Curs 12

Programare Paralela si Distribuita

Etape in dezvoltarea programelor paralele (PCAM)

Sabloane de proiectare

PCAM

- Partitioning
- Communication
- Agglomeration
- Mapping

Dezvoltare programe paralele

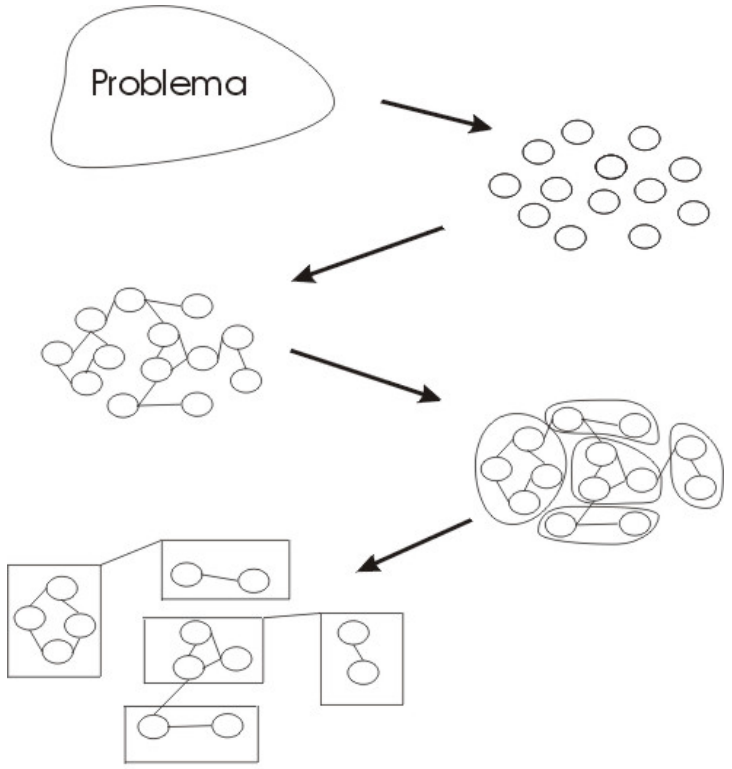
- Nu exista o schema simpla care sa poate fi aplicat pentru dezvoltarea programelor paralele.

- abordare metodologica care sa maximizeze numarul de optiuni,sa furnizeze:
  - mecanisme de evaluare a alternativelor si
  - sa reduca costurile necesare revenirii in cazul luarii unor decizii neperformante.
- O asemenea metodologie de dezvoltare permite programatorului se se concentreze asupra aspectelor independente de arhitectura cum este concurenta, in prima faza de dezvoltare, iar analiza aspectelor dependente de masina sa fie amanata pentru finalul procesului de dezvoltare.

Etape in dezvoltarea programelor paralele – Ian Foster (1995)

- PCAM (http://www.mcs.anl.gov/~itf/dbpp/)
  - partitionarea,
  - comunicatia,
  - aglomerarea si
  - maparea

- Aceasta metodologie defineste aceste etape ca si ***linii directoare*** in procesul de realizare a programelor paralele si ***nu*** neaparat ca niste ***etape stricte*** de dezvoltare.



Partitionarea

- Problema partitionarii are ın vedere ımpartirea problemei de programare ın componente care se pot executa concurent.
- Aceasta nu implica o divizare directa a programului ıntr-un numar de componente egal cu numarul de procesoare disponibile.

- Cele mai importante scopuri ale partitionarii sunt legate de:
  - scalabilitate,
  - abilitatea de a ascunde ıntarzierea(latency) datorata retelei sau accesului la memorie si
  - realizarea unei granularitati cat mai mari.

- Sunt de preferat partitionarile care furnizeza mai multe componente decat procesoare, astfel ıncat sa se permita ascunderea ıntarzierii.

- Un task va fi blocat, sau va astepta, pana cand mesajul care contine informatia dorita va ajunge;

- Daca exista si alte componente program disponibile, procesorul poate continua calculul => multiprogramare => executiei concurente..

Graf de dependenta

- Prin partitionare se urmareste descompunerea problemei in activitati de calcul cat mai fine si concurente.
- Rezultatul este un graf de dependenta ale carui noduri sunt taskurile iar arcele exprima relatii de ***precedenta*** intre activitatile de calcul.
- Relatia de precedenta intre doua activitati de calcul a1 si a2 este indicata de dependenta de date:

- dependenta de curs("flow dependency") – data de intrare pentru a2 este produsa ca data de iesire de catre a1;

- antidependenta("antidependency") – a2 urmeaza dupa a1, iar iesirea lui a2 se suprapune cu intrare lui a1;

- dependenta de iesire("output dependency") – a1 si a2 produc (scriu) aceeasi data de iesire;

- dependenta de I/E (" I/O dependency") – a1 si a2 acceseaza simultan acelasi fisier.

Solutii:

(Teorema lui Berstein $[I(a1) \cap O(a2)] \cup [O(a1) \cap I(a2)] \cup [O(a1) \cap O(a2)] \neq \emptyset$)

Strategii de partitionare

- Exista doua strategii principale de partitionare:
  - descompunerea domeniului de date si
  - descompunerea functionala.

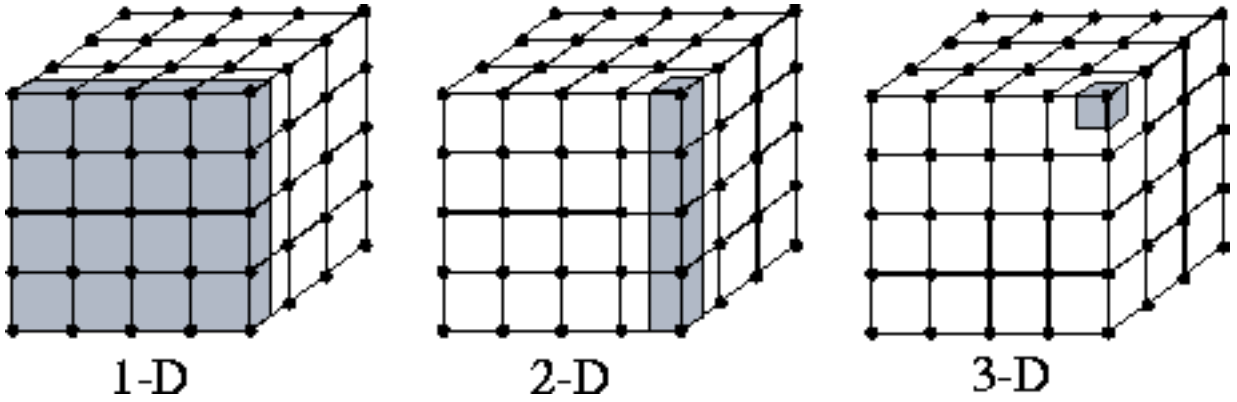
- In functie de acestea putem considera aplicatii paralele bazate pe:
  - descompunerea domeniului de date – paralelism de date, si
  - aplicatii paralele bazate pe descompunerea functionala.

- Cele doua tehnici pot fi folosite insa si impreuna:
  - de exemplu se incepe cu o descompunere functionala si dupa identificarea pricipalelor functii se poate folosi descompunerea domeniului de date pentru fiecare in parte.

Descompunerea domeniului de date

- Este aplicabila atunci cand domeniul datelor este mare si regulat.
- Ideea centrala este de a divide domeniul de date, reprezentat de principalele structuri de date, ın componente care pot fi manipulate independent.
- Apoi se partitioneaza operatiile, de regula prin asocierea calculelor cu datele asupra carora se efectueaza.
- Astfel, se obtine un numar de activitati de calcul, definite de un numar de date si de operatii.
  - Este posibil ca o operatie sa solicite date de la mai multe activitati. In acest caz, sunt necesare comunicatii.

Exemplificare



Descompunerea domeniului de date

- Sunt posibile diferite variante, in functie de structurile de date avute in vedere.

- Datele care se partitioneaza sunt in general datele de intrare, dar pot fi considerate deasemenea si datele de iesire sau intermediare.
  - Trebuie avute in vedere in special structurile de date de dimensiuni mari sau cele care sunt accesate in mod frecvent.

- Faze diferite ale calculului pot impune partitionari diferite ale aceleiasi structuri de date
  - =>redistribuiri ale datelor.
  - in acest caz trebuie avute in vedere de asemenea si costurile necesare redistribuirii datelor.

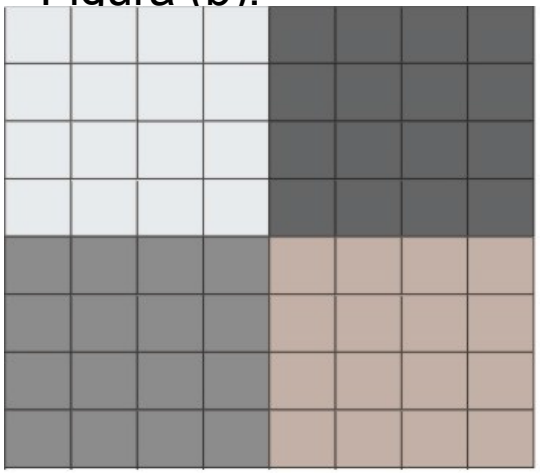
Distributii de date

- Partitionarea datelor conduce la anumite distributii ale datelor per procese si de aici implicit si per procesoare.
- Exista mai multe tehnici de partitionare a datelor, care pot fi exprimate si formal prin functii definite pe multimea indicilor datelor de intrare cu valori in multimea indicilor de procese.

