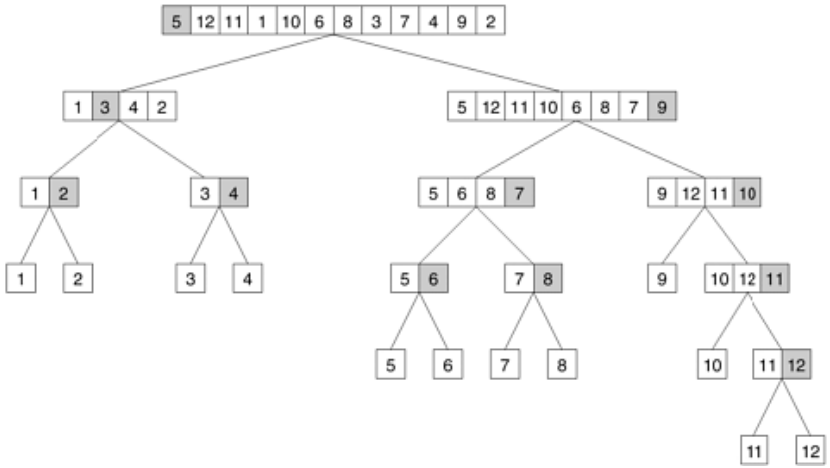
X. Descompunere si orchestrare

Continut

- Descompunere:
  - recursiva,
  - de date,
  - exploratorie,
  - speculativa
  - hibrida
- Orchestrare in
  - Paralelismul datelor,
  - Spatiul de adresare partajat
  - Modelul de transmitere a mesajelor

Descompunere recursiva

- Este o metodă pentru inducerea concurentei în probleme care pot fi rezolvate folosind strategia de împărțire și cucerire.
  - O problemă este rezolvată prin împărțirea acesteia într-un set de subprobleme independente.
  - Fiecare dintre aceste subprobleme este rezolvat prin aplicarea recursivă a unei diviziuni similare în subprobleme mai mici, urmată de o combinație a rezultatelor lor.
  - Strategia împărțire-și-cucerire este natural concurentă, deoarece diferite subprobleme pot fi rezolvate concomitent.
- Exemplu: Quicksort.
  - Sortarea unei secvențe A din n elemente folosind algoritmul de tip quicksort utilizat în mod obișnuit.
  - Definim o sarcină ca fiind activitatea de partiționare a unei subsecvențe date.
  - Fig reprezintă, de asemenea, graful sarcinilor pentru problemă.



Descmpunere recursiva: gasirea minimului

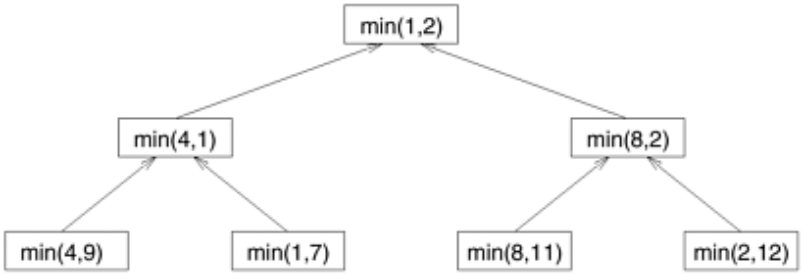
- Uneori, este posibil să restructurați un calcul pentru a face posibilă descompunerea recursivă, chiar dacă algoritmul utilizat frecvent pentru problemă nu se bazează pe strategia de împărțire-și-cucerire.
- Exemple: problema găsirii elementului minim într-o secvență neordonată A din n elemente.
  - Algoritmul serial pentru rezolvarea acestei probleme scanează întreaga secvență A, înregistrând la fiecare pas elementul minim găsit până acum.
  - Odată ce restructurăm acest calcul ca un algoritm de împărțire-și-cucerire, putem folosi descompunerea recursivă pentru a extrage concurenta.
  - Un program recursiv pentru găsirea minimului într-o serie de numere A de lungime n:

```
procedure RECURSIVE_MIN (A,
      n)
begin
if (n = 1) then
min := A[0];
else
lmin := RECURSIVE_MIN (A, n/2);
rmin := RECURSIVE_MIN
      (&(A[n/2]), n - n/2);
if (lmin < rmin) then
min := lmin;
else
min := rmin;
endelse;
endelse;
return min;
end RECURSIVE_MIN
```

Descmpunere recursiva: gasirea minimului

- In acest algoritm,
  - *A se împarte în două subsecvențe, fiecare cu dimensiunea n / 2*,
  - găsirea minimul pentru fiecare dintre aceste subsecvențe efectuând un apel recursiv.
  - acum elementul minim general se găsește prin selectarea minimului acestor două subsecvențe.
  - recursivitatea se încheie atunci când rămâne un singur element în fiecare subsecvență.
- Este ușor să construim un graf de dependență a sarcinilor pentru această problemă.
- Fig. ilustrează un graf de dependență a sarcinilor pentru găsirea minimului de opt numere în care fiecărei sarcini i se atribuie găsirea minimului a două numere.

- Graf de dependență a sarcinilor pentru găsirea numărului minim în secvența {4, 9, 1, 7, 8, 11, 2, 12}.



Descompunerea datelor

- Este o metodă puternică și frecvent utilizată pentru obținerea concurenței în algoritmi care operează pe structuri de date mari.
- Descompunerea calculelor se face în doi pași.
  1. Datele pe care sunt efectuate calculele sunt partiționate,
  2. Această partiționare a datelor este utilizată pentru a induce o partiționare a calculelor în sarcini.
- Operațiunile pe care le realizează aceste sarcini pe diferite partiții de date sunt de obicei similare (de exemplu, înmulțirea matricei care urmează) sau sunt alese dintr-un set mic de operații (de exemplu, factorizarea LU).
- Trebuie să fie explorate și evaluate toate modalitățile posibile de partiționare a datelor și să fie determinat care dintre acestea obține o descompunere de calcul naturală și eficientă.
- ***Partitionare datelor de ieșire***.
  - În multe calcule, fiecare element al ieșirii poate fi calculat independent de alții ca funcție a intrării.
  - În astfel de calcule, o partiționare a datelor de ieșire induce automat o descompunere a problemelor în sarcini,
    - fiecărei sarcini i se atribuie munca de calcul a unei porțiuni din ieșire.

Ex. 1 pentru partitionarea datelor de iesire

- **Multiplicarea matricelor**
- Fie problema înmulțirii a două matrice A și B de dimensiune n x n pentru a obține o matrice C.
- Fig. prezintă o descompunere a acestei probleme în patru sarcini.
- Fiecare matrice este considerată a fi compusă din 4 blocuri sau sub-matrice definite prin împărțirea fiecărui dim. din matrice în jumătate.
- Cele 4 submatrici ale lui C, aproximativ de dimensiunea n / 2 x n / 2 fiecare, sunt apoi calculate independent de 4 sarcini ca sume ale produselor corespunzătoare ale submatricelor A și B.

(a) Partitionarea matricilor de intrare și ieșire în 2 x 2 submatrici.

