiții, referite în literatura de specialitate prin termenul **proprietăți ACID**:

• **Atomicitate** – se referă la faptul că o tranzacție este considerată o unitate elementară de prelucrare, adică execuția unei tranzacții se face pe principiul *totul sau nimic*. O tranzacție este validată numai dacă toate operațiunile care o compun sunt validate, altfel datele trebuie restaurate în starea în care se aflau înainte de începerea tranzacției (tranzacție nu poate fi parțial executată). Rezolvarea tranzacțiilor a căror execuție a fost întreruptă de diverse cauze revine SGBD-ului. După eliminarea cauzei, în funcție de stadiul de execuție în care a rămas tranzacția, SGBD-ul va proceda în unul dintre următoarele două moduri: fie va executa şi restul operațiunilor care compun

tranzacția respectivă, terminând tranzacția cu succes, fie va anula efectele operațiunilor executate până în momentul întreruperii, anulând astfel tranzacția.

• **Consistența** – se referă la corectitudinea sa din punctul de vedere al consistenței datelor. Trecerea de la o stare la alta a datelor, în urma unei tranzacții, nu trebuie să afecteze consistența bazei de date. Tranzacția este corectă dacă transformă baza de date dintr-o stare consistentă într-o altă stare consistentă. Sarcina asigurării acestei proprietăți revine proiectanților şi programatorilor.

• **Izolarea** – presupune ca orice tranzacție să aibă acces doar la stările consistente ale bazei de date. Altfel spus, efectele unei tranzacții nu sunt percepute de o altă tranzacție decât după ce prima tranzacție a fost comisă. De regulă, toate datele solicitate de o tranzacție sunt blocate până în momentul finalizării ei astfel încât o altă tranzacție să nu le poată modifica.

• **Durabilitatea** – se referă la faptul că odată ce tranzacția este validată, efectele sale devin permanente şi vor fi înscrise în baza de date. Efectele unei tranzacții validate vor fi înscrise în baza de date chiar dacă după momentul validării apare un defect care împiedică înscrierea normală a rezultatelor tranzacției în baza de date; această activitate va fi realizată imediat după înlăturarea defecțiunii ivite. Sarcina asigurării acestei proprietăți revine SGBD-ului, iar mecanismul prin care este realizată are la bază conceptul de jurnal. **Jurnalul** este un fişier secvențial în care sunt înregistrate toate operațiunile efectuate de tranzacțiile din sistem. El evidențiază toate operațiunile executate deja, inclusiv starea dinainte şi cea de după a datelor. Dacă tranzacția este complet executată, atunci modificările efectuate asupra datelor vor fi permanentizate şi se spune că *tranzacția a fost comisă*; altfel, sistemul va utiliza jurnalul pentru a restaura baza de date în starea inițială (cea dinaintea începerii tranzacției) şi se spune că *tranzacția a fost anulată*.

## TRANZACȚII DISTRIBUITE

O tranzacție este distribuită dacă ea accesează date gestionate pe mai multe noduri, adică ea accesează date din mai multe tabele aflate pe servere diferite. De exemplu, tranzacția *Adăugare vânzare nouă* este o tranzacție distribuită dacă ea presupune inserarea în tabelele cu facturi şi linii facturi, aflate pe acelaşi nod, actualizarea tabelelor cu clienți şi stocuri, aflate pe alte noduri. Prin urmare, această tranzacție va fi descompusă în trei subtranzacții, transmise celor trei noduri implicate: cele două operațiuni de insert, efectuate pe primul nod; operațiunea de actualizare a soldului clientului, efectuată pe nodul pe care este rezidentă tabela cu clienți; operațiunea de actualizare a stocului produselor de pe factură, efectuată de cel de-al treilea nod, respectiv nodul pe care este stocată tabela cu stocuri.

În exemplul nostru, fiecare din cele trei noduri va fi responsabil cu respectarea locală a proprietăților ACID. În plus, mecanismul tranzacțional în cazul tranzacțiilor distribuite trebuie să garanteze:

• **Atomicitatea globală**, adică toate nodurile implicate în execuția unei tranzacții distribuite să finalizeze în acelaşi mod partea lor de tranzacție (validarea sau anularea). Nu este permis ca unele noduri să valideze partea lor de tranzacție, iar altele să anuleze.

• **Evitarea blocărilor globale**, în care mai multe noduri să fie blocate simultan.

• **Serializarea globală**, care trebuie aplicată tuturor tranzacțiilor, distribuite şi locale.

Ne vom opri în continuare asupra primei probleme. Atomicitatea globală a unei tranzacții este asigurată de SGBD-uri prin intermediul **mecanismului Two Phase Commit (2 PC),** pe care-l vom discuta în cele ce urmează.

În cazul unei tranzacții distribuite, Oracle defineşte un model ierarhic al tranzacției care descrie relațiile dintre nodurile implicate. Acest model este denumit **arborele sesiunii.** În cadrul acestuia, fiecare nod va juca unul sau mai multe din următoarele roluri:

**Client.** Un nod va juca rolul de client atunci când face referire la date situate la un alt nod al bazei de date (acesta se va numi serverul de date).

**Server de date**. Reprezintă nodul care stochează datele referite de un client.

**Coordonator local.** Un nod care face referire la datele stocate pe alte noduri pentru a-şi executa partea sa de tranzacție este numit coordonator local. El este răspunzător de coordonarea tranzacției între nodurile pe care el le referă în mod direct. În acest sens, el va comunica cu aceste noduri pentru primirea şi retransmiterea informațiilor de stare privind tranzacția, transmiterea interogărilor către acestea etc.

**Coordonator global**. Este reprezentat de nodul la care este conectată aplicația care inițiază tranzacția distribuită, şi care reprezintă astfel locul de origine al tranzacției distribuite. El devine părintele sau rădăcina arborelui sesiunii şi realizează următoarele operațiuni: formează arborele sesiunii prin transmiterea comenzilor SQL care formează o tranzacție distribuită către nodurile corespunzătoare (acestea vor fi nodurile referite direct); instruieşte toate nodurile referite direct să pregătească tranzacția; instruieşte nodul de comitere să inițieze comiterea globală; instruieşte toate nodurile să anuleze tranzacția dacă s-a întâmplat ceva.

**Nodul de comitere.** Este nodul care va iniția comiterea sau anularea tranzacției în funcție de ceea ce-i va comunica coordonatorul global. Nodul de comitere trebuie să fie nodul pe care se găsesc datele considerate a fi cele mai critice pentru că el va fi primul care va face comiterea locală. Mai mult, din momentul în care nodul de comitere a realizat comiterea părții de tranzacție care i-a revenit (ceea ce înseamnă implicit că toate celelalte noduri sunt pregătite să comită tranzacția) se consideră că tranzacția distribuită este comisă, chiar dacă unele din nodurile implicate nu au reuşit să facă comiterea. La aceste noduri tranzacția va fi în starea „în dubiu" (in-doubt) până când va dispare problema ivită. Oricum, tranzacția nu va mai putea fi anulată (rollback-uită). Nodul de comitere va fi determinat de către sistem pe baza punctajului atribuit fiecărui server de baze de date. Atribuirea punctajului de comitere se poate face prin parametrul COMMIT_POINT_STRENGTH. Dacă mai multe noduri vor avea acelaşi punctaj, atunci Oracle va desemna unul dintre acestea drept nod de comitere. Oricum, trebuie reținut că nodul de comitere desemnat în cazul unei tranzacții distribuite poate fi diferit de coordonatorul global al acelei tranzacții.

## MECANISMUL DE COMITERE ÎN DOUĂ FAZE (TWO-PHASE COMMIT)

În cazul tranzacțiilor distribuite trebuie asigurată atomicitatea globală adică, fie toate nodurile implicate validează partea lor de tranzacție, fie toate nodurile anulează partea lor din tranzacție. Chiar dacă un nod a efectuat toate operațiunile care compun partea sa din tranzacție şi este pregătit pentru validare, el nu poate lua această decizie unilateral, deoarece este posibil ca alte noduri să nu poată realiza partea lor de tranzacție. Prin urmare, nodurile pregătite pentru validare trebuie să aştepte până când

toate nodurile implicate vor fi pregătite şi apoi vor iniția validarea. Respectarea acestei proprietăți este garantată de nodul coordonator prin inițierea mecanismului 2 PC. Protocolul (sau mecanismul) de comitere în două faze presupune un schimb de mesaje între nodul coordonator şi celelalte noduri implicate în efectuarea tranzacției.