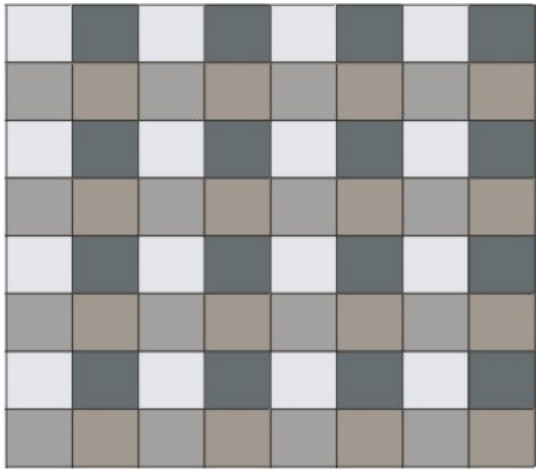
- Cele mai folosite tehnici de partitonare sunt prin “taiere” si prin “incretire” care corespund distributiilor liniara si ciclica.

Distributii de date

- De exemplu, pentru o matrice A de dimeniune 8×8, partitionarea sa in p = 4 parti:
  - prin tehnica taierii (distr. liniara) conduce la partitionarea aratata de Figura (a), iar
  - prin tehnica incretirii (distr. ciclica) conduce la partitionarea aratata de Figura (b).

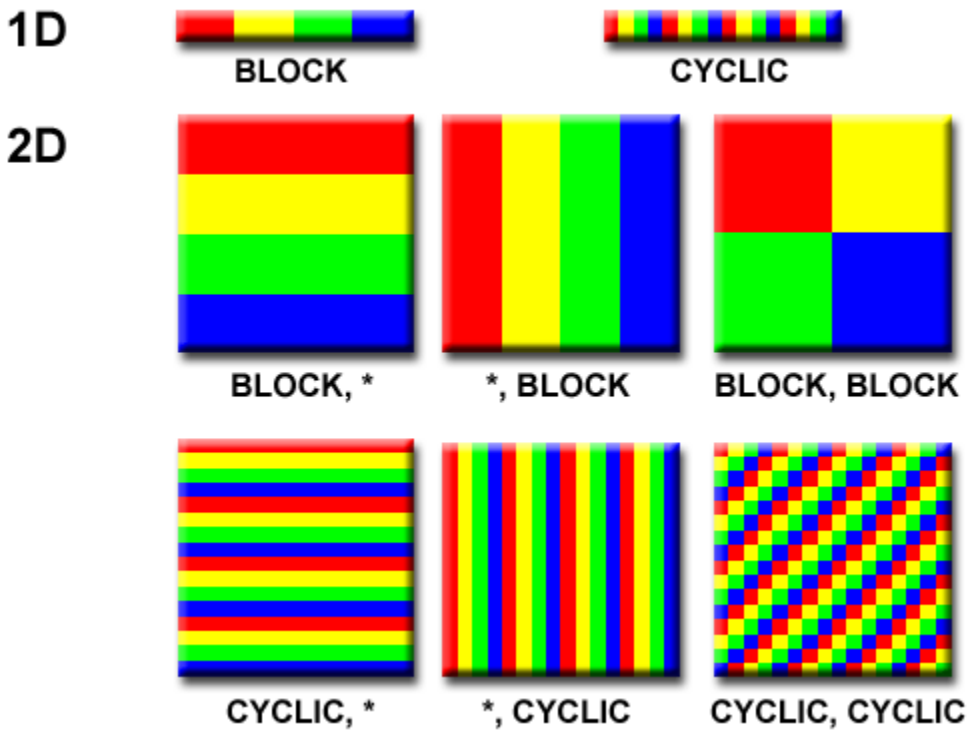


(a) Taiere (distributie bloc)



(b) Incretire (distributie grid)

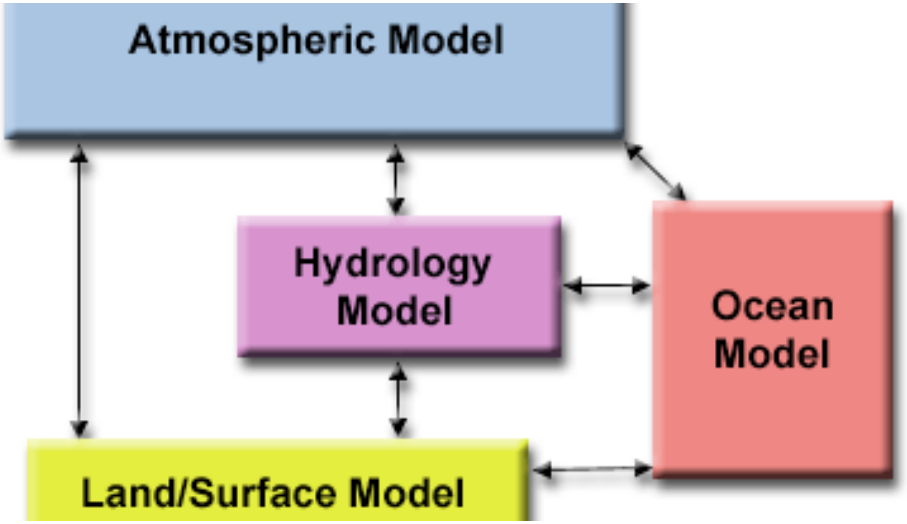
Variante


1D
BLOCK
CYCLIC
2D
BLOCK, *
*, BLOCK
BLOCK, BLOCK
CYCLIC, *
*, CYCLIC
CYCLIC, CYCLIC

Descompunerea functionala

- Descopunerea functionala este o tehnica de partitionare folosita atunci cand aspectul dominant al problemei este functia, sau algoritmul, mai degraba decat operatiile asupra datelor.
- Obiectivul este descompunerea calculelor ın activitati de calcul cat mai fine.
- Dupa crearea acestora se examineaza cerintele asupra datelor.
- Focalizarea asupra calculelor poate revela uneori o anumita structura a problemei, de unde oportunitati de optimizare, care nu sunt evidente numai din studiul datelor.
- In plus, ea are un rol important ca si tehnica de structurare a programelor.
- Aceasta varianta de descompunere nu conduce in general la o granulatie fina a sarcinilor de calcul, care se executa in paralel.

Exemplificare



Cerinte pentru partitionare

- Task-urile obtinute sunt de dimensiuni comparabile.

- Scalabilitatea poate fi obtinuta.
  - Aceasta inseamna ca numarul de sarcini de calcul sunt definite in functie de dimensiunea problemei;
  - deci cresterea dimensiunii datelor implica cresterea numarului de sarcini de lucru.

- Intazierile pot fi reduse prin multitasking.

- Granularitatea aplicatiei este suficient de mare astfel incat sa poata fi implementata cu succes pe diferite arhitecturi.

Analiza comunicatiei

- Procesele de calcul rezultate in urma partitionarii nu pot fi realizate in general independent; ele necesita anumite comunicatii.

- Comunicatia este strans legata de partitionare, mai exact modul de partitionare implica structura comunicatiei.
- Daca consideram procesele obtinute la faza de partitionare ca fiind nodurile unui graf, o comunicatie intre doua procese poate fi reprezentata ca fiind o muchie in acel graf.

- Numarul de comunicatii intre doua procese poate fi reflectat in acest graf ca fiind costul muchiei dintre cele doua.

- In general pentru problemele care se bazeaza pe paralelismul de date, cerintele de comunicatie sunt mai dificil de evaluat.
  - In acest caz, organizarea comunicatiei intr-un mod eficient este mai dificila.
- In cazul descompunerii functionale cerintele de comunicatie se stabilesc mai usor.
  - Interfetele dintre functii sunt mai bine precizate, deci si fluxul de date intre functii se determina intr-o maniera directa.

Tipuri de comunicatii

- Comunicatiile *locale* ->daca fiecare proces comunica cu un numar mic de alte procese, = vecini,
- comunicatii *globale* -> daca (aproape) toate procesele sunt implicate in operatia de comunicatie.

- comunicatie *structurata* => un proces si vecinii sai formeaza o structura regulata,
- comunicatiilor *nestructurate* => nu se poate stabili o structura intre vecini.

- O comunicatie *statica* implica faptul ca vecinii unui proces nu se modifica in timp, iar in cazul unei comunicatii dinamice vecinii unui proces se cunosc doar la executie si se pot modifica in timpul executiei.

- In cazul comunicatiei *asincrone*, procesorul care comunica informatie continua executia dupa ce aceasta a fost transmisa la destinatie, ın timp ce procesorul care receptioneaza mesajul va suferi o ıntarziere datorata asteptarii.
- Comunicatia *sincrona* cere ca atat procesorul care trimite cat si cel care receptioneaza mesajul sa fie disponibile pana ın momentul cand transmisia s-a terminat, ambele procesoare putand apoi sa-si continue lucrul. In felul acesta, nu este necesara stocarea mesajelor (lucru care poate fi necesar ın cazul transmiterii asincrone).

Aglomerarea- Cresterea granularitatii

- Aglomerarea are in vedere combinarea proceselor determinate in faza de partitionare cu scopul principal rde a reducere numarul de comunicatii care ar putea reduce substantial eficienta.
- Trebuie insa sa se urmareasca pastrarea echilibrului de calcul si intre procesele nou rezultate in urma acestei faze si de asemenea pastrarea scalabilitatii aplicatiei.
- Aceasta faza nu mai este legata de proiectarea abstracta a aplicatiei ci are in vedere o anumita arhitectura pe care se doreste implementarea.
- Deci deciziile de combinare vor fi luate si in functie de reteaua de comunicatie a sistemului ales, astfel incat sa se ajunga la o implementare eficienta.
  - Daca din analiza comunicatiei sunt depistate procese care nu pot fi executate in paralel acestea se vor grupa, formand un singur proces.