(b) O descompunere a înmulțirii matricei în patru sarcini bazate pe împărțirea matricelor

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

(a)

Task 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$
Task 2: $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$
Task 3: $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$
Task 4: $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

(b)

Ex. 1 pentru partitionarea datelor de iesire

- Majoritatea alg. matriceali, incluzând înmulțirea matrice-vector și înmulțirea matrice-matrice, pot fi formulate în termeni de operații cu matrice bloc.
  - Matricea este privită ca fiind compusă din blocuri sau submatrici și op. aritmetice scalare pe elementele sale sunt înlocuite cu op. matriceale echivalente pe blocuri.
- Descompunerea datelor este distinctă de descompunerea calculului în sarcini.
  - Deși cele două sunt adesea legate, o descompunere dată de date nu are ca rezultat o descompunere unică în sarcini.
  - Exemplu, Fig. arată două descompuneri ale înmulțirii matricei, fiecare în opt sarcini, care corespund aceleiași descompuneri de date ca în fig.

Decomposition I	**Decomposition II**
Task 1: $C_{1,1} = A_{1,1}B_{1,1}$	Task 1: $C_{1,1} = A_{1,1}B_{1,1}$
Task 2: $C_{1,1} = C_{1,1} + A_{1,2}B_{2,1}$	Task 2: $C_{1,1} = C_{1,1} + A_{1,2}B_{2,1}$
Task 3: $C_{1,2} = A_{1,1}B_{1,2}$	Task 3: $C_{1,2} = A_{1,2}B_{2,2}$
Task 4: $C_{1,2} = C_{1,2} + A_{1,2}B_{2,2}$	Task 4: $C_{1,2} = C_{1,2} + A_{1,1}B_{1,2}$
Task 5: $C_{2,1} = A_{2,1}B_{1,1}$	Task 5: $C_{2,1} = A_{2,2}B_{2,1}$
Task 6: $C_{2,1} = C_{2,1} + A_{2,2}B_{2,1}$	Task 6: $C_{2,1} = C_{2,1} + A_{2,1}B_{1,1}$
Task 7: $C_{2,2} = A_{2,1}B_{1,2}$	Task 7: $C_{2,2} = A_{2,1}B_{1,2}$
Task 8: $C_{2,2} = C_{2,2} + A_{2,2}B_{2,2}$	Task 8: $C_{2,2} = C_{2,2} + A_{2,2}B_{2,2}$

Ex. 2 pentru partitionarea datelor de iesire

- **Calcularea frecvențelor seturilor de articole dintr-o bază de date de tranzacții.**
- Fiecare tranzacție și set de articole conține un număr mic. articole, dintr-un set de articole.

Ex.: Bază de date a magazinelor alimentare cu vânzări de către clienți, fiecare tranzacție fiind o listă individuală de cumpărături a unui cumpărător și un set de articole ar putea fi un grup de articole din magazin.

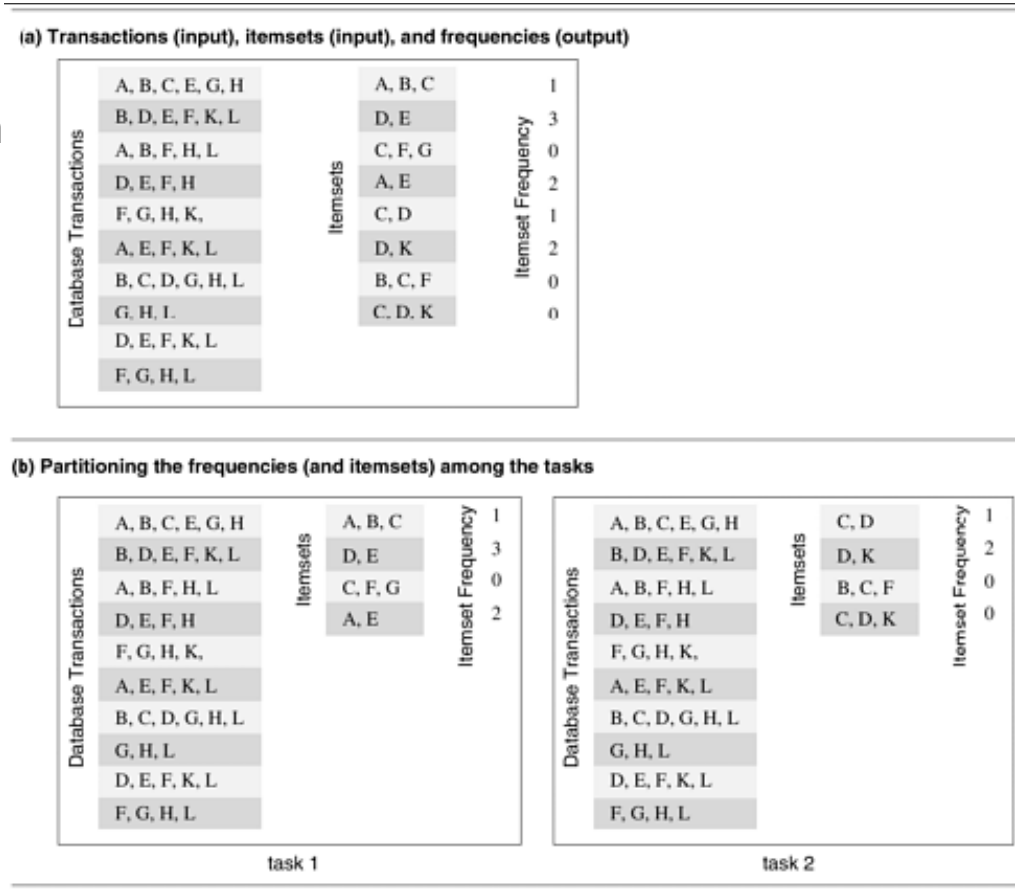
Dacă magazinul dorește să afle câți clienți au cumpărat fiecare dintre grupurile de articole desemnate, atunci va trebui să găsească nr. ori în care fiecare set de articole apare în toate tranzacțiile (numărul de tranzacții nr. al cărui set de articole este un subset)

Fig. (a) arata un exemplu

- Baza de date prezentată constă în 10 tranzacții,
- Ne interesează să calculăm frecvența celor 8 seturi de articole prezentate în a doua coloană.
- Frecvențele reale ale acestor seturi din baza de date (ieșire) sunt afișate în a treia coloană.
- De exemplu, itemet {D, K} apare de două ori, o dată în a doua și o dată în a noua tranzacție.

Fig. (b) arată modul în care calculul frecvențelor seturilor poate fi descompus în 2 sarcini:

- prin împărțirea ieșirii în 2 părți și având fiecare sarcină calculează jumătate din frecvențe.