În cazul ORACLE, mecanismul 2 PC presupune parcurgerea a următoarelor 3 faze:

• **Faza de pregătire**, în care coordonatorul global instruieşte toate nodurile implicate să pregătească comiterea tranzacției, cu excepția nodului de comitere. Fiecare nod în parte va verifica dacă poate să comită tranzacția şi va transmite răspunsul său coordonatorului global, rămânând în aşteptarea indicațiilor acestuia dacă să comită sau să anuleze tranzacția în funcție de răspunsurile celorlalte noduri. Oricum, din momentul în care nodul a transmis răspunsul el va garanta fie comiterea tranzacției fie anularea acesteia. Din momentul în care toate nodurile sunt pregătite şi până la comiterea sau anularea modificărilor, se spune că tranzacția este în dubiu.

• **Faza de validare** realizează comiterea propriu-zisă a tranzacției distribuite (dacă toate nodurile au răspuns pozitiv). În această fază, coordonatorul global îi spune nodului de comitere să efectueze comiterea, iar după ce primeşte confirmarea de la acesta va transmite câte un mesaj tuturor nodurilor implicate prin care să le ceară să comită tranzacția, după care va aştepta confirmările din partea acestora. În momentul în care au fost primite toate confirmările, faza de comitere este completă, iar consistența bazei de date globale este asigurată.

• **Faza de uitare**, în care nodul de comitere şi coordonatorul global şterg toate informațiile privind starea tranzacției ce tocmai a fost comisă.

Dacă se întâmplă ca una din cele trei faze să nu poată fi realizată complet, atunci tranzacția respectivă va fi în starea „în dubiu”. O tranzacție distribuită poate ajunge în această stare datorită apariției unor defecte ale unuia din serverele de baze de date implicate, întreruperii conexiunii de rețea între două sau mai multe servere Oracle sau unor erori care privesc logica aplicației utilizatorului (de exemplu apariția unor erori care nu sunt gestionate prin program, iar aplicația se blochează).

Mecanismul 2 PC din Oracle este complet transparent utilizatorilor. Totuşi, intimitățile acestui mecanism trebuie cunoscute pentru a putea interveni în cunoştință de cauză dacă se ivesc unele probleme generate de întreruperea mecanismului.

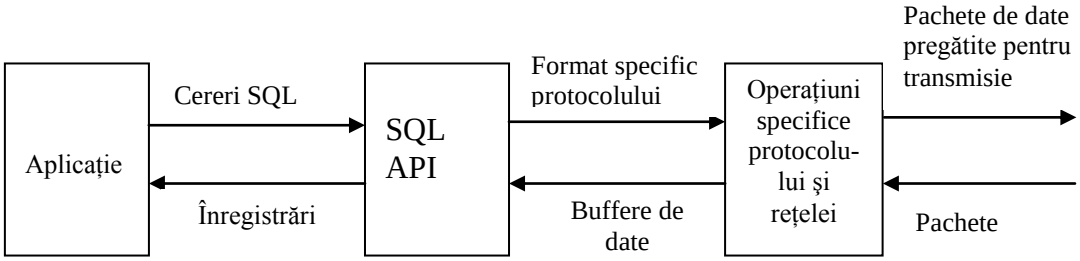
## ACCESAREA BAZELOR DE DATE ÎN APLICAȚIILE CLIENT/SERVER

BD relaționale au devenit astăzi un standard *de facto* pentru stocarea datelor, iar SQL (Structured Query Language) a devenit, mai întâi, standard *de facto* ca limbaj de accesare a datelor din BD, şi standard *de jure* astăzi. Normele limbajului SQL au fost stabilite prima dată de ANSI (1986) şi îmbunătățite de ISO (1989). În 1993 a apărut o nouă versiune, numită SQL/2. În aceste condiții, toți producătorii de SGBDR-uri trebuie să se conformeze acestui standard, ceea ce oferă posibilitatea accesării unor date gestionate de SGBD-uri diferite. Mai mult, unii autori afirmă că dezvoltarea limbajului SQL ca standard se datorează creşterii popularității tehnologiei client/server. În afara acestor standarde mai există alte dialecte, cum sunt cele ale SQL Server (Microsoft), Oracle, Ingres.

Accesarea bazelor de date, ca una din principalele componente ale platformei client, reprezintă o interfață prin care clientul poate transmite cererile de accesare a BD,

cel mai adesea sub forma instrucțiunilor SQL, ele sunt analizate şi verificate, şi se primesc răspunsurile de la server. În general, interogările SQL sunt generate de client şi executate de server, iar unele interogări sunt stocate pe server ca proceduri stocate ce pot fi apelate de aplicațiile client şi executate tot pe server.

Această componentă este formată din două părți, aşa cum rezultă din figura următoare.




Limbajul SQL nu este un limbaj de programare propriu-zis ci este specializat în extragerea, organizarea şi actualizarea datelor din bazele de date relaționale. Majoritatea limbajelor de programare permit apelarea instrucțiunilor SQL din interiorul lor în cadrul programelor. Pentru înglobarea instrucțiunilor SQL în programele scrise în diferite limbaje există mai multe modalități de apelare a SQL API:

**SQL înglobat**. Programatorul poate include instrucțiuni SQL direct în program, la fel de normal ca şi celelalte instrucțiuni. Programul va fi analizat de un precompilator furnizat de producătorul SGBDR-ului. Acest precompilator are rolul de a analiza programul sursă ce urmează a fi preluat de compilator, în vederea înlocuirii unor instrucțiuni cu echivalentul lor în limbajul de ansamblare sau comenzi de apelare a funcțiilor.

Înaintea compilării programului, acesta este prelucrat de un precompilator SQL care caută instrucțiunile SQL înglobate în codul programului sursă. Precompilatorul creează două fişiere de ieşire: primul este un program translatat, în care toate instrucțiunile SQL înglobate au fost înlocuite cu apeluri la rutinele bibliotecii SGBD-ului corespunzător; al doilea este un modul de interogare a bazei de date – DBRM (Database Request Module), care conține instrucțiunile SQL şterse din programul sursă original. Programul sursă translatat este prelucrat de compilator, în timp ce modul DBRM este prelucrat de un program special, numit *bind*, pentru crearea unui *plan al aplicației* (reprezintă o versiune executabilă a instrucțiunilor SQL înglobate în program). La execuția programului, planul aplicației este utilizat pentru accesul la baza de date.

Această abordare oferă anumite avantaje: caracterul simplu şi direct al interfeței program, portabilitatea relativă a programului obținut (majoritatea precompilatoarelor utilizează o sintaxă standard iar cei mai mulți furnizori de SGBDR-uri respectă cel puțin nivelul 1 al standardului ANSI SQL). Principalul dezavantaj este legat de faptul că cele mai multe precompilatoare sunt disponibile doar pentru câteva compilatoare (COBOL, FORTRAN, C).

**Interfață de apelare a funcțiilor (Function call interface)**. Această abordare implică accesarea directă a bibliotecii de funcții puse la dispoziție de furnizor. Prin intermediul funcțiilor apelate se pot transmite instrucțiunile SQL şi primi rezultatele prin apelarea unor funcții adiționale care au rolul de a pune datele în variabilele de program specificate (ex. de astfel de funcții: SQLCONNECT, SQLCOMMAND, SQLEXEC, SQLGETDATA).

Avantajele acestei abordări constau în posibilitatea utilizării în orice limbaj de programare sau instrument de dezvoltare care poate iniția apelul de funcție şi posibilitatea oferită programatorului de a scrie programe de acces la baza de date mai eficient prin apelarea directă la funcțiile din bibliotecă. Dezavantajele sunt legate de complexitatea programului rezultat (se lucrează la nivel jos, spre deosebire de primul caz) şi de faptul că majoritatea bibliotecilor de funcții furnizate sunt specifice unui anumit SGBD.

Utilizarea unei interfețe pentru nivelul de apelare **- CLI (Call Level Interface)**. Unii autori consideră că numai această tehnică este referită prin SQL API. Această modalitate permite introducerea de rutine speciale de interfață cu limbajul SQL pentru accesarea bazei de date, în locul înglobării de instrucțiuni SQL în program.