Observatii

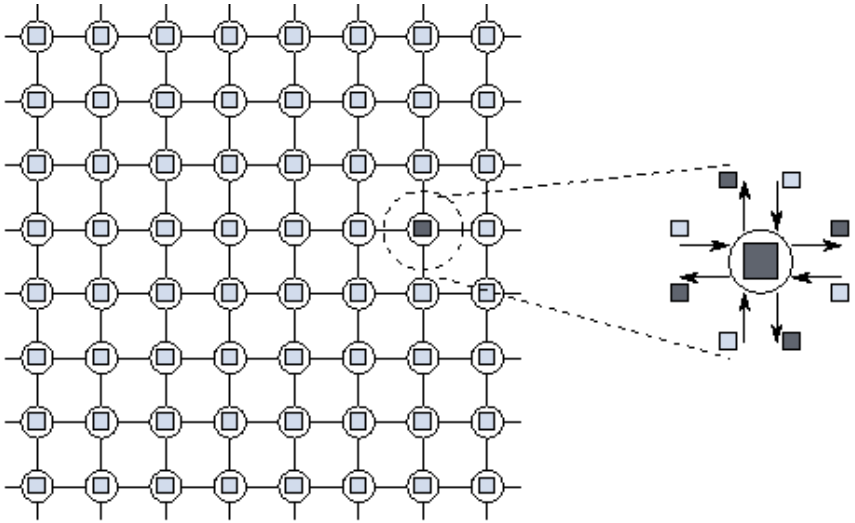
- Aglomerarea nu presupune obtinerea unui numar de procese egal cu numarul de procesoare.

- Este indicat de altfel ca mai multe procese sa fie mapate pe acelasi procesor, pentru a ascunde intarzierile datorate comunicatiilor.

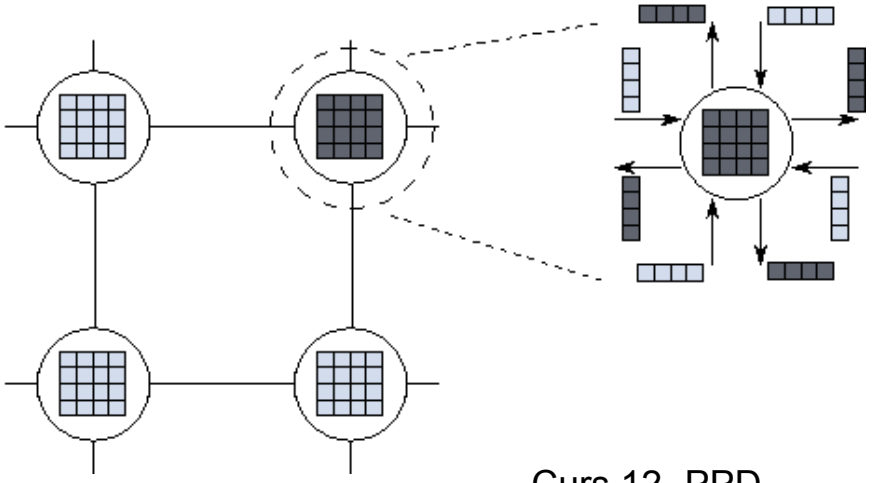
- Daca un proces este blocat din diferite motive (asteapta o informatie, sau ajungerea la o anumita stare), procesorul poate sa execute un alt proces, micsorandu-se astfel timpi morti ai procesoarelor.

Exemplificare

(a)



(b)



Maparea

- Scop mentinerea ***localizarii:***
  - plasarea componentelor problemei care schimba informatie, cat mai aproape posibil, pentru a reduce costul comunicatiilor.
- Deoarece maparea are ın vedere folosirea unei arhitecturi particulare, trebuie sa fie cunoscute anumite detalii ale acesteia.

- Pentru simplificare, presupunem existenta a p procesoare, numarate secvential ıncepand cu 0 pana la p−1.

- Problema maparii este cunoscuta ca fiind NP-completa, adica nu exista un algoritm cu complexitate polinomiala care sa asigure plasarea optima a proceselor pe procesoare.
- Multi algoritmi construiti folosind tehnicile de descompunere a domeniului de date, sunt formati dintr-un numar de procese de marime egala ıntre care comunicatia locala si globala este structurata.
  - In asemenea cazuri, maparea este directa. Mapam procesele astfel ıncat sa minimizam comunicatia interprocesor; este posibil deasemenea, sa mapam mai multe procese pe acelasi procesor.

Mapare(2)

- Pentru algoritmii bazati pe descompunerea domeniului de date cu ıncarcari de calcul diferite pe procese si/sau cu comunicatii nestructurate, strategiile de mapare nu sunt atat de directe.

- Se pot aplica algoritmi de echilibrare a ıncarcarii (load balancing) – in special algoritmi euristici – pentru a identifica strategii eficiente.
  - Timpul consumat executiei acestor algoritmi trebuie sa duca la beneficii considerabile ale timpului de executie global.
  - Metodele probabilistice de echilibrare a ıncarcarii tind sa aiba o ıntarziere mai mica decat cei bazati pe structura aplicatiei.

Mapare(3)

- Cele mai complexe probleme sunt acelea pentru care numarul de procese si cantitatea de calcul si comunicare per proces se schimba ***dinamic*** ın timpul executiei.
  - In cazul problemelor dezvoltate folosind tehnici de descompunere a domeniului de date, se pot folosi strategii dinamice de echilibrare a ıncarcarii (dynamic load-balancing), ın care un algoritm de echilibrare a ıncarcarii se executa periodic.
  - In cazul folosirii descompunerii functionale, poate fi folosit un algoritm de distribuire a proceselor (task-scheduling). Un proces manager centralizat sau distribuit are sarcina de a retine si a distribui procesele.
  - Cel mai complicat aspect al algoritmului de distribuire a proceselor este strategia folosita pentru alocarea problemelor procesoarelor (workers).
  - In general strategia reprezinta un compromis ıntre cerintele pentru operatiile independente – de a ımbunatatii ıncarcarea de calcul si pentru informatiile globale ale starii calculului – de a reduce costurile de comunicatie.

SABLOANE DE PROIECTARE

… in general proiectarea se bazeaza pe…

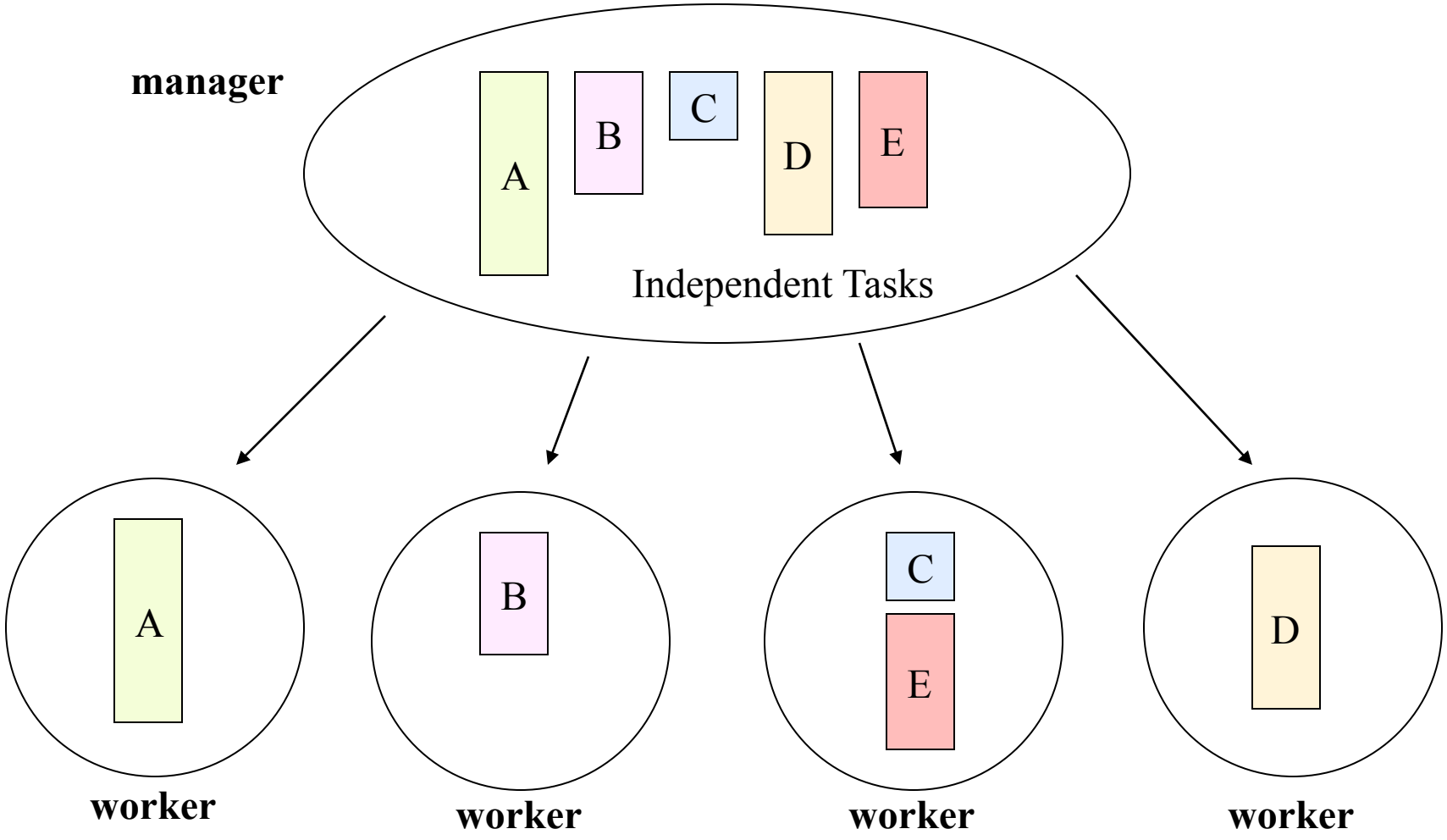
- Descompunere functionala
- Descompunere domeniului de date
- Fluxul de date