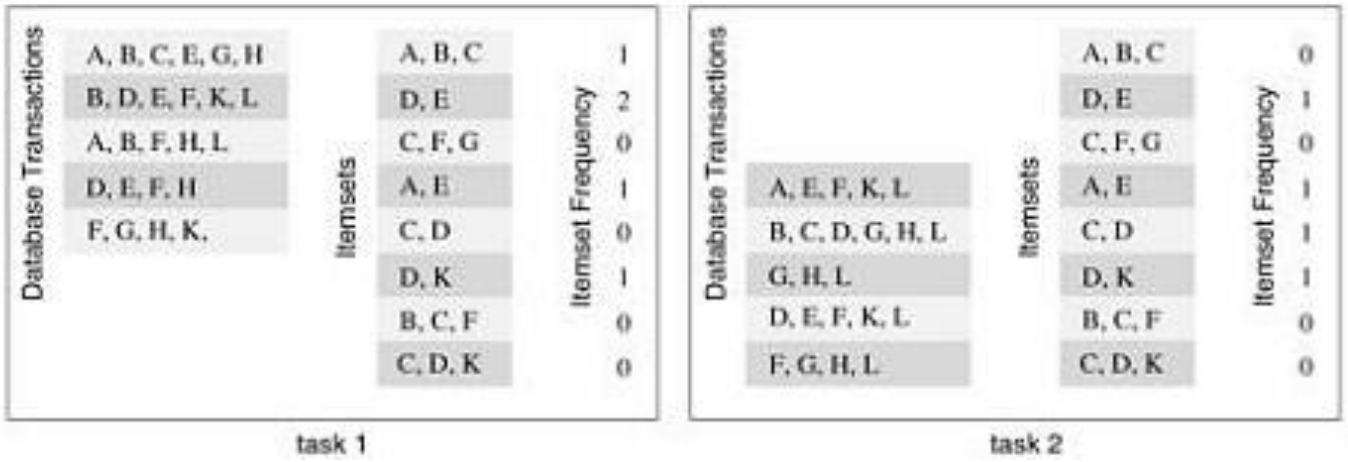


Partitionarea datelor de intrare

- Observație: Partajarea datelor de ieșire poate fi efectuată numai dacă fiecare ieșire poate fi calculată în mod natural în funcție de intrare.
  - În mai mulți algoritmi, nu este posibil sau de dorit să se partajeze datele de ieșire.
  - De exemplu:
    - În timp ce găsiți minimul, maximul sau suma unui set de numere, ieșirea este o singură valoare necunoscută.
    - Într-un algoritm de sortare, elementele individuale ale ieșirii nu pot fi determinate în mod eficient izolate.
- Uneori este posibil să fie partitionate datele de intrare, apoi să fie folosiți această partiție pentru a induce concurență.
  - O sarcină este creată pentru fiecare partiție a datelor de intrare și această sarcină realizează cât mai multe calcule cu ajutorul acestor date locale.
  - Soluțiile pentru sarcinile induse de partițiile de intrare pot să nu rezolve direct problema inițială.
    - În astfel de cazuri, este necesar un calcul de urmărire pentru a combina rezultatele.
- Exemplu: găsirea sumei de N numere folosind p procese ($N > p$):
  - Putem partiționa intrarea în p subseturi de dimensiuni aproape egale.
  - Fiecare sarcină calculează apoi suma numerelor dintr-unul dintre subseturi.
  - În cele din urmă, p rezultate parțiale pot fi adunate pentru a obține rezultatul final.

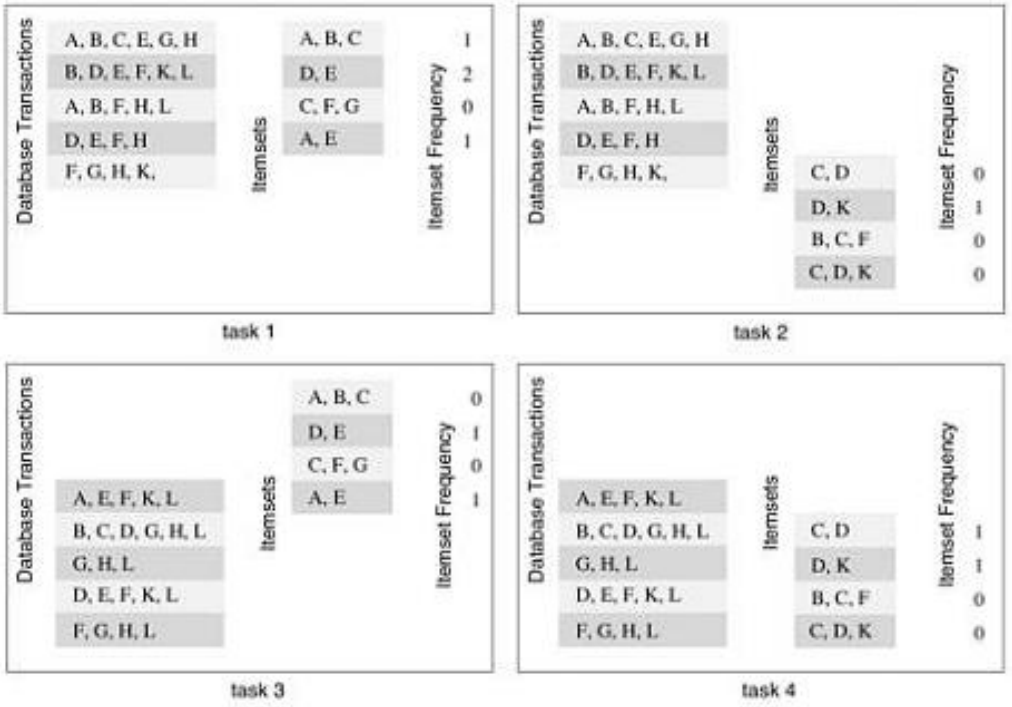
Exemplu pentru partitonarea datelor de intrare

- Problema calculării frecvenței unui set de articole poate fi, de asemenea, descompusa pe baza unei partiționări a datelor de intrare.
- Fig. prezintă o descompunere bazată pe o partiționare a setului de tranzacții de intrare.
  - Fiecare dintre cele două sarcini calculează frecvențele tuturor seturilor din subsetul său de tranzacții.
  - Cele două seturi de frecvențe, care sunt ieșirile independente ale celor două sarcini, reprezintă rezultate intermediare.
  - Combinarea rezultatelor intermediare prin adunarea în perechi dă rezultatul final.



Partitionarea atât a datelor de intrare cât și a celor de ieșire

- În unele cazuri, în care este posibilă partitionarea datelor de ieșire, partiționarea datelor de intrare poate oferi o concurență suplimentară.
- Exemplu: luați în considerare descompunerea în 4 moduri prezentată în Fig. pentru calcularea frecvențelor setului de elemente.
  - atât setul de tranzacții, cât și frecvențele sunt împărțite în 2 părți, iar una dintre cele 4 combinații posibile este atribuită fiecăreia dintre cele patru sarcini.
  - Fiecare sarcină calculează apoi un set local de frecvențe.
  - În cele din urmă, rezultatele sarcinilor 1 și 3 sunt adunate, la fel ca rezultatele sarcinilor 2 și 4.



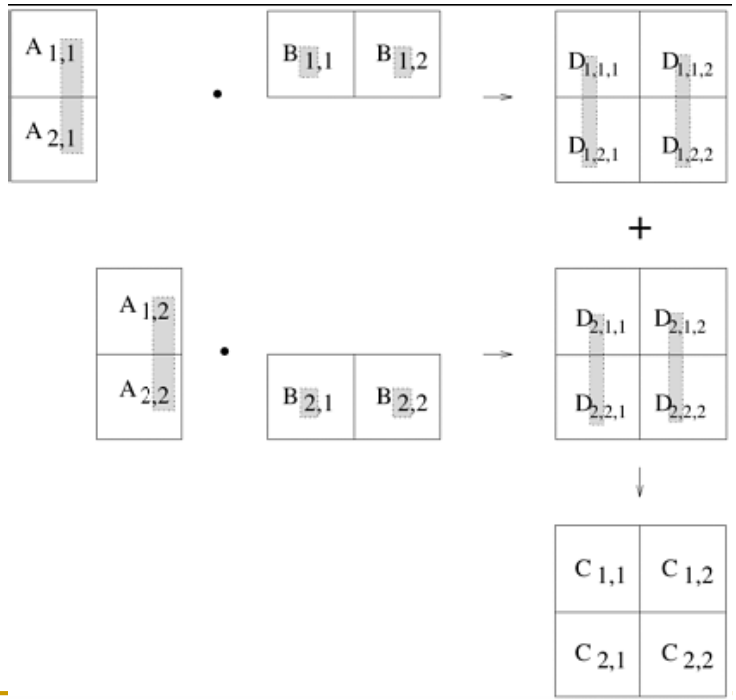
Partitionarea datelor intermediare

- Partitionarea datelor intermediare poate duce uneori la o concurență mai mare decât datele de intrare sau ieșire a partiționării.
- Adesea, datele intermediare nu sunt generate explicit în alg. serial pentru rezolvarea problemei.