În ultimul timp se constată preocupări pe linia dezvoltării de standarde asociate cu interfața de apelare a funcțiilor, ceea ce a dus la întocmirea specificațiilor standard privind CLI. Pe baza acestor specificații, Microsoft a dezvoltat standardul API numit **Open Database Connectivity (ODBC)** şi oferit liber distribuitorilor de SGBDR-uri şi altor furnizori preocupați de oferirea pe piață de versiuni specifice de baze de date ale acestei interfețe comune de apelare. Un alt API este **Integrated Database Aplication Programming Interface (IDAPI)** şi a fost propus de un grup de distribuitori de soft, în frunte cu firmele Borland, IBM şi Novell.

Ambele API-uri împart mecanismul de accesare a bazei de date în două module independente din punct de vedere funcțional: primul furnizează aplicațiilor o interfață CLI standardizată, în timp ce cel de-al doilea preia şi converteşte o cerere CLI într-o comandă SQL specifică SGBD-ului respectiv.

De fapt, există două tipuri distincte de interfețe CLI: o interfață API specifică unui SGBDR (şi care este unică), iar cealaltă este o interfață API standard, suportată de mai mulți producători. Interfața API specifică producătorului este, de obicei, calea cea mai eficientă de accesare a BD însă, presupune limitarea la utilizarea SGBDR-ului respectiv.

Instrumentele de dezvoltare de nivel înalt proprietare protejează programatorii de complexitatea interfațării directe cu diferite biblioteci de funcții specifice diferitelor SGBD-uri, prin oferirea unor mecanisme generice de accesare a bazei de date. Mai mult, asemenea mecanisme permit oferirea unei viziuni unice asupra datelor, chiar dacă ele se găsesc pe servere diferite şi gestionate de SGBD-uri diferite, lucru aproape imposibil de realizat cu ajutorul instrumentelor de nivel jos.

## OPTIMIZAREA INTEROGĂRILOR DISTRIBUITE

Optimizarea interogărilor priveşte reducerea timpului de răspuns şi a costurilor de comunicație implicate de execuția interogărilor asupra bazei de date. De regulă, proiectanții de aplicații au un control limitat asupra optimizării interogărilor de vreme ce majoritatea SGBD-urilor dispun de o componentă care rezolvă această problemă în mod automat. De exemplu, Oracle dispune de două metode de optimizare automată: **optimizarea bazată pe costuri (CBO – Cost-Based Optimizer)** şi **optimizarea bazată pe reguli (RBO - Rule-Based Optimizer)**. Aplicând una din aceste metode, Oracle va rescrie interogările distribuite ale utilizatorilor utilizând "collocated inline views", în funcție de datele statistice care privesc tabelele referite şi calculele de optimizare efectuate. Procesul de rescriere a interogărilor este transparent pentru utilizator.

Totuşi, analiza interogărilor asupra bazei de date trebuie luată în considerare în cadrul a două activități specifice dezvoltării aplicațiilor: proiectarea bazei de date şi optimizarea manuală a interogărilor în anumite situații.

Analiza interogărilor este importantă atât în cazul proiectării bazelor de date distribuite cât şi a celor centralizate. În cazul bazelor de date centralizate proiectantul poate modifica schema inițială prin denormalizarea tabelelor. În cazul bazelor de date distribuite, analiza interogărilor poate sprijini proiectantul în luarea unor decizii privind: plasarea tabelelor pe diferite noduri,

• fragmentarea tabelelor în diferite moduri şi plasarea lor pe diferite noduri,

• replicarea tabelelor (sau a fragmentelor) şi plasarea acestor copii pe diferite noduri.

Oricum ar fi, alternativa de proiectare a bazei de date aleasă nu va putea să asigure execuția eficientă a tuturor operațiunilor de interogare; execuția unor interogări va fi încetinită, iar a altora va fi optimizată. De aceea, proiectantul va trebui să ia în considerare frecvența fiecărei operațiuni de interogare, precum şi importanța timpului de răspuns pentru acea operațiune. În plus, trebuie analizate şi tranzacțiile bazei de date, mai ales în cazul în care se optează pentru replicarea datelor.

Optimizarea manuală a interogărilor este necesară în cazul sistemelor cu baze de date eterogene sau datorită limitelor SGBD-ului în optimizarea lor automată. În primul caz, fiecare SGBD dispune de o interfață SQL proprie, motiv pentru care orice interogare a unor date situate pe noduri diferite trebuie descompusă, manual, într-o secvență de comenzi SQL, fiecare comandă fiind executată pe un anumit nod. Pentru cel de-al doilea caz, vom da de exemplu ORACLE: în documentația sa se specifică faptul că optimizarea automată bazată pe costuri este inadecvată în cazul interogărilor distribuite care conțin agregări, subinterogări sau comenzi SQL complexe. În astfel de situații, utilizatorul poate să rescrie el interogările, utilizând "collocated inline views" şi/sau "sugestiile" (hints) (vezi la laborator).

În continuare ne vom opri asupra optimizării interogărilor în condițiile unei scheme date a bazei de date, în situația existenței unei scheme globale, fără a lua în considerare sistemele cu baze de date multiple, limitându-ne doar la optimizarea globală a interogărilor distribuite, fără a lua în considerare optimizarea locală.

Procesul de optimizare a interogărilor distribuite este şi el distribuit, ceea ce implică un schimb de informații între servere. Cu alte cuvinte, procesul de optimizare presupune doi paşi: **optimizarea globală**, moment în care are loc schimbul de informații între servere în vederea transformării interogării distribuite într-o succesiune de operații ce vor fi executate pe noduri diferite, şi **optimizarea locală**, care priveşte optimizarea părții din interogare executată de fiecare nod. De exemplu, în ORACLE acest proces de optimizare este realizat astfel: serverul Oracle va descompune o interogare distribuită în mai multe interogări la distanță care vor fi transmise nodurilor corespunzătoare. Fiecare nod va executa partea sa de interogare (realizând în prealabil optimizarea) şi va transmite rezultatul înapoi nodului de pe care a fost inițiată interogarea distribuită. După primirea rezultatelor de la toate nodurile solicitate, acesta va efectua prelucrările necesare şi va transmite rezultatul final utilizatorului.

Vom pune în evidență procesul de optimizare globală prin intermediul unui exemplu. La evaluarea alternativelor vom lua în considerare doar costurile de comunicație, atât timp cât ele sunt reprezentative atât din punctul de vedere al costurilor implicate, cât şi al timpului de răspuns (costurile de comunicație sunt mult mai mari în cazul interogărilor distribuite decât costurile asociate operațiunilor de accesare a

fişierelor - operaţiuni I/O). Costurile asociate operaţiunilor I/O sunt luate în considerare în faza optimizării locale, ele fiind considerate mult mai mari decât cele care privesc operaţiunile de prelucrare.

Interogările care presupun joncţiunea unor tabele situate pe noduri diferite sunt costisitoare, deoarece ele implică un volum mare de date care trebuie transmise prin reţea în vederea determinării tuplurilor rezultate. Să presupunem că o aplicaţie de pe nodul A lansează o interogare care implică joncţiunea a două tabele situate pe nodurile B, respectiv C. Cele două tabele sunt Clienti şi Facturi.

Pentru a compara planurile alternative de execuţie a interogării, se consideră următoarele date de ipoteză:

• lungimea tuplurilor din cele două tabele este de 25, respectiv 70 bytes, iar lungimea câmpului CODCL este 8 bytes (prin CODCL se va realiza joncţiunea);

• firma are relaţii comerciale cu 52.000 de clienţi, din care anual sunt emise facturi pentru 11.000 clienţi, cu o medie anuală de 5 facturi pe client. Prin urmare,

• tabela Facturi va conţine 55.000 de tupluri;

• rata de transfer a datelor în reţea este de 50.000 bytes/sec.

Prin joncţiunea celor două tabele vor rezulta 55.000 de tupluri, iar lungimea unui tuplu va fi de 87 bytes (25+70-8).

Optimizatorul va lua în considerare următoarele alternative de execuţie:

• transmiterea ambelor tabele pe nodul A şi realizarea joncţiunii pe acel nod;

• transmiterea celei mai mici tabele pe nodul pe care se află cealaltă tabelă (în cazul nostru, transmiterea tabelei Clienti pe nodul C). Pentru claritate, se va lua în calcul şi varianta transmiterii tabelei Facturi pe nodul B.