Sabloane de baza

Parallel Programming Patterns
Eun-Gyu Kim
2004

- Embarassingly Parallel/Master-Slave
- Replicable
- Repository
- Divide&Conquer
- Pipeline
- Recursive Data
- Geometric Decomposition
- IrregularMesh

Embarassingly Parallel

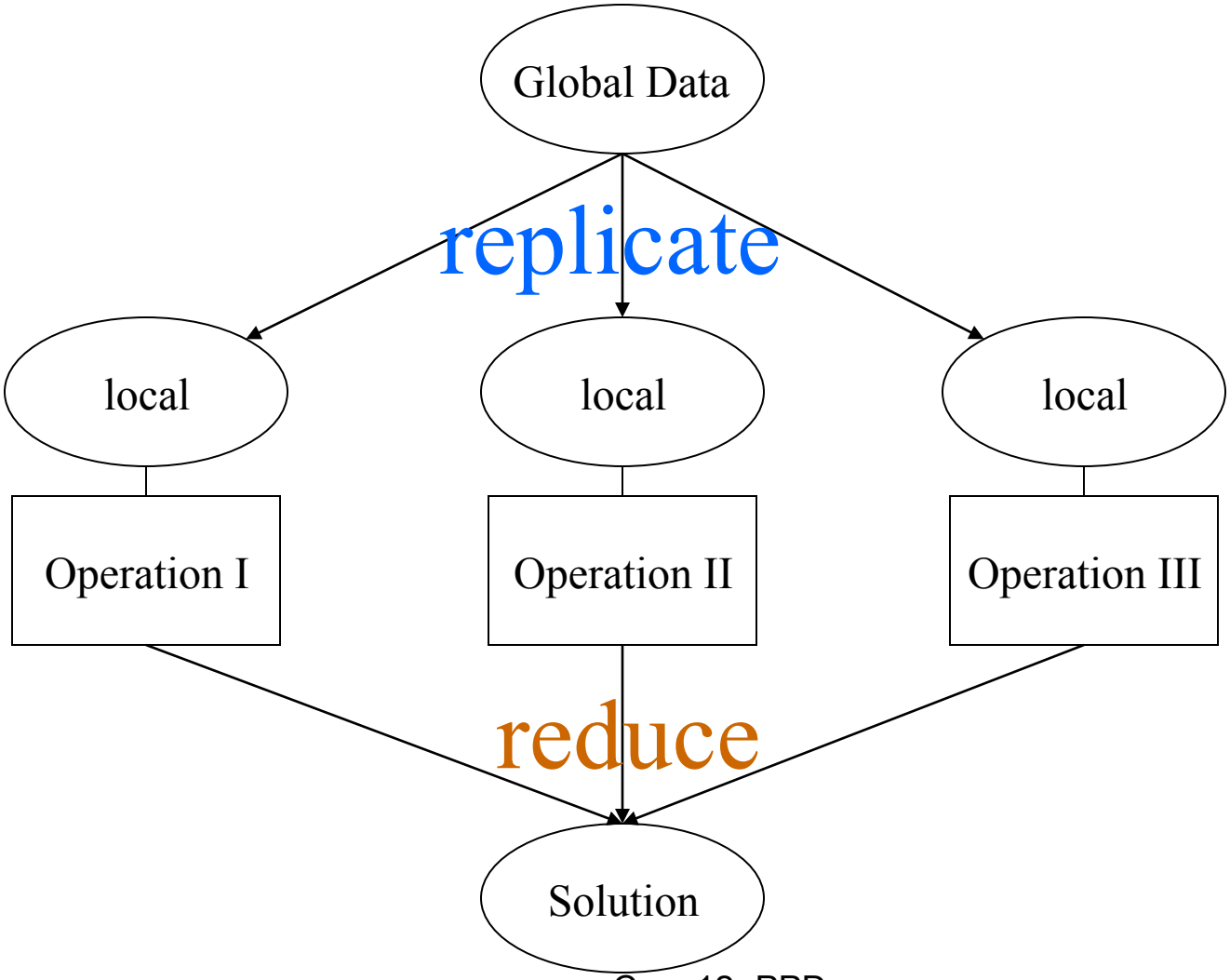


Replicarea datelor

- Cu scopul de a crea taskuri independente
- Exemplu: inmultire matrice: replicare pe dimensiunea 3

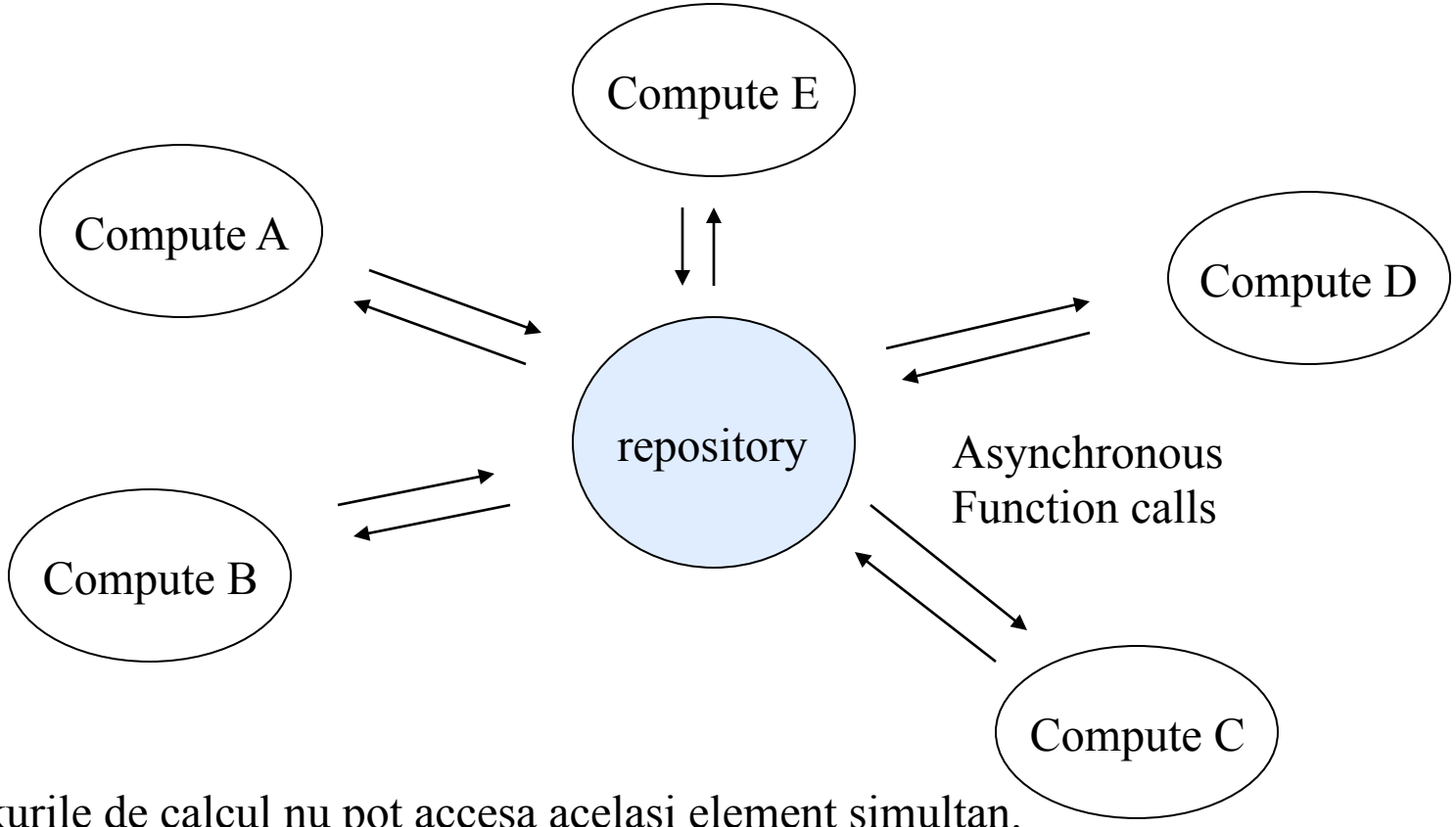
Replicable

Seturi de operatii care trebuie sa se execute folosind o structura globala de date => dependenta



Repository

Calcule independente care trebuie sa se execute pe o structura de date centralizata in tr-un mod nedeterministic.