⇒ O anumită restructurare a alg.inițial. poate fi solicitată folosirea partiționării intermediare a datelor pentru a induce o descompunere.

Exemplu: inmultirea matricelor

- Reamintim că descompunerile induse de o partiționare 2 x 2 a matricei C de ieșire au un grad maxim de concurgență de patru.
- Creste gradul de concurenta prin introducerea unei etape intermediare in care 8 sarcini isi calculeaza submatricile de produs respective si stocheaza rezultatele intr-o matrice 3-D temporara, asa cum se arata in Fig.
- Submatricea *Dk,i,j* este produsul lui *Ai,k* cu *Bk,j*.



Partitionarea datelor intermediare - exemplu

- O partiționare a matricei intermediare D induce o descompunere în opt sarcini. (vezi Fig.)
  - După faza de înmulțire, o etapă de adunare relativ ieftină poate calcula matricea C de rezultat.
  - Toate submatricele *D*,i,j* cu aceeași a doua și a treia dimensiune *i* și *j* se aduna pentru a obține *Ci,j*.
  - Cele opt sarcini numerotate de la 1 la 8 din Fig. efectuaza O(n^3/8) operatii de inmultire a submatricelor *n*/2 x *n*/2 ale lui *A* si *B*.
  - Apoi, patru sarcini numerotate de la 9 până la 12 petrec timp O(n^2/4) fiecare pentru adunarea submatricilor *n*/2 x *n*/2 corespunzătoare ale matricei intermediare *D* pentru a produce matricea rezultatului final *C*.
- A doua figură arată graficul de dependență a sarcinilor

Stage I

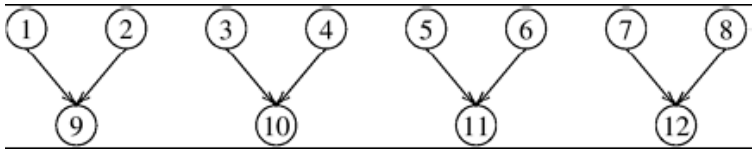
$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} \begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} \\ \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \end{pmatrix}$$

Stage II

$$\begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} + \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

A decomposition induced by a partitioning of D

$$\begin{aligned}
\text{Task 01:} \quad & D_{1,1,1} = A_{1,1}B_{1,1} \\
\text{Task 02:} \quad & D_{2,1,1} = A_{1,2}B_{2,1} \\
\text{Task 03:} \quad & D_{1,1,2} = A_{1,1}B_{1,2} \\
\text{Task 04:} \quad & D_{2,1,2} = A_{1,2}B_{2,2} \\
\text{Task 05:} \quad & D_{1,2,1} = A_{2,1}B_{1,1} \\
\text{Task 06:} \quad & D_{2,2,1} = A_{2,2}B_{2,1} \\
\text{Task 07:} \quad & D_{1,2,2} = A_{2,1}B_{1,2} \\
\text{Task 08:} \quad & D_{2,2,2} = A_{2,2}B_{2,2} \\
\text{Task 09:} \quad & C_{1,1} = D_{1,1,1} + D_{2,1,1} \\
\text{Task 10:} \quad & C_{1,2} = D_{1,1,2} + D_{2,1,2} \\
\text{Task 11:} \quad & C_{2,1} = D_{1,2,1} + D_{2,2,1} \\
\text{Task 12:} \quad & C_{2,2} = D_{1,2,2} + D_{2,2,2}
\end{aligned}$$

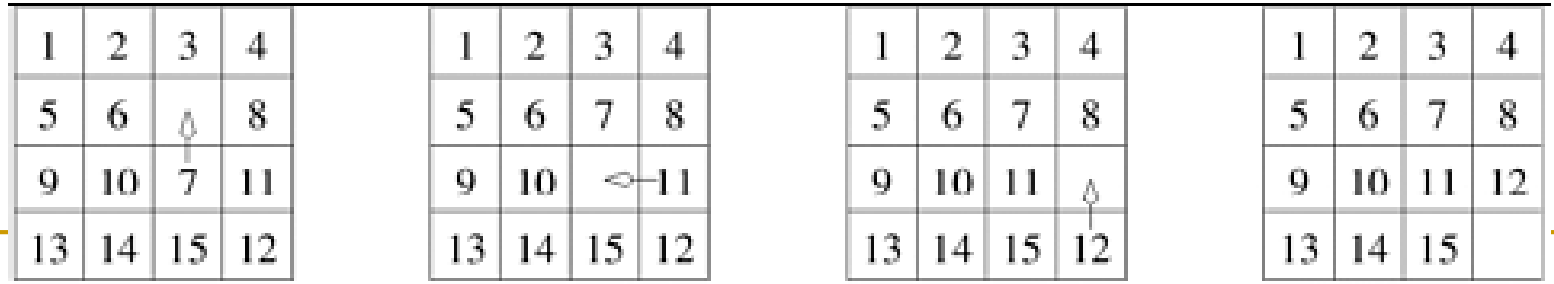


Regula proprietarul-calculeaza

- O descompunere bazată pe date de ieșire sau de intrare de partitionare este, de asemenea, denumită regulă proprietarul-calculeaza.
- Ideea din spatele acestei reguli este că fiecare partiție efectuează toate calculele care implică date pe care le deține.
- În funcție de natura datelor sau de tipul de partiționare a datelor, regula proprietarului-calculeaza poate însemna lucruri diferite:
  - Când alocăm partiții ale datelor de intrare sarcinilor, atunci regula înseamnă că o sarcină realizează toate calculele care pot fi realizate folosind aceste date.
  - Dacă partiționăm datele de ieșire, atunci regula înseamnă că o sarcină calculează toate datele din partiția care i-a fost atribuită.