Rezultatele obţinute în urma calculelor sunt prezentate în tabelul următor. Se observă că cea mai avantajoasă este prima alternativă de execuţie.

| **Alternative de execuţie** | **Volumul de date transferat** | **Timpul aprox. de răspuns** |
|---|---|---|
| 1. Transmiterea ambelor tabele pe nodul A | 52.000*25 + 55.000*70 = 5.150.000 bytes | 103 sec. |
| 2. Transmiterea tabelei Clienti pe nodul C şi a rezultatului joncţiunii către nodul A | 52.000*25 + 55.000*87 = 6.085.000 bytes | 121,7 sec. |
| 3. Transmiterea tabelei Facuri pe nodul B şi a rezultatului joncţiunii către nodul A | 55.000*70 + 55.000*87 = 8.635.000 bytes | 172,7 sec. |

O altă soluţie care ar putea fi luată în considerare presupune transmiterea de la nodul B către nodul C numai a acelor tupluri din tabela Client care vor participa efectiv în realizarea joncţiunii. Apoi, la nodul C se va realiza joncţiunea dintre aceste tupluri şi toate tuplurile din tabela Facturi. Această abordare mai este denumită **planificarea semijoncţiunilor**, şi implică parcurgerea a trei paşi:

• La nodul C se va obţine o tabelă P, rezultată din aplicarea proiecţiei asupra tabelei Facturi, care va conţine doar coloanele implicate în operaţiunea de joncţiune (în cazul nostru doar coloana CODCL). Această tabelă va fi transmisă nodului A (ea va conţine doar codurile clienţilor pentru care există cel puţin o factură).

• La nodul B se va efectua echijoncţiunea dintre tabelele Client şi P, obţinându-se tabela Q. Această tabelă va conţine toate tuplurile din Client care participă la

joncțiune şi numai coloanele necesare în obținerea rezultatului final. Ea va fi transmisă nodului C.

• La nodul C se va efectua echijoncțiunea dintre tabelele Q şi Facturi. Rezultatul obținut va fi transmis nodului A.

Tabela Q, obținută ca rezultat în cel de-al doilea pas, este denumită semijoncțiunea tabelei Client cu Facturi. La o primă vedere, s-ar părea că această soluție este mai costisitoare de vreme ce o joncțiune a fost înlocuită prin două joncțiuni. Însă, prin primul pas al acestei abordări se poate obține o importantă reducere a costurilor de comunicație, deoarece proiecția tabelei Facturi poate să fie mult mai mică decât tabela Facturi, evitându-se astfel transmisia unui volum mare de date prin legătura de comunicație.

Pornind de la aceleaşi date ca în cazul primelor trei variante, în urma calculelor rezultă un volum al datelor de transmis de 5.148.000 bytes (11.000*8 + 11.000*25 + 55.000*87), respectiv un timp de acces de 102,9 sec. (5.148.000/50.000). Comparativ cu variantele anterioare, aceasta ar fi cea mai avantajoasă în cazul problemei noastre.

## Capitolul 7

# LIMBAJUL JAVA

Proiectul Java a început în anul 1990 la firma Sun. Scopul său a fost crearea unui limbaj asemănător cu C++, dar mai simplu, portabil şi mai flexibil, pentru aparatele electrocasnice de larg consum. Caracteristicile noului limbaj sunt:

• simplitate: s-a renunțat la o parte dintre operatori, la pointeri şi la moştenirea multiplă. A fost introdus un colector automat de reziduuri, care rezolvă de-alocarea memoriei fără intervenția programatorului.

• portabilitate: compilatorul generează instrucțiuni ale unei maşini virtuale. Execuția aplicațiilor înseamnă interpretarea acestor instrucțiuni. Singura parte care trebuie deci portată este interpretorul şi o parte din bibliotecile standard care depind de sistem.

• necesită resurse scăzute: interpretorul şi bibliotecile standard pentru legarea dinamică necesită aproximativ 300 kB.

• este orientat obiect: se pot crea clase şi instanțe ale lor, se pot încapsula informațiile, se pot moşteni variabilele şi metodele de la o clasă la alta. Nu există moştenire multiplă, dar s-a introdus conceptul de interfață, care permite definirea comportamentului clasei.

• este distribuit: posedă biblioteci pentru lucrul în rețea, oferind TCP/IP, URL şi încărcarea resurselor la distanță; aplicațiile se pot rula în rețele eterogene.

• este robust: legarea se face la execuție şi informațiile legate de compilare sunt disponibile până la execuție. Indicii tablourilor sunt verificați permanent la execuție.

• este sigur: se verifică operațiile disponibile fiecărui obiect; are sistem de protecție a obiectelor prin declararea lor private/protected/public. În plus, se poate configura mediul de execuție astfel încât să protejeze calculatorul gazdă al aplicației.

• este concurent: permite definirea firelor de execuție şi sincronizarea lor, independent sau conectat la sistemul de operare.

• este dinamic: legarea claselor se face la interpretare şi regăsirea câmpurilor se face prin calificarea numelui clasei cu numele câmpului. Astfel, dacă se modifică superclasa, nu este necesară recompilarea subclaselor.

Java a ajuns cunoscut în 1995, când Netscape l-a licențiat pentru a putea fi folosit în browser-ul său, Navigator. Applet-urile Java (programe cu interfață grafică, înglobate într-un navigator de web) au revoluționat paginile web. Odată cu lansarea versiunii Java 2, limbajul a fost extins şi pentru a putea fi folosit în dezvoltarea generală de programe.

Pachetul JDK (Java Development Kit) al firmei Sun este disponibil gratuit la adresa *http://java.sun.com*. Acesta oferă un set de instrumente în linie de comandă folosite pentru scrierea, compilarea şi testarea programelor Java. Au apărut medii vizuale pentru programare în Java: Borland JavaBuilder, Microsoft Visual Java, Xinox Software Jcreator Pro, etc.

## JAVA – LIMBAJ TOTAL ORIENTAT SPRE OBIECTE

Programarea orientată spre obiecte înseamnă organizarea programelor după modelul în care sunt organizate obiectele în lumea reală. Un program este alcătuit din

elemente denumite **clase**. O clasă poate genera prin instaţiere un **obiect.** Obiectele lucrează împreună în program, fiecare în felul său, pentru rezolvarea problemei propuse. Clasa se defineşte prin cele două aspecte ale sale: stare („cum este") şi comportament („ce face"). Clasele se pot lega, deoarece se pot crea ierarhii de clase, bazate pe relaţia de moştenire: o clasă poate fi sub-clasa alteia, moştenindu-i atât starea cât şi comportamentul. În plus, sub-clasa poate avea noi caracteristici şi noi comportamente. De exemplu, dacă definim clasa `Student` ca sub-clasă a clasei `Persoana`, atunci orice instanţă a clasei `Student` are toate caracteristicile şi comportamentele unei instanţe a clasei `Persoana` (de exemplu, are o adresă de domiciliu) şi altele în plus (de exemplu, are un permis de intrare la biblioteca facultăţii).

Spre deosebire de alte limbaje de programare, Java nu acceptă moştenirea multiplă. Toate clasele derivă dintr-o clasă generică – `Object` – iar moştenirea simplă creează un arbore al tuturor claselor. Această caracteristică a limbajului îl face uşor de folosit, dar restricţionează proiectarea aplicaţiilor. Cazul cel mai des întâlnit este al diferitelor ramuri din arborele de clase care au comportamente asemănătoare. Rezolvarea acestei situaţii este folosirea interfeţelor.

O **interfaţă** este o colecţie de **metode** care descriu un comportament suplimentar pentru o clasă, faţă de ceea ce moşteneşte de la super-clasa sa. O clasă poate implementa mai multe interfeţe, ceea ce rezolvă problema moştenirii multiple. Spre deosebire de clasă, interfaţa conţine numai definiţii abstracte de metode, deci acestea trebuie scrise de programator, pentru a rezolva problema specifică la care lucrează.

Clasele care au elemente comune de concepţie se pot grupa în pachete. De exemplu Java API este setul standard de pachete Java. Din acest set, cele mai folosite pachete sunt: `java.lang`, `java.io`, `java.util`, `java.awt`, `java.swing`, `java.applet`.

Caracteristicile obiectelor şi claselor sunt următoarele: identitatea (fiecare obiect este unic), încapsularea (se pot ascunde o parte din date şi metode), agregarea (se pot încorpora obiecte în obiecte), clasificarea (un obiect este instanţa unei clase), moştenirea (o clasă conţine toate variabilele şi metodele super-clasei) şi polimorfismul (metodele din super-clasă se pot rescrie pentru a implementa alt comportament).