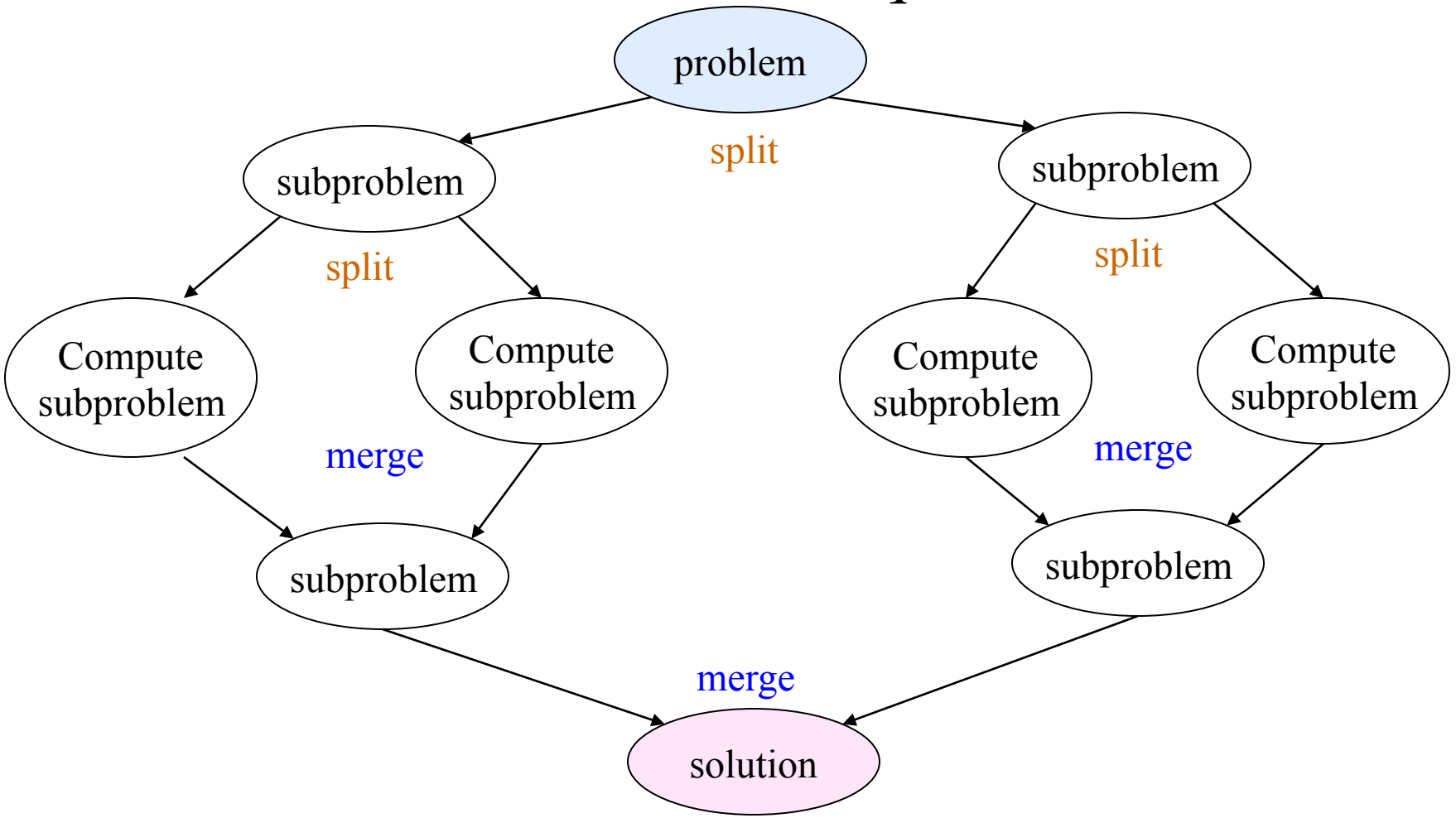


Taskurile de calcul nu pot accesa acelasi element simultan.
(Controlul accesului prin repository)

Descompunere Recursiva

- In general pentru probleme care se pot rezolva prin **divide-and-conquer**.

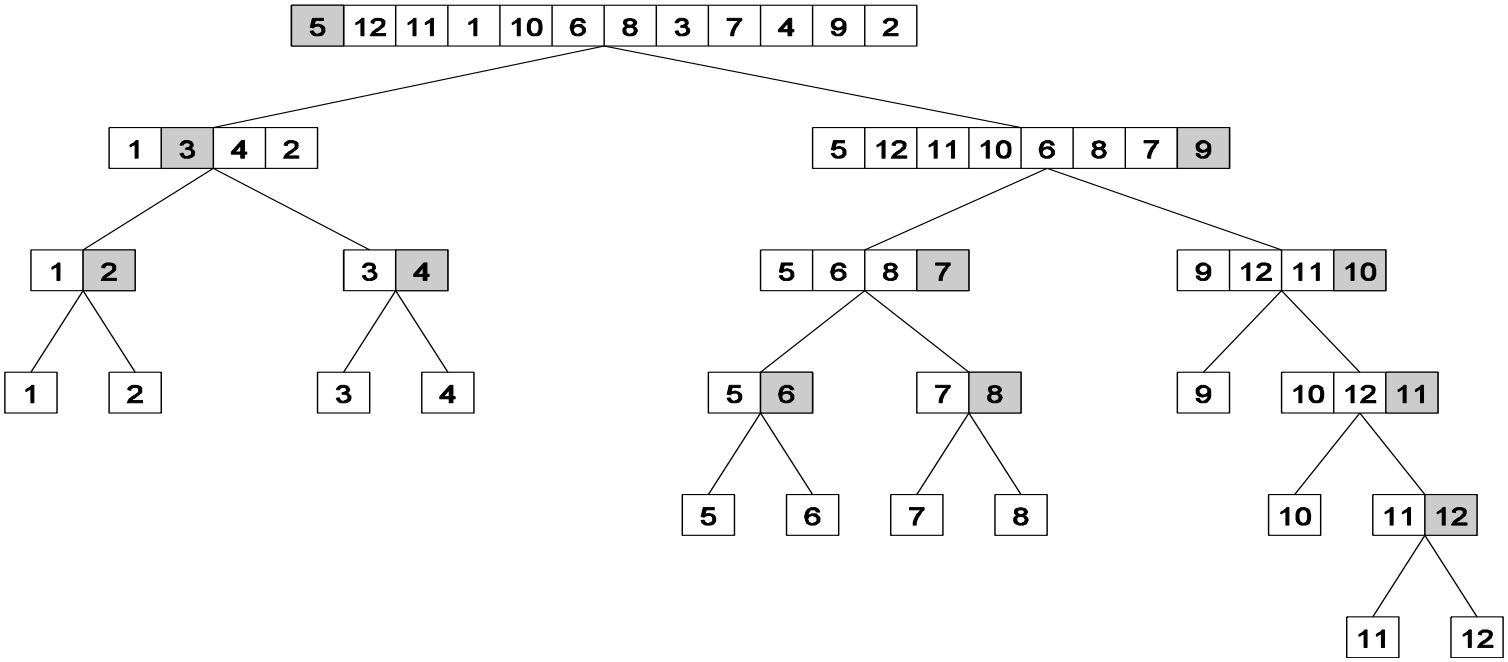
Divide & Conquer



* Nivelurile de descompunere trebuie sa fie ajustate corespunzator.

Recursive Decomposition: Exemplul 1

Quicksort.



Fiecare sublista reprezinta un task.

Recursive Decomposition: Exemplul 2

Cautarea minimului:

```
1. procedure SERIAL_MIN (A, n)
2. begin
3. min = A[0];
4. for i := 1 to n − 1 do
5.                 if (A[i] < min) min := A[i];
6. endfor;
7. return min;
8. end SERIAL_MIN
```

Recursive Decomposition

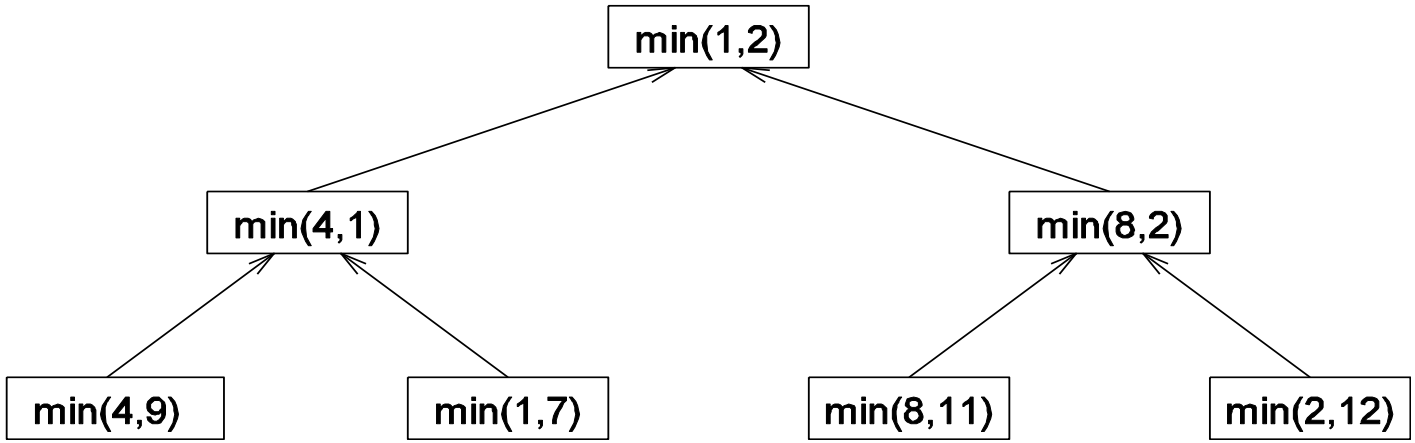
Rescriere

```
1. procedure RECURSIVE_MIN (A, n)
2. begin
3. if ( n = 1 ) then
4.     min := A [0]  ;
5. else
6.     lmin := RECURSIVE_MIN ( A, n/2 );
7.     rmin := RECURSIVE_MIN (  &(A[n/2]), n - n/2 );
8.     if (lmin  < rmin) then
9.                 min := lmin;
10.    else
11.                min := rmin;
12.    endelse;
13. endelse;
14. return min;
15. end RECURSIVE_MIN
```