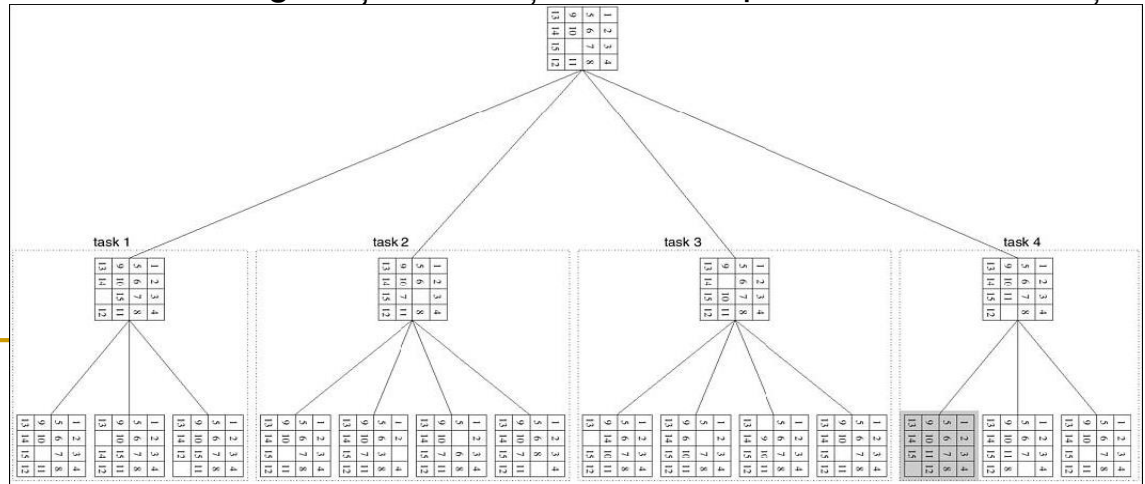
Descompunere exploratorie

- Se utilizează pentru a descompune problemele ale căror calcule de bază corespund unei căutări intr-un spațiu pentru soluții.
- Partiționarea spațiul de căutare în părți mai mici și căutarea simultana in fiecare dintre aceste părți, până la găsirea soluțiilor dorite.
- Exemplu: fie problema unui puzzle cu 15 piese.
  - Constă din 15 piesa numerotate de la 1 la 15 și o placă goală plasată într-o grilă de 4 x 4.
  - O piesă poate fi mutată în poziția goală dintr-o poziție adiacentă acesteia, creând astfel un gol în poziția inițială a plăcii.
  - Sunt posibile patru mișcări: sus, jos, stânga și dreapta.
  - Sunt specificate configurațiile inițiale și finale ale pieselor.
  - Obiectivul este de a determina orice secvență sau cea mai scurtă secvență de mutări care transformă configurația inițială în configurația finală.
  - Fig. Ilustrează configurațiile inițiale și finale finale și o secvență de mutări care duc de la configurația inițială la configurația finală.



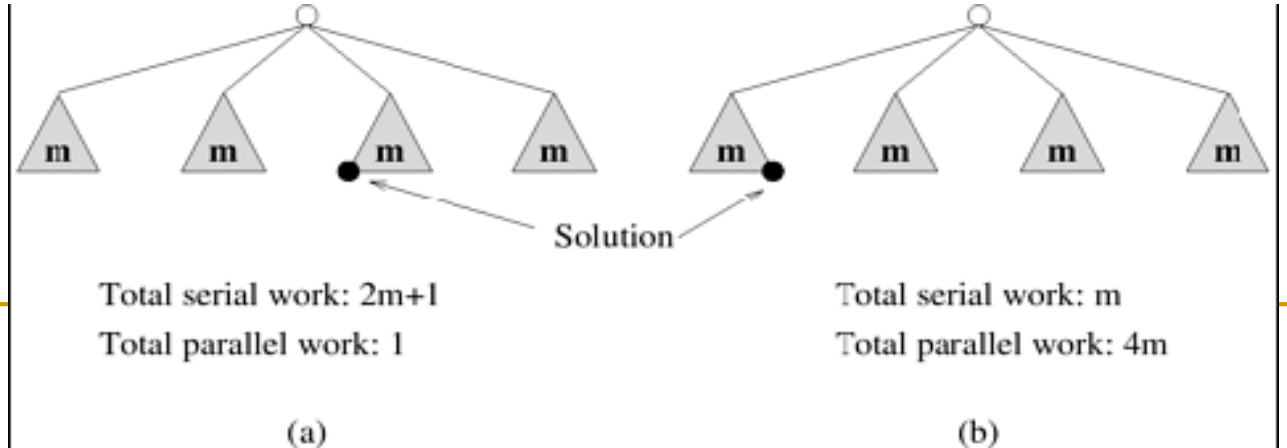
Ex. descompunere exploratory - puzzle

- Puzzle-ul este de obicei rezolvat folosind tehnici de căutare în arbori.
  - Pornind de la configurația inițială, sunt generate toate configurațiile succesoare posibile.
  - O configurație poate avea 2, 3 sau 4 configurații posibile, fiecare corespunzând ocupării slotului gol de către unul dintre vecinii săi.
  - Sarcina de a găsi o cale de la configurația inițială la cea finală se traduce acum la găsirea unei căi de la una dintre aceste configurații recent generate la configurația finală.
  - Deoarece una dintre aceste configurații recent generate trebuie să fie mai aproape de soluție printr-o singură mișcare (dacă există o soluție), s-a făcut un progres spre găsirea soluției.
- Spațiul de configurare generat de căutarea în arbore este graful spațiului de stare.
  - Fiecare nod al graficului este o configurație și fiecare margine a graficului conectează configurații la care se poate ajunge unul de la altul printr-o singură mișcare a unei plăci.
- O metodă pentru rezolvarea acestor probleme în paralel:
  - În primul rând, câteva niveluri de configurații începând de la configurația inițială sunt generate în serie până când arborele de căutare are un număr suficient de noduri de frunze.
  - Acum fiecare nod este atribuit unei sarcini pentru a explora mai departe până când cel puțin unul dintre ei găsește o soluție.
  - De îndată ce una dintre sarcinile concomitente găsește o soluție, aceasta poate informa ceilalți să își încheie căutările.
- Figura ilustreaza
  o asemenea descompunere
  in patru sarcini
  in care sarcina 4 gaseste
  solutia.



Exploratoriu vs. descompunerea datelor

- Sarcinile induse de descompunerea datelor se realizează în întregime și fiecare sarcină realizează calcule utile pentru soluționarea problemei.
- În descompunerea exploratorie, sarcinile neterminate pot fi încheiate imediat ce se găsește o soluție globală.
  - Porțiunea din spațiul de căutare căutat (și cantitatea totală de muncă efectuată) in varianta paralelă poate fi diferită de cea căutată de un alg. serial.
  - Volumul efectuat de varianta paralelă poate fi fie mai mic sau mai mare decât cel efectuat de algoritmul serial.
- Exemplu: fie un spațiu de căutare care a fost partiționat în patru sarcini concomitente, așa cum se arată în Fig.
  - Dacă soluția se află chiar la începutul spațiului de căutare corespunzător sarcinii 3 (Fig. (A)), atunci aceasta va fi găsită aproape imediat prin formularea paralelă.
  - Pe de altă parte, dacă soluția se află spre sfârșitul spațiului de căutare corespunzător sarcinii 1 (Fig (b)), atunci formularea paralelă va efectua de aproape patru ori munca algoritmului serial și nu va produce nicio accelerare.