## TIP DE DATĂ, CLASĂ, METODĂ, VARIABILĂ

Toate programele scrise în Java definesc unul sau mai multe tipuri de date, folosind construcţia uneia sau mai multor clase. Metoda este o construcţie de program care oferă mecanismul de realizare a unei acţiuni. De exemplu, în unul dintre cele mai simple programe Java:

```
public class Simulare {
  public static void main(String[] args) {
    System.out.println("Aceasta este o carte de programare
     distribuita.");
  }
}
```

autorul defineşte tipul de dată `Simulare`, prin construirea clasei asociate. Această clasă începe prin cuvântul cheie `class`, urmat de numele definit de programator (`Simulare`) şi de specificarea a ceea ce trebuie să fie acest tip de dată, folosind o pereche de acolade. În acest exemplu, există doar metoda specială `main`, care are o singură acţiune de realizat: afişează pe monitor textul

```
Aceasta este o carte de programare distibuita.
```

Metoda `main` este declarată totdeauna de tipul `void` (adică nu furnizează valori la terminare), iar modificatorii săi sunt `public static` (adică accesul este neprotejat şi metoda aparţine clasei `Simulare`). Parametrul formal al acestei metode este `args`, un tablou de şiruri de caractere. Afişarea se realizează prin invocarea metodei `println` din pachetul `System (java.lang.System)`.

Principiul programării orientate-obiect impune programe care manevrează clase şi obiecte, ce comunică prin mesaje. O clasă conține date şi metode. Obiectul este instanţa unei clase. Prin operația de instanțiere se creează o concretizare a clasei - obiectul - care se poate folosi în program. Clasa este un model abstract, un şablon al tuturor caracteristicilor pe care trebuie să le îndeplinească un obiect al său. Aceste caracteristici se referă la atribute şi comportament. Atributele unei clase se referă la starea obiectului care face parte din clasa respectivă. Atributele se definesc prin variabile, ca în exemplul următor:

```
static class Actiuni implements ActionListener {
      static int S;
      static int T;
      …}
```

În acest exemplu, variabilele `S` şi `T` aparțin clasei `Actiuni`, deci fiecare obiect-instanțiere a acestei clase le posedă. Dacă se defineşte obiectul `act`:

```
static Actiuni act=new Actiuni();
```

atunci se pot folosi în program variabilele `S` şi `T` ale obiectului `act`:

```
scara=170.0/(act.S+act.T); //scara desenului
```

Referitor la comportamentul unei clase, acesta este specificat prin metodele sale, care arată ce pot face obiectele sale în raport cu ele însele sau față de alte obiecte.

## APLICAȚIE, APPLET, SERVLET

Acest prim program, prezentat anterior, este o **aplicație**. După editarea sa, se salvează cu identificatorul `Simulare.java` (în cazul general, `numele_clasei.java`). Etapa următoare constă în compilarea aplicației şi generarea fişierului `Simulare.class`, dacă nu există erori. Interpretarea înseamnă execuția aplicației, prin realizarea efectivă a instrucțiunilor specificate - pe baza datelor specificate - începând cu metoda `main`.

O aplicație poate conține mai multe clase, obligatoriu una dintre ele definind metoda `main`. În acest caz, aplicația se salvează cu numele clasei care conține metoda `main`.

Una dintre caracteristicile limbajului Java este alocarea dinamică, drept metodă de management a memoriei: la interpretare, se alocă pe stivă spațiul necesar obiectelor. Atunci când obiectele de pe stivă nu mai sunt referite, spațiul lor este eliberat în mod automat de către colectorul de reziduuri (garbage collector).

Un **applet** Java este inițiat, afişat şi distrus de o aplicație-gazdă: un browser Internet. Applet-ul nu defineşte metoda main, se compilează ca şi aplicația, dar la interpretare rolul principal revine browser-ului web, care afişează rezultatul în fereastra sa. Unele applet-uri redau o imagine într-o anumită zonă a ferestrei browser-ului, altele generează interfețe grafice, permițând inițierea unor operații cu ajutorul butoanelor de comandă.

Applet-urile lucrează în condiții de restricție a securității, pentru protejarea calculatorului gazdă de acțiuni nedorite (preluare de date, distrugerea sistemului); applet-urile nu au acces la fişierele sau sistemul de intrare/ieşire al gazdei. Dacă se

utilizează un browser care nu suportă Java, utilizatorii nu văd nimic în zona unde ar trebui să se execute applet-ul. Dacă proiectanții paginii doresc, pot oferi un text sau o imagine ca alternativă de afişare în zona respectivă.

Applet-ul este cea mai populară utilizare a limbajului Java, la momentul actual.

Un alt lucru remarcabil referitor la Java este orientarea sa către Internet. Comunicarea în rețea este capacitatea unei aplicații de a stabili conexiuni cu un alt sistem de calcul prin intermediul rețelei. Pachetul `java.net` oferă metode multi-platformă pentru principalele operații de rețea: conectarea, transferul de fişiere, crearea de socluri UNIX. O aplicație care se execută pe un server se numeşte **servlet**.

## FUNDAMENTELE LIMBAJULUI JAVA

**Comentarii.** Tot ce urmează, pe un rând, după caracterele // este interpretat drept comentariu. Dacă se doreşte un comentariu pe mai multe rânduri, tot ce se află între /* şi */ este ignorat la compilare.

**Tipuri de date şi inițializări.** Data se defineşte prin specificarea clasei din care face parte şi numele său. În Java există (ca şi în alte limbaje de programare) tipuri primitive de date şi tipuri definite de programator (derivate). Spre deosebire însă de alte limbaje de programare, Java este total orientat spre obiecte şi singurul tip derivat de date este clasa (chiar şi tablourile sunt obiecte).

```
int varf1, varf2;
double scara;
JLabel label, eti, intro;
```

Variabilele `varf1, varf2` şi `scara` sunt primitive: întregi, respectiv reală în dublă precizie. Variabilele `label`, `eti`, `intro` sunt instanțe ale clasei `JLabel`.

Dacă se doreşte inițializarea unor variabile chiar în momentul definirii lor, acest lucru se poate realiza astfel:

```
int x=4, y=200, z, t;
```

Variabilele `z` şi `t` sunt inițializate implicit cu 0.

Tipurile de date primitive (încorporate în limbaj) sunt următoarele:

- `boolean`, alcătuit din `true` şi `false`.
  Operatorii suportați de variabilele de acest tip sunt: atribuirea `(a=b)`, comparațiile `(a<b, a<=b, a>b, a>=b, a==b, a!=b)`, negația `(!a)`, şi logic `(a&b, a&&b)`, sau logic `(a|b, a||b)`, sau exclusiv `(a^b)`.
- numerice, cu variantele: întregi, reale şi `char`. Operațiile suportate de variabilele numerice sunt atribuirea, comparațiile, operațiile aritmetice şi conversia. Conversia poate fi implicită (când se realizează către un tip mai cuprinzător de date şi nu se pierde informație) sau explicită (când se realizează către un tip mai restrâns de date şi se poate solda cu pierdere de precizie). Tipurile întregi de date sunt: `byte` (reprezentat pe un octet), `short` (reprezentat pe 2 octeți), `int` (reprezentat pe 4 octeți) şi `long` (reprezentat pe 8 octeți). Tipurile reale sunt: `float` (reprezentat în virgulă mobilă pe 4 octeți) şi `double` (reprezentat pe 8 octeți). Tipul `char` se reprezintă intern pe 2 octeți, prin numere întregi pozitive.

Două variabile primitive diferite pot avea valori egale. De exemplu:

```
int x=2, y=2;
```

Egalitatea lor se poate testa folosind o instrucțiune if:

```
if (x==y) System.out.prinln("Sunt egale");
```

```
else System.out.println("Nu sunt egale");
```
În urma execuției acestei secvențe, pe monitor apare mesajul
```
Sunt egale
```
Două variabile derivate sunt egale numai dacă sunt identice (reprezintă acelaşi obiect). De exemplu, două şiruri de aceeaşi lungime şi cu aceleaşi caractere nu sunt egale dacă al doilea, în ordinea creării, s-a creat cu operatorul new. La testarea cu operatorul == se primeşte `false`. Există însă metoda equals care testează egalitatea valorilor unui obiect (a componentelor de stare). Dacă toate sunt egale, metoda returnează `true`. La testarea celor două şiruri descrise mai sus se primeşte deci `true`.

În memorie, variabila de tip primitiv are alocat un anumit număr de octeți, în care este reprezentată. Variabila de tip derivat este o referință către zona de memorie unde se află obiectul ce reprezintă valoarea sa. De exemplu, variabilele `label`, `eti` şi `intro` definite mai sus ca obiecte `JLabel` conțin adresele de memorie unde se află cele trei obiecte.