Recursive Decomposition

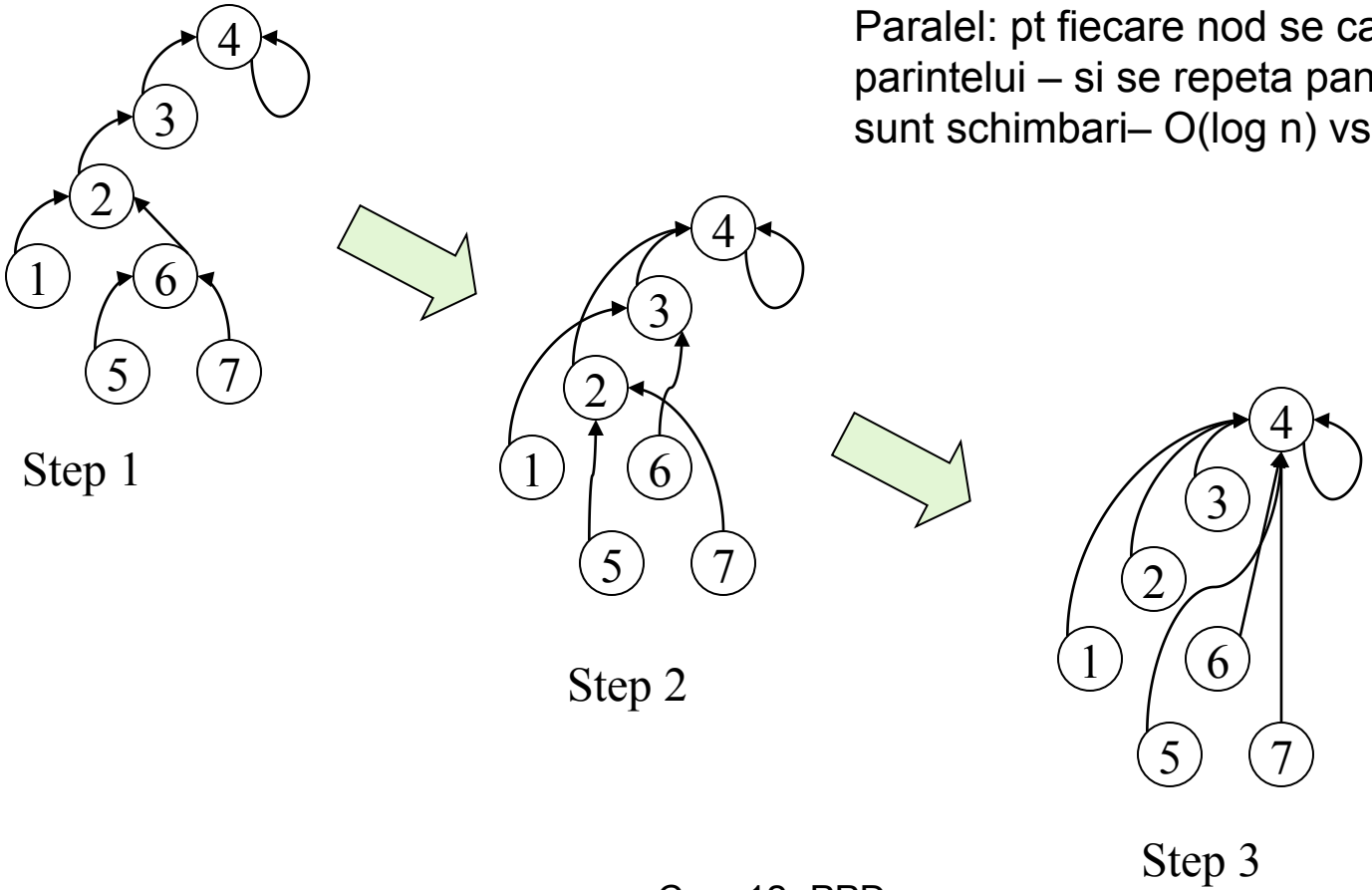
{4, 9, 1, 7, 8, 11, 2, 12}.

- task dependency graph



Recursive Data

Desi structurile de date recursive nu par a putea fi exploatate concurent (liste, arbori), in anumite cazuri se poate tranforma structura pentru a facilita paralelismul.



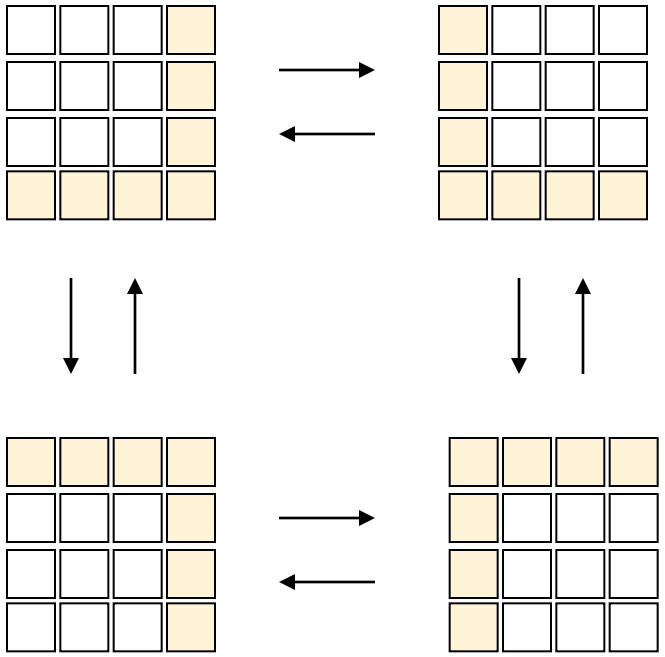
Paralel: pt fiecare nod se cauta parintele parintelui – si se repeta pana cand nu mai sunt schimbari– O(log n) vs. O(n)

Data Decomposition (geometric)

- Identificarea datelor implicate in calcul.
- Partitionarea datelor pe taskuri.
  - Diferite modalitati care implica impact important pt performanta.

Geometric

Exista dependente dar comunicarea se face intr-un mod predictibil (geometric) -> vecini.



Neighbor-To-Neighbor communication

Data Decomposition: Output Data Decomposition

output(i) = f_k (input)

- Partitionare naturala in special daca functiile sunt independente.

Output Data Decomposition: Exemplu

inmultire de matrice
n x ***n*** matricele ***A*** x ***B*** =***C***.

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

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

Output Data Decomposition: Exemplu

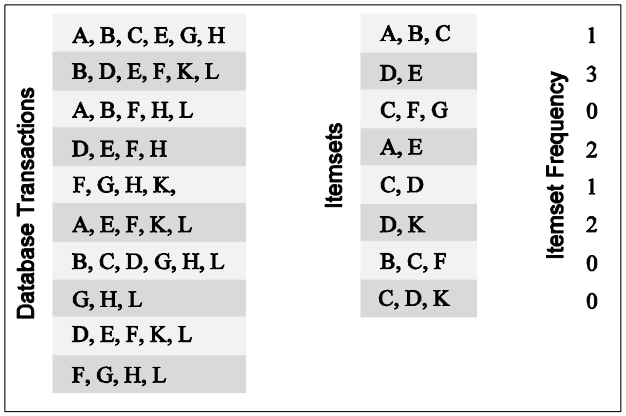
- nu exista doar o descompunere unica – variante!

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: $C_{2,2} = C_{2,2} + A_{2,2} B_{2,2}$

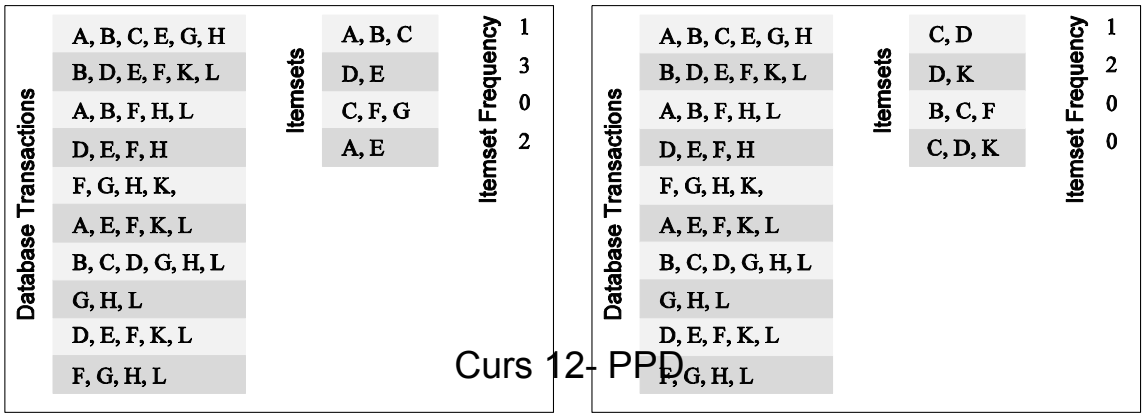
Output Data Decomposition: Exemplu

Problema: numararea instantelor intr-o baza de date de tranzactii.
Output= itemset frequencies
se partitioneaza intre taskuri.

(a) Transactions (input), itemsets (input), and frequencies (output)



(b) Partitioning the frequencies (and itemsets) among the tasks



Output Data Decomposition: Exemplu

Analiza:

- Daca baza de tranzactii este replicata pe procese fiecare task se poate executa fara comunicatii.
- Daca baza de tranzactii este distribuita pe procese (memory …) atunci fiecare task calculeaza frecvente partiale care trebuie ulterior agregate.

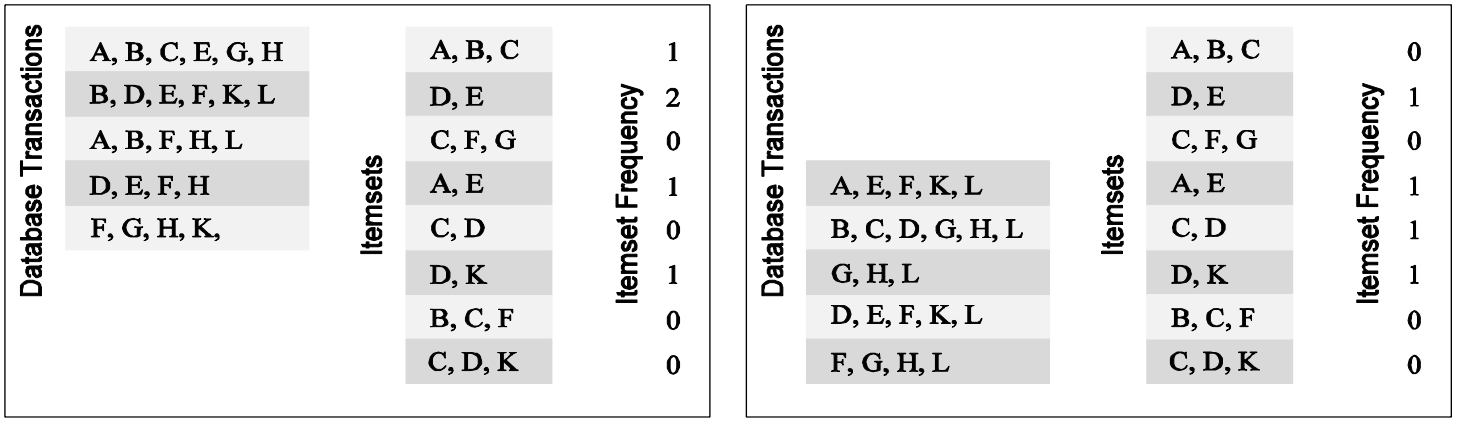
Input Data Partitioning

- Exemple: minim intr-o lista, sortare,....
- Task <=> partitie input

- Procese ulterioare pot agrega/ combina rezultatele partiale.