Descompunerea speculativa

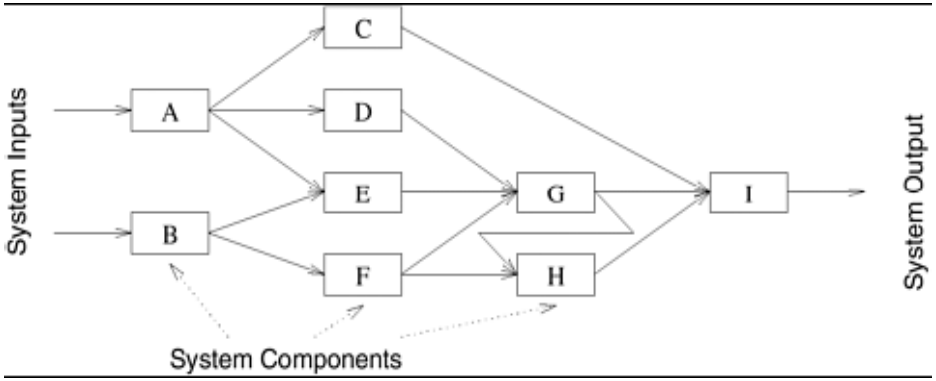
- Este utilizat atunci când un program poate urma una dintre numeroasele ramuri, în funcție de rezultatul altor calcule care îl preced.
  - În timp ce o sarcină realizează calculul a cărui ieșire este utilizată pentru a decide următorul calcul, alte sarcini pot porni simultan calculele din următoarea etapă.
- Acest scenariu este similar cu evaluarea uneia sau a mai multor ramuri ale unei instrucțiuni switch în C în paralel, înainte ca intrarea pentru switchsă fie disponibilă.
  - În timp ce o sarcină efectuează calculul care va rezolva în cele din urmă comutatorul, alte sarcini ar putea ridica mai multe ramuri ale comutatorului în paralel.
  - Când s-a calculat intrările pentru comutator în cele din urmă, se va utiliza calculul corespunzător ramurii corecte, în timp ce cea corespunzătoare celorlalte ramuri ar fi aruncată.
  - Timpul de rulare paralel este mai mic decât timpul de rulare în serie cu timpul necesar pentru evaluarea stării de care depinde următoarea sarcină, deoarece acest timp este utilizat pentru a efectua un calcul util pentru următoarea etapă în paralel.
- Pentru a minimiza calculul irosit, ar putea fi utilizată o formulare ușor diferită de descompunere speculativă, în special în situațiile în care unul dintre rezultatele switchului este mai probabil decât celelalte..
  - În acest caz, numai cea mai promițătoare ramură a preluat o sarcină în paralel cu calculul precedent.
  - În cazul în care rezultatul comutatorului este diferit de cel anticipat, calculul este recuperat și se ia ramura corectă a comutatorului..
  - Accelerarea datorată descompunerii speculative se poate aduna dacă există mai multe etape speculative.

Ex. Pentru descompunerea speculativa

- **Simulare paralela de evenimente discrete.**
- Fie simularea unui sistem care este reprezentata ca o rețea sau un grafic direcționat.
  - Nodurile acestei rețele reprezintă componente.
  - Fiecare componentă are un buffer de intrare pentru joburi.
  - Starea inițială a fiecărui component sau nod este inactivă.
- O componentă inactivă:
  - alege un job din coada de intrare, dacă există una,
  - procesează acea muncă într-o perioadă de timp finită,
  - îl introduce în bufferul de intrare al componentelor care sunt conectate la acesta prin arcele de ieșire.
- O componentă trebuie să aștepte dacă tamponul de intrare al unuia dintre vecinii săi este complet, până când vecinul alege un loc de muncă pentru a crea spațiu în buffer.
- Figura prezinta un exemplu simplu.
- Definiți sarcini speculative care încep să simuleze o subparte a rețelei, fiecare presupunând una dintre mai multe intrări posibile la acea etapă.
- Atunci când o intrare efectivă într-o anumită etapă devine disponibilă (ca urmare a finalizării unei alte sarcini de selecție dintr-o etapă anterioară), atunci toate sau o parte a lucrărilor necesare pentru a simula această intrare ar fi fost deja finalizate

  dacă speculațiile erau corecte,

  sau simularea acestei etape este repornită cu cea mai recentă introducere corectă dacă speculațiile erau incorecte.

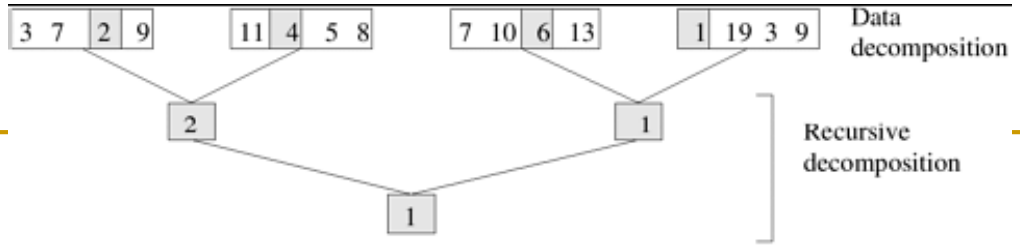


Speculativa vs. exploratorie

- În descompunerea exploratorie:
  - nu se cunoaște rezultatul mai multor sarcini ale unor ramuri.
  - algoritmul serial poate explora diferite alternative una după alta, deoarece ramura care poate duce la soluție nu este cunoscută dinainte;
  - ⇒ programul paralel poate efectua mai mult, mai puțin sau aceeași cantitate de lucru total comparativ cu algoritmul serial în funcție de locația soluției în spațiul de căutare.
- În descompunerea speculativă:
  - intrarea la o ramură care conduce la mai multe sarcini paralele nu este cunoscută;
  - algoritmul serial ar îndeplini strict una dintre sarcinile într-o etapă speculativă, deoarece atunci când ajunge la începutul acelei etape, știe exact ce ramură trebuie să ia.
  - ⇒ un program paralel care utilizează descompunerea speculativă realizează mai multe sarcini agregate decât omologul său serial.

Descompuneri hibride

- Tehnologiile de descompunere nu sunt exclusive și pot fi adesea combinate împreună.
  - Adesea, un calcul este structurat în mai multe etape și uneori este necesar să se aplice diferite tipuri de descompunere în diferite etape.
- Ex. 1: în timp ce găsim minimul unui set mare de n numere,
  - o descompunere pur recursivă poate avea ca rezultat mai multe sarcini decât numărul de procese, *P*.
  - O descompunere eficientă ar împărți intrarea în *P* aproximativ părți egale și ar face ca fiecare sarcină să calculeze minimul secvenței atribuite.
  - Rezultatul final poate fi obținut prin găsirea minimului rezultatelor intermediare P utilizând descompunerea recursivă prezentată în Fig.
- Ex. 2: quicksort in paralel.
  - S-a folosit o descompunere recursivă pentru a rezulta o formulare concurentă.
    - Această formulare are ca rezultat O (n) sarcini pentru problema sortării unei secvențe cu dimensiunea n.
    - Dar, datorită dependențelor dintre aceste sarcini și datorită dimensiunilor inegale ale sarcinilor, concurenta efectivă este destul de limitată.
    - De ex., prima sarcină pentru împărțirea listei de intrare în două părți necesită timp O (n), ceea ce pune o limită superioară câștigului de performanță posibil prin paralelizare.
  - Etapa de divizare a listelor efectuate de sarcini în quicksort paralel poate fi, de asemenea, descompusă folosind tehnica de descompunere a intrării.
    - Descompunerea hibridă rezultată care combină descompunerea recursivă și descompunerea datelor de intrare duce la o formulare extrem de concurentă a quicksort.



Orchestrarea intr-un exemplu

- Versiunea simplificată a nucleului a problemei Oceanului: soluționarea ecuației sale.
- Utilizează soluționatorul ecuației pentru a săpa mai adânc și a ilustra modul de implementare a unui program paralel folosind cele trei modele de programare.
- Nucleul rezolvator de ecuații rezolvă o ecuație diferențială parțială simplă pe o grilă, folosind ceea ce se numește o metodă de diferențiere finită.
- Funcționează pe o rețea obișnuită, în 2 d sau cu o serie de (n + 2) -by- (n + 2) elemente, cum ar fi o singură secțiune transversală orizontală a bazinului oceanic.
- Rândurile și coloanele de margine conțin valori de graniță care nu se modifică, în timp ce punctele interioare n-cu-n sunt actualizate de solver începând de la valorile lor inițiale.
- Calculul se desfășoară pe mai multe iteratii.
- În fiecare iteratie, acesta operează pe toate elementele grilei, pentru fiecare element înlocuindu-și valoarea cu o medie ponderată proprii și de cele patru elemente vecine ale sale (deasupra, de jos, stânga și dreapta).
- Actualizările sunt făcute în loc în grilă, astfel încât un punct vede noile valori ale punctelor de deasupra și la stânga, precum și valorile vechi ale punctelor de sub și la dreapta sa.
- Această formă de actualizare se numește metoda Gauss-Seidel.
- În timpul fiecărei iteratie, nucleul calculează, de asemenea, diferența medie a unui element actualizat față de valoarea sa anterioară.
- Dacă această diferență medie față de toate elementele este mai mică decât un parametru „toleranță” predefinit, se spune că soluția a convers și problema e rezolvata.
- În caz contrar, efectuează încă o iteratia și testează din nou convergența.