Cuvântul cheie `null` referă un obiect `null` şi poate fi folosit pentru orice referință a unui obiect.

Mediul de execuție Java (Java RunTime Environment) conține şi un nucleu de clase predefinite, pentru comunicarea între programele Java şi sistemul de operare al gazdei. Clasele uzuale se află în pachetul `java.lang`. Referirea la aceste clase se face prin specificare numelui lor (de exemplu `String`). Referirea la o clasă din alte pachete se face prin specificarea completă a şirului de pachete din care face parte clasa respectivă (de exemplu `java.awt.Color`).

**Expresii**. Expresia este o combinație legală de simboluri, ce reprezintă o valoare. Tipul expresiei este tipul valorii acesteia. Iată exemple de expresii:

`a=b*a` (dacă `a=2` şi `b=3`, atunci `a` devine `6`)

`x=y=z=5` (`x` şi `y` şi `z` devin `5`)

`y=x%y` (dacă `x=17` şi `y=4`, atunci `y` devine `1`, adică restul împărțirii lui 17 la 4)

`m=t++` (dacă `t` este `14`, atunci `m` devine `14` şi apoi `t` creşte cu `1` şi devine `15`)

`x/=y` (dacă `x=20` şi `t=5`, atunci `x` devine `20/5`, adică `4`)

`b=(a<35)?a:30` (dacă `a=15` atunci `b` devine `15`, deoarece `a<35` este adevărat, şi `b` primeşte valoarea lui `a`; dacă `a=41`, atunci `b` devine `30`, deoarece `a<35` este fals).

**Instrucțiuni.** O instrucțiune indică una sau mai multe acțiuni ale calculatorului. Instrucțiunile simple sunt: declarațiile de variabile locale, instrucțiunile-expresie şi instrucțiunea vidă. Exemple de instrucțiuni simple:
```
boolean ultima;
int x, nprobe=0;
S=T=0;
S++;
; //instructiunea vida
```
Instrucțiunile structurate sunt: blocul, instrucțiunile de test şi instrucțiunile repetitive.

Blocul sau secvența constă dintr-un grup de instrucțiuni cuprinse între două caractere acoladă:
```
{int m=0;
 if (a<b) m=2;
 System.out.println(m);
}
```

Variabilele declarate în bloc sunt locale, adică ele există de la momentul declarării până la sfârşitul blocului.

Instrucțiunile de test sunt instrucțiunea `if` şi instrucțiunea `switch`. Instrucțiunea `if` are două variante:

```
if (expresie logică) instrucțiune
if (expresie logică) instrucțiune1 else instrucțiune2
```

şi ea lucrează astfel: dacă *expresie logică* este adevărată, atunci se execută instruțiunea care urmează. Pentru a doua variantă, dacă expresia logică este falsă, atunci se execută *instrucțiune2*. Dacă sunt necesare mai multe prelucrări, atunci se foloseşte secvența. Instrucțiunile cuprinse într-o instrucțiune `if` pot fi alte instrucțiuni `if` (cazul instrucțiunilor imbricate). Exemplu:

```
if (n>0) {// test la numarul de extrageri n
      if (n>20) n=20;
      if (rez[i][14]==1) g.setColor(Color.red);
      g.drawString(Integer.toString(rez[i][j]),5*i,45);
      g.setColor(Color.black);
}
```

Instrucțiunea `switch` realizează o testare generalizată, nu numai cu două opțiuni ca în cazul instrucțiunii `if`. Expresia testată poate fi numai de tip primitiv: `byte, char, short` sau `int`, testul poate fi doar egalitatea, iar valorile testate pot fi de asemenea numai de tip `byte, char, short` sau `int`:

```
switch (expresie) {
case val_1: secventa_1 [break;]
case val_2: secventa_2 [break;]
…
case val_n: secventa_n
[default: secventa]
}
```

Modul de lucru al instrucțiunii `switch` este următorul: *expresie* este comparată pe rând cu fiecare dintre valorile `case`; dacă se găseşte o potrivire, se execută *toate* secvențele care urmează. Dacă se doreşte ieşirea înainte de sfârşitul lui `switch`, se foloseşte instrucțiunea break - aceasta întrerupe execuția în punctul curent şi o reia după prima acoladă închisă. Dacă există `default, secventa` se execută dacă nu se găseşte nici o potrivire. Exemplu:

```
switch (nota) {
case 5: ;
case 6: ;
case 7: System.out.println(„Se putea mai bine”); break;
case 8: ;
case 9: System.out.println(„Bine”);break;
case 10: System.out.println(„Excelent”);
default: System.out.println(„Ai picat examenul”);
}
```

Instrucțiunile repetitive sunt: instrucțiunea `while` (condiționată anterior), instrucțiunea `do-while` (condiționată posterior) şi instrucțiunea `for` (cu contor).

Instrucțiunea `while` are sintaxa:

```
while (condiție) instrucțiune
```

şi execută *instrucțiune* cât timp *condiție* este îndeplinită. Exemplu:

```
while (y*y+x*x<=raza[2]*raza[2]) {
      aleat2=Math.random();
      y=min+(int) Math.floor(aleat2*(max-min));
}
```

Această instrucțiune `while` conține o secvență alcătuită din două instrucțiuni de atribuire. Prima dintre acestea generează o valoare de tip double, aleatoare, din intervalul [0, 1) şi o depune în variabila `aleat2`. A doua instrucțiune atribuie variabilei `y` o valoare calculată pe baza valorii din `aleat2`. Metoda `floor` din clasa `Math` returnează partea întreagă a argumentului său, ca dată `double`, deci este necesară o conversie explicită la tipul `int`. De fapt, expresia pentru `y` permite alegerea unei valori întregi aleatoare în intervalul [min, max).

Instrucțiunea `do-while` execută o *instrucțiune* cât timp *condiție* rămâne adevărată. Sintaxa instrucțiunii este:

`do` *instrucțiune* `while (`*condiție*`);` Exemplu:

```
do {
      x*=i;
      i++;
} while (i<10);
```

Instrucțiunea `for` se foloseşte atunci când o anumită instrucțiune se repetă de un număr specificat de ori; contorul este cel care controlează instrucțiunea `for`. Sintaxa acestei instrucțiuni este:

`for (`*iniţializare*`;` *condiție*`;` *trecere_pas*`)` *instrucțiune*

unde *iniţializare* înseamnă una sau mai multe expresii separate de virgulă, care se execută o singură dată, la început. Dacă în această zonă se declară variabile, acestea sunt locale instrucțiunii for, deci îşi încetează existența după încheierea execuției sale. *condiție* este testul care se realizează pentru încheierea execuției. Dacă valoarea sa este `false`, execuția se opreşte. Dacă valoarea sa este `true`, atunci se execută *instrucțiune*, urmată de *trecere_pas*.*trecere_pas* este alcătuită din una sau mai multe expresii, separate de virgulă. Această etapă asigură modificarea contorului la fiecare iterație. Exemplu:

```
for (int i=0;i<n;i++) {
  n1=n2=n3=n4=n5=n6=0;
  for (int j=0;j<aruncari;j++) {
        aleat=Math.random(); …  }}
```

Principiile programării structurate interzic folosirea etichetelor şi a întreruperii structurilor repetitive. Limbajul Java nu este total structurat, deoarece permite ambele acțiuni. Eticheta este un identificator care precede o instrucțiune, după modelul:

*eticheta*: *instrucțiune*

Forțarea ieşirii din instrucțiunile `switch` sau din cele repetitive se face cu instrucțiunea `break`. În instrucțiuni repetitive, `break` produce părăsirea instrucțiunii curente. Astfel, dacă există instrucțiuni repetitive imbricate, un `break` în cea interioară predă controlul celei exterioare. Întreruperea iterației curente şi trecerea la următoarea iterație se face folosind instrucțiunea `continue`. Instrucțiunile `break` şi `continue` se pot eticheta, pentru a transfera controlul instrucțiunilor etichetate cu etichete identice. Exemple:

```
while (index1<20) {
      if (A[index2]==1) continue;
      B[index2++]= (float) A[index];
}

afara: for (i=1;i<5;i++)
    for (j=1;j<5;j++) {
            System.out.println("i= "+i+" j= "+j);
            if (i+j>4) break afara;
      }
```