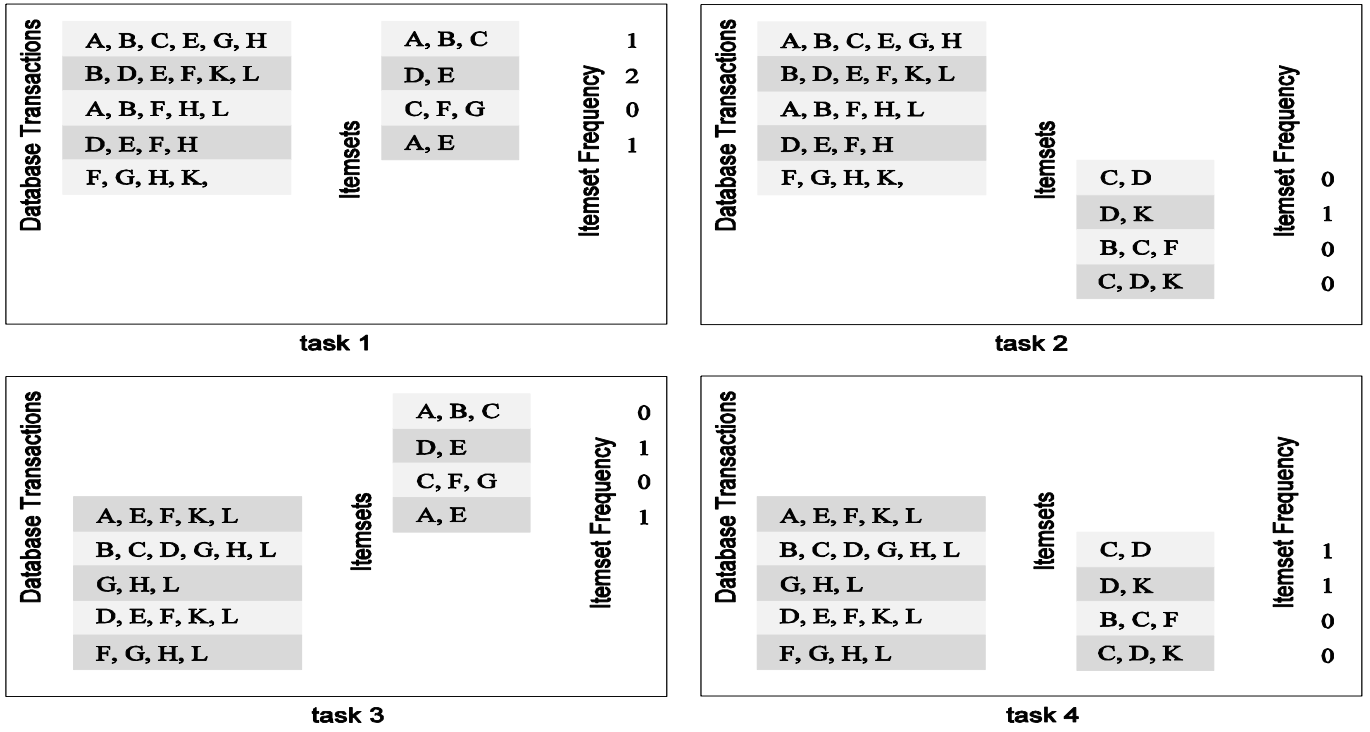
Input Data Partitioning: Exemplu

Partitioning the transactions among the tasks



Partitioning Input *and* Output Data

Partitioning both transactions and frequencies among the tasks



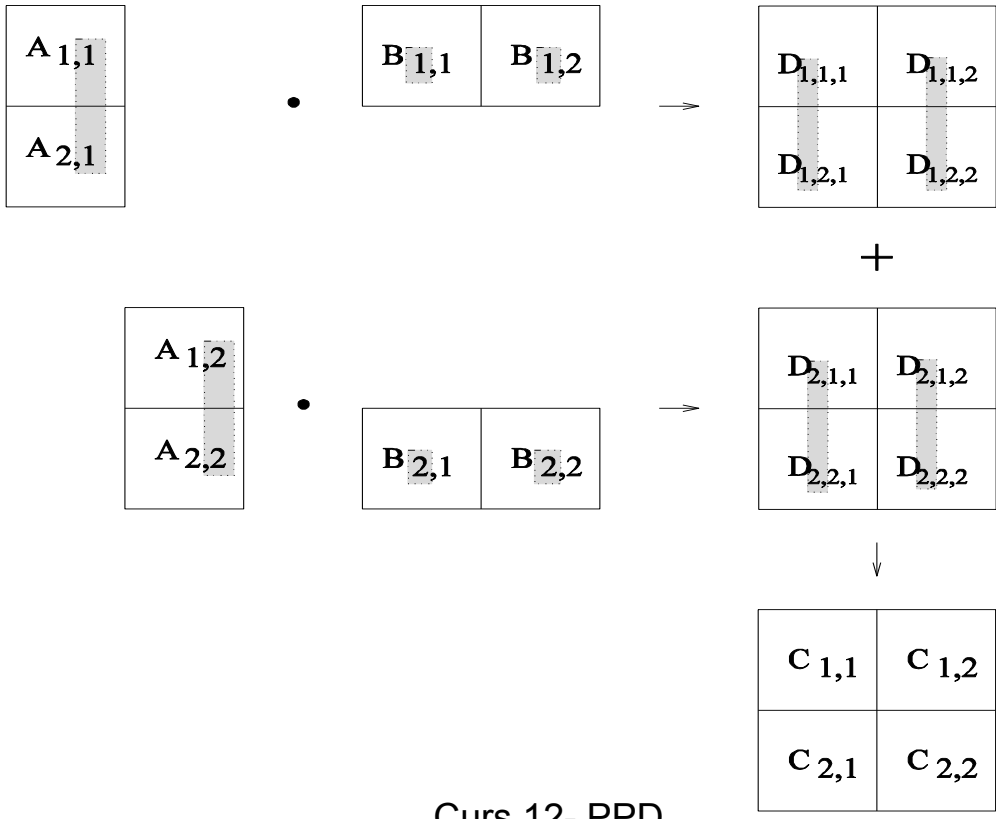
Intermediate Data Partitioning

- *Calcul = secventa de trasformari*
- *Baza de descompunere = stagiu intermediar*

Intermediate Data Partitioning: Exemplu

Inmultire de matrice

- matricea intermediara ***D***.



Intermediate Data Partitioning: Exemplu

Descompunerea matricei D conduce la o descompunere in 8 + 4 taskuri:

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

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

Task 01: $\boldsymbol{D}_{1,1,1} = \boldsymbol{A}_{1,1}\,\boldsymbol{B}_{1,1}$

Task 02: $\boldsymbol{D}_{2,1,1} = \boldsymbol{A}_{1,2}\,\boldsymbol{B}_{2,1}$

Task 03: $\boldsymbol{D}_{1,1,2} = \boldsymbol{A}_{1,1}\,\boldsymbol{B}_{1,2}$

Task 04: $\boldsymbol{D}_{2,1,2} = \boldsymbol{A}_{1,2}\,\boldsymbol{B}_{2,2}$

Task 05: $\boldsymbol{D}_{1,2,1} = \boldsymbol{A}_{2,1}\,\boldsymbol{B}_{1,1}$

Task 06: $\boldsymbol{D}_{2,2,1} = \boldsymbol{A}_{2,2}\,\boldsymbol{B}_{2,1}$

Task 07: $\boldsymbol{D}_{1,2,2} = \boldsymbol{A}_{2,1}\,\boldsymbol{B}_{1,2}$

Task 08: $\boldsymbol{D}_{2,2,2} = \boldsymbol{A}_{2,2}\,\boldsymbol{B}_{2,2}$

Task 09: $\boldsymbol{C}_{1,1} = \boldsymbol{D}_{1,1,1} + \boldsymbol{D}_{2,1,1}$

Task 10: $\boldsymbol{C}_{1,2} = \boldsymbol{D}_{1,1,2} + \boldsymbol{D}_{2,1,2}$

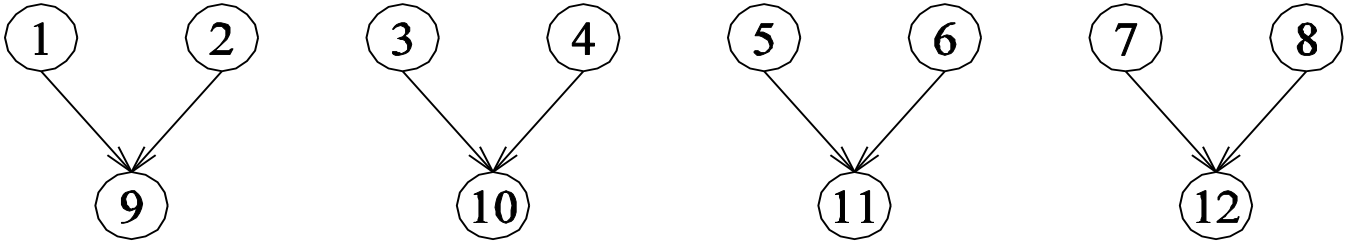
Task 11: $\boldsymbol{C}_{2,1} = \boldsymbol{D}_{1,2,1} + \boldsymbol{D}_{2,2,1}$