Exemplu - descompunere

Secvential:

```
float diff = 0, temp;
while (!done) do
    /*outermost loop over sweeps */
diff = 0; /* initialize max.diff. to 0 */
for i >=□1 to n do
/* sweep over non-border points of grid */
for j >=□1 to n do
temp = A[i,j]; /* save old value of elem*/
A[i,j] <- 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
     A[i,j+1] + A[i+1,j]);
/*compute average */
diff += abs(A[i,j] - temp);
end for
end for
```

Descompunere

- Pentru programele care sunt structurate în cicluri succesive sau cicluri imbricate, o modalitate simplă de a identifica concurenta este de a porni de la structura ciclului în sine.
- Fie ciclul interior (in *j*).
  - Fiecare iterație a acestei bucle citește punctul de grilă (A [i, j-1]) care a fost scris în iterația anterioară.
  - Prin urmare, iterațiile sunt dependente secvențial și numim acest ciclu secvențial.
  - Ciclul exterioar a acestei imbricari este de asemenea secvențial, deoarece elementele din rândul i-1 au fost scrise în iterația anterioară (i-1) a acestei bucle.
- Prin urmare, această analiză simplă a ciclurilor existente și a dependențelor lor descoperă că nu există o concurentță în acest caz.

Exemplu

- O alternativă pentru a găsi concurență este să te întorci la dependențele fundamentale din algoritmii de bază folosiți, indiferent de structura programului.
- Analizați dependențele de date fundamentale la granularitatea punctelor de grilă individuale.
  - Calcularea unui anumit punct de grilă din programul secvențial folosește valorile actualizate ale punctelor de grilă direct deasupra și spre stânga.
  - Elementele de-a lungul unei anti-diagonale date (de la sud-vest la nord-est) nu au dependențe între ele și pot fi calculate în paralel, în timp ce punctele din următoarea anti-diagonală depind de unele puncte din precedentul.
  - Din această diagramă, putem observa cea a lucrării implicate în fiecare măturare, există o dependență secvențială proporțională cu n de-a lungul diagonalei și concurență inerentă proporțională cu n.
- Descompunrea în puncte de grilă individuale, astfel încât actualizarea unui singur punct de grilă este o sarcină.
  - Abordarea 1: Lăsați structura buclă a programului așa cum este. Introduceți sincronizarea de la un punct la altul pentru a vă asigura că a fost produs un punct de grilă în mătura curentă înainte de a fi utilizat de către punctele din dreapta sau dedesubtul acestuia. S-ar putea să se desfășoare simultan diferite cuiburi de bucle și chiar iteratii diferite pe elemente diferite, cu condiția să nu fie încălcate dependențele la nivel de element. Supravegherea acestei sincronizări la nivel de punct de grilă poate fi prea mare.
  - Abordarea 2: Schimbați buclele: bucla exterioară să fie peste antidiagonale și bucla interioară să fie peste elemente dintr-o anti-diagonală. Bucla interioară poate fi acum executată complet în paralel, cu o sincronizare globală între iterațiile exterioare pentru bucla pentru a păstra dependențele în mod conservator între antidiagonale. Sincronizarea globală este încă foarte frecventă: o dată pe antidiagonal. De asemenea, numărul de iterații în bucla paralelă (interioară) se modifică cu iterații succesive ale buclei exterioare, provocând dezechilibre de încărcare între procesoare, în special la antidiagonale mai scurte.
- Din cauza frecvenței de sincronizare, a dezechilibrelor de încărcare și a complexității programării, niciuna dintre aceste abordări nu este folosită prea mult în arhitecturile moderne.

Exemplu – ordonarea rosu-negru

- Abordarea 3: exploatarea cunoștințelor problemei dincolo de programul secvențial.
- Soluția Gauss-Seidel: se iterează până la convergență, putem actualiza punctele grilei într-o ordine diferită, atât timp cât folosim valori actualizate pentru punctele de grilă suficient de frecvent.
- O astfel de ordine care este folosită adesea pentru versiuni paralele se numește ordine roșu-negru.
  - Ideea de aici este de a separa punctele grilei în puncte roșii și puncte negre alternate ca pe un tablou de sah, astfel încât niciun punct roșu să nu fie adiacent unui punct negru sau invers.
  - Pentru a calcula un punct roșu nu avem nevoie de valoarea actualizată a niciunui alt punct roșu, ci doar de valorile actualizate ale punctelor negre de mai sus și de la stânga și invers.
  - Prin urmare, putem împărți o iteratiei în două faze: mai întâi calculând toate punctele roșii și apoi calculând toate punctele negre.
  - În cadrul fiecărei faze nu există dependențe între punctele grilei,a.i.putem calcula toate punctele roșii în paralel, apoi sincronizăm la nivel global și calculăm toate punctele negre în paralel.
  - Sincronizarea globală este conservatoare și poate fi înlocuită cu sincronizarea punct la punct la nivelul punctelor de grilă - deoarece nu toate punctele negre trebuie să aștepte calcularea tuturor punctelor roșii - dar este convenabil.
- Comanda roșu-negru este diferită de ordonarea secvențială inițială și, prin urmare, poate converge în mai puține sau mai multe iteratii, precum și produce valori finale diferite pentru punctele de grilă (deși încă în toleranța convergenței)).
  - Convergenta va fi mai lenta.
  - Aceasta se numește iteratie Jacobi si nu Gauss-Seidel.

Asignarea pentru exemplul considerat

- Asignare statică:
  - Cea mai simplă opțiune este o alocare statică (predeterminată) în care fiecare procesor este responsabil pentru un bloc de rânduri contigu: atribuirea blocului.
  - Alternativă: alocare ciclică în care rândurile sunt intercalate între procese.
- Asignare dinamică:
  - fiecare proces apucă în mod repetat următorul rând disponibil (încă nu calculat) după ce se termină cu sarcin unui rând;
  - nu este predeterminat ce proces calculează ce rânduri.
- Asignarea blocului statică:
  - Prezintă un echilibru de încărcare bun între procese atâta timp cât numărul de rânduri este divizibil după numărul de procese, deoarece munca pe linie este uniformă.