Limbajul Java conține o instrucțiune care realizează tratarea excepțiilor. Atunci când programatorul consideră că se poate ajunge la o situație ce conduce la eşuarea aplicației, foloseşte instrucțiunea `try-catch` pentru a trata excepția respectivă şi a specifica modul de revenire în aplicație. O excepție este un obiect (deci o instanță a unei clase), care conține informații despre situația specială respectivă. Dacă într-o secvență din `try` se produce un eveniment special (deci o excepție), obiectul se captează şi se tratează într-o secvență `catch`:

```
try { // protectie la valoare neintreaga
      n=Integer.parseInt(text);
      S=0;
for (int i=0;i<n;i++) {…      }
    iug.label.setText(text1+" numarul intreg "+n);
      iug.eti.setText("Cazuri favorabile: "+S);
      iug.gr.repaint();
}
catch(Exception e1) {
      iug.gr.repaint();
      iug.eti.setText("Cazuri favorabile: 0   ");
      iug.label.setText(" nu este un numar intreg!");
}
```

În exemplu, dacă utilizatorul introduce o valoare care nu este întreagă, atunci programul avertizează prin afişarea mesajului

```
nu este un numar intreg!
```

**Şiruri.** Şirurile de caractere sunt obiecte ale clasei `String`. Definirea şi manevra lor se realizează folosind metodele acestei clase:

```
String prenume = "Adela";
System.out.println("Hello! My name is "+prenume);
int x = prenume.compareTo(cod);
```

În acest exemplu, se defineşte un şir prenume, cu care apoi se afişează mesajul `Hello! My name is Adela` şi apoi se compară şirul memorat în variabila prenume cu şirul din variabila `cod`. Variabila `x` este negativă dacă şirul din prenume precede lexicografic şirul din `cod`, pozitivă dacă şirul din `cod` precede pe cel din prenume şi `0` dacă cele două şiruri sunt identice.

**Tablouri.** Un tablou este o colecție indexată de variabile de acelaşi tip. Nefiind date primitive, tablourile sunt deci obiecte. Ca orice obiect, tabloul este instanță a unei clase. Variabilele care au tablouri ca valori sunt deci referințe către zona de memorie unde este stocat tabloul respectiv. Tablourile cu un indice se declară astfel:

```
int a[];                sau         int[] a;
String tab[];           sau         String[] tab;
```

Crearea unui tablou se poate face prin specificarea elementelor sale sau prin apelarea operatorului `new`:

```
int[] b={1, 2, 3, 4, 5};
String[] tab = new String[20];
```

În primul caz, `b` este un tablou cu 5 elemente de tip `int`, primul element `b[0]` are valoarea 1, al doilea element `b[1]` este 2, etc. (indicii tablourilor încep de la 0). În cel de-al doilea caz, `tab` este un tablou de 20 de şiruri de caractere, toate elementele sale fiind şiruri vide. Aceasta deoarece operatorul `new` inițializează toate pozițiile tablourilor cu 0 pentru date numerice, cu `false` pentru tipul `boolean`, cu şiruri vide pentru `String` şi cu `null` pentru obiecte. După crearea unui tablou, lungimea sa poate fi referită prin variabila asociată lui:

```
int x=b.length; //x devine 5
```

Tablourile multidimensionale sunt tablouri de tablouri: un tablou cu `n` dimensiuni este un tablou de obiecte, fiecare dintre acestea este un tablou cu `n-1` dimensiuni. Exemplu:

```
private static double adjMat[][], d[][];
adjMat = new double [MAX_VERTS][MAX_VERTS];
d = new double [MAX_VERTS][MAX_VERTS];
for (int i=0;i<n;i++)
      d[i][n] = Math.sqrt(((v[i].x-v[n].x) *(v[i].x-v[n].x) +
(v[i].y-v[n].y) * (v[i].y-v[n].y)));
```

Elementele $d_{in}$ din matricea `d` din acest exemplu primesc valorile $d_{in} = \sqrt{(v(i).x - v(n).x)^2 + (v(i).y - v(n).y)^2}$, pentru $1 \le i \le n$.

## ELEMENTE GRAFICE ÎN JAVA

Pentru realizarea elementelor de grafică, platforma Java oferă pachetele de clase `java.awt` (Abstract Windowing Toolkit), care conține elemente grafice din toate platformele (toate sistemele de operare) şi pachetul `javax.swing`, care foloseşte stilul sistemului propriu de operare. În aplicații se pot folosi simultan clase din ambele pachete.

O **interfață grafică** este un obiect grafic structurat, alcătuit din diverse componente. O componentă este un obiect grafic afişabil pe ecran şi care poate interacționa cu utilizatorul. Aceste componente pot fi atomice (etichete, butoane, liste, meniuri, câmpuri de text) sau containere (care conțin alte componente, cum ar fi: panouri, casete de dialog). Aranjarea componentelor într-un container se face folosind un gestionar de poziționare, care este tot un obiect Java, dar este invizibil în interfața grafică.

Clasa `BorderLayout` din pachetul `java.awt` determină 5 regiuni ale containerului (NORTH, SOUTH, WEST, EAST, CENTER), în care se pot aranja diverse componente. Mai există şi alte posibilități: clasa `FlowLayout` aranjează componentele una sub alta, clasa `GridLayout` le aliniază într-o grilă iar clasa `BoxLayout` le aşează pe o singură direcție – orizontal sau vertical.

Culorile folosite într-o interfață grafică sunt obiecte ale clasei `Color` din acelaşi pachet `java.awt`. Culorile uzuale se pot specifica prin numele lor (`Color.blue`), dar se poate edita orice culoare prin specificarea a patru atribute, valori întregi cuprinse între 0 şi 255: componenta roşu, componenta verde, componenta albastru şi transparența. Pentru componente, o valoare de 255 înseamnă culoarea respectivă la maxim; iar 255 pentru transparență înseamnă opacitate.

Componentele grafice sunt caracterizate de **aspect** (cum arată pe ecran), **stare** (valorile câmpurilor lor) şi **comportament** (cum reacționează la acțiunile utilizatorilor, la invocarea de metode).

După crearea containerului, acesta devine vizibil prin invocarea metodei `setVisible(true)`. Scrierea în interfețele grafice necesită metode speciale, fiind asimilată cu desenarea. Pentru scrierea şirurilor de caractere se foloseşte metoda `drawString`, cu trei argumente: şirul care trebuie scris, coordonata pe orizontală (abscisa) şi coordonata pe verticală (ordonata) față de colțul din stânga-sus al panoului curent de desenare. Unitatea de măsură a sistemului de coordonate este pixelul, deci coordonatele trebuie să fie întregi. Cu metodele corespunzătoare, se pot desena dreptunghiuri, arce de cerc, segmente, linii poligonale, elipse şi alte imagini.

Toate comenzile de desenare sunt metode ale clasei `Graphics`. Locul uzual unde se invocă comenzile de desenare este metoda `paint`, deoarece această metodă este specială din următorul punct de vedere: se apelează automat atunci când containerul curent trebuie redesenat, deoarece a fost acoperit de o altă fereastră. Programatorul poate cere redesenarea în mod explicit apelând metoda `repaint`.

## EVENIMENTE

Un **eveniment** este orice modificare a stării dispozitivelor de intrare sau a obiectelor de pe ecran. În Java programarea orientată spre evenimente foloseşte modelul bazat pe delegare. Acest model clasifică obiectele în:

- generatoare de evenimente (surse), care sunt componente ale interfeței grafice;
- captatori de evenimente (ascultători), care captează şi tratează evenimentele;
- evenimente, care pot fi la rândul lor: de fereastră, de mouse, de tastă.

Lucrul acestor categorii de obiecte se descrie astfel: sursa transmite evenimentele generate numai ascultătorilor înregistrați. Cele mai multe programe folosesc ferestre Windows care reacționează la clasicele manevre de minimizare şi închidere. Aceste evenimente sunt captate de metoda `windowClosing`, care face parte dintr-o clasă (AF) care extinde clasa `WindowAdapter`. Legarea ascultătorului de fereastră se realizează prin metoda `addWindowListener`. Câmpurile de text, care sunt singurele care trebuie să genereze evenimente în aplicațiile noastre (despre manevrele ferestrelor am discutat deja) sunt ascultate de clasa `Actiuni`, care implementează interfața `ActionListener`. Această interfață are o singură metodă – `actionPerformed` – care trebuie rescrisă astfel încât să descrie acțiunile care trebuie executate.

## GENERAREA NUMERELOR ALEATOARE ÎN JAVA

Limbajul Java are mai multe facilități destinate generării numerelor aleatoare. Cea mai simplă facilitate este metoda `random()` din clasa `Math`, care se află în pachetul `lang` şi care este identificată în mod standard astfel:

```
Java.lang.Math.random()
```

Întrucât `lang` este biblioteca sistemului, nu este necesar să se folosească cuvântul `lang` în denumirea unor elemente care aparțin acestei biblioteci. Cu alte cuvinte, apelul se mai sus poate fi scris: `Math.random()`.

Când este apelată prima dată metoda `random()`, se crează un generator de numere aleatoare, folosind o valoare inițială bazată pe ora curentă (obținută de la orologiul calculatorului) şi o formulă liniară de congruențe. Acest generator de numere aleatoare este folosit apoi la apelurile ulterioare ale metodei `random()`, pentru a genera un şir de numere aleatoare.

Un alt instrument oferit de limbajul Java pentru generarea numerelor aleatoare este clasa `Random` din pachetul `util` şi care este identificată în mod standard astfel:

```
Java.util.Random
```

Pentru a genera un şir de numere aleatoare, se poate folosi un obiect de clasă `Random`. Această clasă foloseşte o valoare inițială reprezentată pe 48 de biți şi o formulă de congruențe liniare. Crearea unui obiect de clasă `Random` se poate face cu oricare din următorii doi constructori:

`Random()` - acest constructor creează un nou generator de numere aleatoare, folosind o valoare inițială obținută pe baza orei curente;

`Random(long seed)` - acest constructor creează un nou generator de numere aleatoare, folosind o valoare inițială `seed`, dată ca parametru de către programator.

Clasa `Random` mai conține următoarele metode care pot fi folosite la crearea unor programe Java, pentru generarea numerelor aleatoare în diferite formate.

`setSeed(long seed)` - comunică generatorului curent de numere aleatoare valoarea inițială `seed` cu care să înceapă generarea;

`next(int nrbiți)` - generează următorul număr aleator pentru şirul creat de generatorul curent (în format `int`, pe un număr de biți precizat în parametrul `nrbiți`, un număr întreg cuprins între 1 şi 32);

`nextBytes(byte[] octeți)` - generează un tablou de octeți care conține numere aleatoare, dimensiunea tabloului fiind precizată de către utilizator;

`nextInt()` – generează următorul număr aleator de tip `int` (întreg pe 32 de biți) din secvența de valori cu repartiția uniformă pe (0,1) a generatorului curent. Toate cele $2^{32}$ valori posibile sunt generate cu aproximativ aceeaşi probabilitate;

`nextInt(int n)` – generează următorul număr aleator de tip `int` (întreg pe 32 de biți) din secvența de valori cu repartiția uniformă pe [0, n) a generatorului curent. Toate cele n valori posibile sunt generate cu aproximativ aceeaşi probabilitate;

`nextLong()` - generează următorul număr aleator de tip `long` (întreg pe 64 de biți) din secvența de valori cu repartiția uniformă pe (0,1) a generatorului curent. Toate cele $2^{64}$ valori posibile sunt generate cu aproximativ aceeaşi probabilitate;

`nextBoolean()` - generează următoarea valoare aleatoare de tip `bolean` (`true` sau `false`) cu repartiția uniformă din secvența de valori a generatorului curent. Valorile `true` şi `false` sunt generate cu aproximativ aceeaşi probabilitate;

`nextFloat()` – generează următorul număr aleator de tip `float` (real simplă precizie) din secvența de valori cu repartiția uniformă pe [0.0, 1.0) a generatorului curent. Toate cele $2^{24}$ valori posibile de forma $m*2^{-24}$, unde $m$ este un întreg pozitiv mai mic decât $2^{24}$, sunt generate cu aproximativ aceeaşi probabilitate;

`nextDouble()` – generează următorul număr aleator de tip `double` (real dublă precizie) din secvența de valori cu repartiția uniformă pe [0.0, 1.0) a generatorului curent. Toate cele $2^{53}$ valori posibile de forma $m*2^{-53}$, unde $m$ este un întreg pozitiv mai mic decât $2^{53}$, sunt generate cu aproximativ aceeaşi probabilitate;

`nextGaussian()` – generează următoarea valoare aleatoare de tip `double` (real dublă precizie), pe baza unei repartiții normale (Gaussiene) standard cu media 0.0 şi abaterea standard 1.0, din secvența de valori a generatorului curent de numere aleatoare.

## BIBLIOGRAFIE


1. Athanasiu I. - **Java ca limbaj pentru programarea distribuită**, Matrix Rom, 2000
2. Baltac V. (coordonator) - **Calculatoarele Electronice, Grafica Interactivă şi Prelucrarea Imaginilor**, Editura Tehnică, Bucureşti, 1986
3. Boian F.M. - **Programarea distribuită în Internet**, Ed. Albastră, Cluj-Napoca, 1999.
4. Bumbaru S. – **Curs practic de programare orientată pe obiecte în limbajul Java**, Universitatea Dunărea de Jos, Galaţi, 2000
5. Chiorean I. - **Calcul paralel. Fundamente**, Ed. Microinformatica, 1995
6. Cormen T., Leiserson C. Rivest R. – **Introducere în algoritmi**, Computer Libris Agora, 2000
7. Craus M. – **Algoritmi pentru prelucrări paralele**, Editura "Gh.Asachi", Iaşi, 2002
8. Cristea V. - **Algoritmi de prelucrare paralelă**, Ed. Matrix Rom, 2005
9. Croitoru C. - **Introducere in proiectarea algoritmilor paraleli**, Ed. Matrix Rom, 2004
10. Dollinger R. - **Baze de date şi gestiunea tranzacţiilor**, Editura Albastră, Cluj-Napoca, 1999
11. Gorunescu F., Prodan A. – **Modelare stochastică şi simulare**, Editura Albastră, Cluj Napoca, 2001
12. Grigoraş D. – **Calculul Paralel: De la sisteme la programarea aplicaţiilor**, Computer Libris Agora, 2000
13. Hockney R.W., Jesshope C.R. - **Calculatoare paralele. Arhitectura, programare, algoritmi**, Ed. Tehnică, 1991
14. Jalobeanu M. - **Internet, Informare şi Instruire: Paşi în lumea comunicaţiilor**, Ed. Promedia Plus, Cluj-Napoca, 1995.
15. Jalobeanu M. - **Acces în Internet -Poşta electronică şi transferul de fişiere**, Ed. Promedia Plus, Cluj-Napoca, 1996.
16. Jalobeanu M. - **WWW în învăţământ: Instruirea prin Internet, Cum căutăm şi Cum publicăm pe Web**, Ed. CCD, Cluj-Napoca, 2001.
17. Lungu, I., Bodea, C., Bădescu, G., Ioniţă, C. - **Baze de date. Organizare, proiectare şi implementare**, Editura ALL Educational, Bucureşti, 1995
18. Petcu D. - **Procesare paralelă**, Editura Eubeea, Colecţia Informatica, Timişoara, 2001.
19. Petcu D., Negru V. - **Procesare distribuită**, Editura Universităţii de Vest, Seria Alef, Timişoara, 2002
20. Petcu D. - **Algoritmi paraleli**, Tipografia Universităţii Timişoara, 1994
21. * * * ORACLE8i, Replication
22. * * * ORACLE8i, Replication Management API Reference
23. * * * ORACLE8i, Distributed Database Systems

## RESURSE WEB


1. http://www-users.cs.umn.edu/~karypis/parbook/ - **Introduction to Parallel Computing**, V. Kumar, A. Grama, A. Gupta, G. Karypis, Benjamin-Cummings
2. http://www.cs.vu.nl/~ast/books/ds1/ - **Distributed systems. Principles and paradigms**, A. Tannenbaum
3. http://www-unix.mcs.anl.gov/dbpp/ - **Design and building parallel programs: Concepts and Tools for Parallel Software Engineering**, Ian Foster
4. http://www.cs.usfca.edu/mpi/ - **Parallel Programming with MPI**, Peter Pacheco, Morgan Kaufmann, 1996
5. http://www.cs.brown.edu/courses/cs176/ - **Introduction to Distributed Computing**
6. http://relis.uvvg.ro/~jalobean/Cursuri/Paralel/algorithms.html - **Internet Parallel Computing Archive**
7. http://www.dcd.uaic.ro/default.php?t=site&pgid=58 - **Serviciul DNS la D.C.D. şi RoEduNet**
8. http://www.cerfacs.fr/algor/ - **The Parallel Algorithms Project**
9. http://www.ornl.gov/sci/techresources/Human_Genome/home.shtml - **Human Genome Project Information**