Orchestrarea in cazul paralelismului datelor

- Diferenta fata de codul secvential:
  - Datele partajate alocate dinamic, sunt alocate cu un apel G_MALLOC (malloc global), decât cu un malloc obișnuit.
  - utilizare DECOMP,
  - utilizare for_all in loc de for,
  - Utilizarea unei variable mydiff private per process,
  - Utilizarea REDUCE.
- for_all specifica iteratii executate in parallel.
- DECOMP are un dublu scop.
  - alocarea iterațiilor la procese:
    - [BLOCK, *, nprocs] assignment: prima dim (rânduri) este partiționata în bucăți contigue între procesele nprocs, iar cea de-a doua dimensiune nu este deloc partiționată.
    - [CYCLIC, *, nprocs] ar fi presupus o partiționare ciclică sau intercalată a rândurilor între procesele nprocs,
    - [BLOCK, BLOCK, nprocs] o descompunere pe subblocs,
    - [*, CYCLIC, nprocs] impartire intercalată a coloanelor.
  - specifică modul în care datele trebuie distribuite între memoriile unui system cu memorie distribuita.
- mydiff variabila este utilizată pentru a permite fiecărui proces să calculeze în mod independent suma valorilor diferenței pentru punctele de grilă alocate.
- REDUCE: instruieste sistemul să adune toate valorile mydiff parțiale în variabila dif partajată.
  - Operația de reducere poate fi implementată într-o bibliotecă într-o manieră adecvată cel mai bine arhitecturii de bază.

```
int n, nprocs;
/* grid size (n+2-by-n+2) and number of processes*/
float **A, diff = 0;
main()
begin
read(n); read(nprocs);; /* read input grid size and no.processes*/
A <- G_MALLOC (a 2-d array of size n+2 by n+2 doubles);
initialize(A); /* initialize the matrix A somehow */
Solve (A); /* call the routine to solve equation*/
end main
procedure Solve(A) /* solve the equation system */
float **A; /* A is an n+2 by n+2 array*/
begin
int i, j, done = 0;
float mydiff = 0, temp;
DECOMP A[BLOCK,*];
while (!done) do /* outermost loop over sweeps */
mydiff = 0; /* initialize maximum difference to 0 */
for_all i >= 1 to n do /* sweep over non-border points of grid */
for_all j >= 1 to n do
temp = A[i,j]; /* save old value of element */
A[i,j] <- 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
A[i,j+1] + A[i+1,j]); /*compute average */
mydiff += abs(A[i,j] - temp);
end for_all
end for_all
REDUCE (mydiff, diff, ADD);
if (diff/(n*n) < TOL) then done = 1;
end while
end procedure
```

Orchestrarea in cazul spatiului de adresare partajat

- Declarați matricea A ca un singur tablou comun;
- Procesele pot face referire la părțile din care au nevoie folosind încărcări și stocarile cu exact aceiași indici de matrice ca într-un program secvențial.
- Comunicarea va fi generată implicit, după cum este necesar.
- Cu procese paralele explicite, acum avem nevoie de mecanisme:
  - De creare procese,
  - De coordononare prin sincronizare și
  - De control a sarcinii.
- Diferențele de codul secvențial sunt afișate cu caractere aldine cursive.

```
1. int n, nprocs; /* matrix dimension and number of processors to be used */
2a. float **A, diff; /*A is global (shared) array representing the grid */
/* diff is global (shared) maximum difference in current sweep */
2b. LOCKDEC(diff_lock); /* declaration of lock to enforce mutual exclusion */
2c. BARDEC (bar1); /* barrier declaration for global sync between sweeps*/
3. main()
4. begin
5. read(n); read(nprocs); /* read input matrix size and number of processes*/
6. A <- G_MALLOC (a two-dimensional array of size n+2 by n+2 doubles);
7. initialize(A); /* initialize A in an unspecified way*/
8a. CREATE (nprocs-1, Solve, A);
8 Solve(A); /* main process becomes a worker too*/
8b. WAIT_FOR_END; /* wait for all child processes created to terminate */
9. end main
10. procedure Solve(A)
11. float **A; /*A is a n+2-by-n+2 shared array,as in the sequential program */
12. begin
13. int i,j, pid, done = 0;
14. float temp, mydiff = 0; /* private variables/
14a. int mymin <- 1 + (pid * n/nprocs); /*assume that n is divisible by */
14b. int mymax <- mymin + n/nprocs - 1; /* nprocs for simplicity here*/
15. while (!done) do /* outer loop over all diagonal elements */
16. mydiff = diff = 0; /* set global diff to 0 (okay for all to do it) */
17. for i >= mymin to mymax do /* for each of my rows */
18. for j >= 1 to n do /* for all elements in that row */
19. temp = A[i,j];
20. A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21. A[i,j+1] + A[i+1,j]);
22. mydiff += abs(A[i,j] - temp);
23. endfor
24. endfor
25a. LOCK(diff_lock); /* update global diff if necessary */
25b. diff += mydiff;
25c. UNLOCK(diff_lock);
25d. BARRIER(bar1,nprocs);/* ensure all have got here before checking if done*/
25e. if (diff/(n*n)<TOL) then done=1; /*check convergence;all get same answer*/
25f. BARRIER(bar1, nprocs); /* see Exercise c */
26. endwhile
27.end procedure
```

Orchestrarea in modelul transmiterii de mesaje

```
1. int pid, n, nprocs; /* process id, matrix dimension & no.
       processors to be used */
2. float **myA;
3. main()
4. begin
5. read(n); read(nprocs); /* read input matrix size and number of
       processes*/
8a. CREATE (nprocs-1 processes that start at procedure Solve);
8b. Solve(); /* main process becomes a worker too*/
8c. WAIT_FOR_END; /* wait for all child processes created to
       terminate */
9. end main
10. procedure Solve()
11. begin
13. int i,j, pid, n' = n/nprocs, done = 0;
14. float temp, tempdiff, mydiff = 0; /* private variables/
6. myA <-malloc(2d array of size[n/nprocs+2] by n+2);/*my
       assigned rows of A */
7. initialize(myA); /* initialize my rows of A, in an unspecified way*/
15.while (!done) do
16. mydiff = 0; /* set local diff to 0 */
16a.if (pid != 0) then SEND(&myA[1,0],n*sizeof(float),pid-1,ROW);
16b. if (pid = nprocs-1) then
       SEND(&myA[n',0],n*sizeof(float),pid+1,ROW);
16c. if (pid != 0) then RECEIVE(&myA[0,0],n*sizeof(float),pid-
       1,ROW);
16d. if (pid != nprocs-1) then
       RECEIVE(&myA[n'+1,0],n*sizeof(float),pid+1,ROW);
/*border rows of neighbors have now been copied into myA[0,*]
       and myA[n'+1,*]*/
17. for i >=□1 to n' do /* for each of my rows */
18. for j >=□1 to n do /* for all elements in that row */
19. temp = myA[i,j];
20. myA[i,j] <- 0.2 * (myA[i,j] + myA[i,j-1] + myA[i-1,j] +
21. myA[i,j+1] + myA[i+1,j]);
22. mydiff += abs(myA[i,j] - temp);
23. endfor
24. endfor
/* communicate local diff values and obtain determine if done;
       can be replaced
by reduction and broadcast */
25a. if (pid != 0) then /* process 0 holds global total diff*/
25b. SEND(mydiff,sizeof(float),0,DIFF);
25c. RECEIVE(mydiff,sizeof(float),0,DONE);
25d. else
25e. for i >=□1 to nprocs-1 do /* for each of my rows */
25f. RECEIVE(tempdiff,sizeof(float),*,DONE);
25g. mydiff += tempdiff; /* accumulate into total */
25h. endfor
25i. for i >=□1 to nprocs-1 do /* for each of my rows */
25j. SEND(done,sizeof(int),i,DONE);
25k. endfor
25l. endif
26. if (mydiff/(n*n) < TOL) then done = 1;
27. endwhile
28. end procedure
```

Comments: in the textbook