Task 12: $\boldsymbol{C}_{2,,2} = \boldsymbol{D}_{1,2,2} + \boldsymbol{D}_{2,2,2}$

Intermediate Data Partitioning: Exemplu

- task dependency graph

pentru cele 12 taskuri



The Owner Computes Rule

- Procesul care are data asignata lui este responsabil cu calculele asociate acelei date
- Diferente:
  - input data decomposition
  - output data decomposition

Exploratory Decomposition

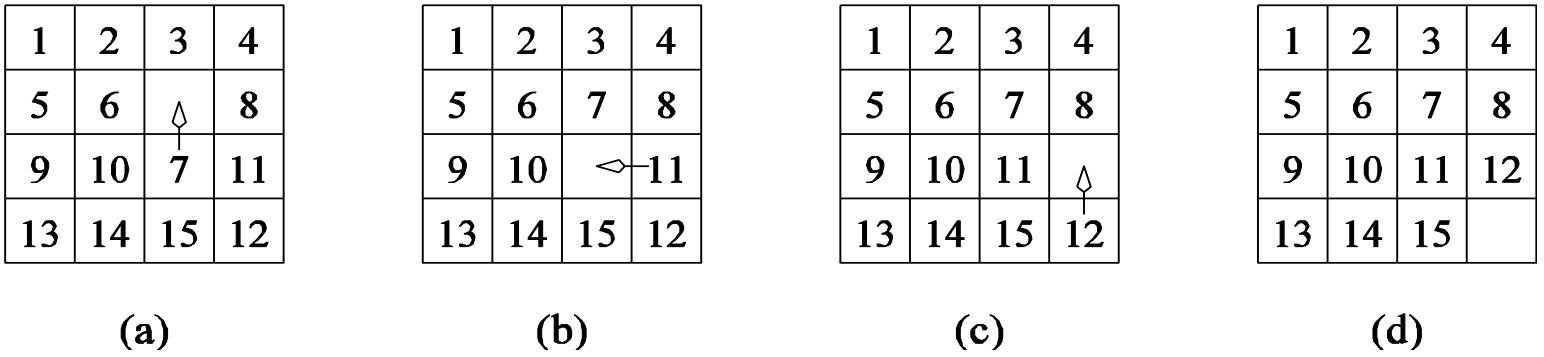
- Descompunerea merge in acelasi timp cu executia.

- Probleme care implica cautare exploratorie intr-un saptiu de solutii.
-

Exploratory Decomposition: Exemplu

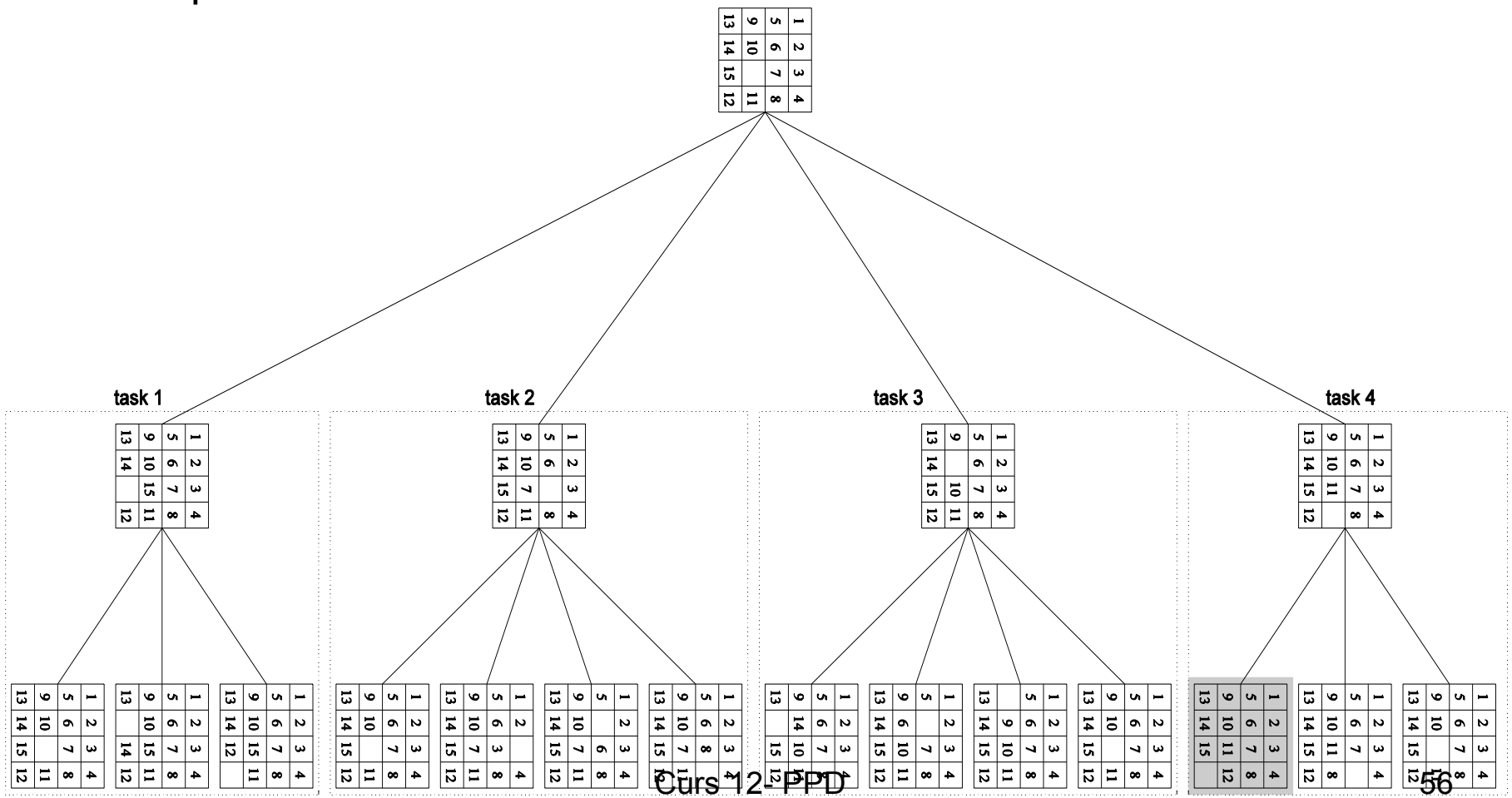
15 puzzle (a tile puzzle).

Exemplicare – 3 mutari care transforma starea initiala (a) in starea finala to (d).



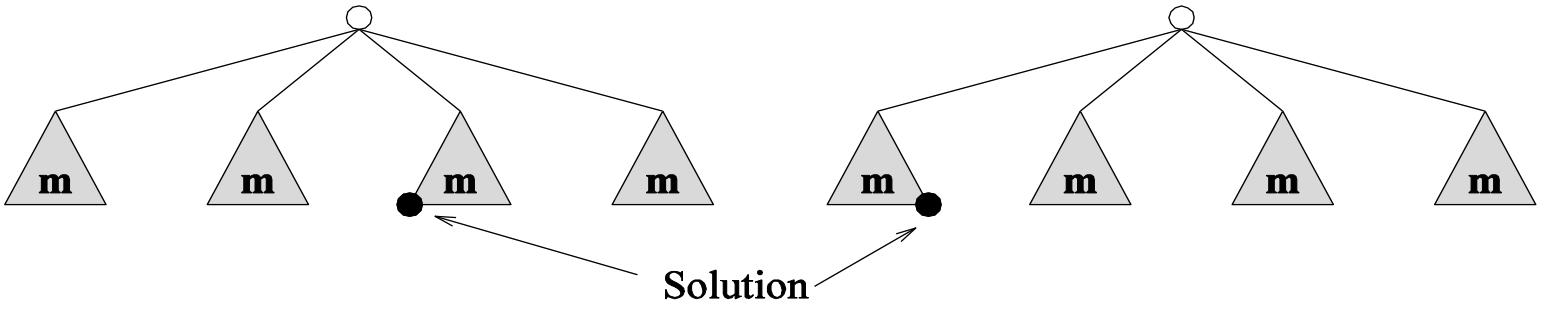
Exploratory Decomposition: Example

- se genereaza variantele posibile de succesiune ca si taskuri independente.



Exploratory Decomposition: performante greu de anticipat

- In multe cazuri se poate modifica cantitatea de calcul executata in varianta paralela
- Se poate ajunge la
  - super- ori
  - sub-linear speedups.



Total serial work: 2m+1
Total parallel work: 1

(a)

Total serial work: m
Total parallel work: 4m

(b)

Speculative Decomposition

- In anumite cazuri dependenta intre taskuri nu se cunoaste apriori.
  - Nu se pot identifica taskurile paralele.
- Doua abordari:
  - *conservatoare*, prin care se identifica taskurile doar atunci cand se stie cu siguranta ca nu sunt alte dependente
    - Se poate ajunge la DOP mic
  - *Optimiste*, prin care se creeaza/programeaza taskurile chiar daca exista un potential ca ele sa nu mai fie necesare.
    - Poate sa necesite mecanisme *de roll-back* in caz de eroare.

Speculative Decomposition: Exemplu

discrete event simulation

- Structura centrala = a time-ordered event list.
- Evenimentele se extrag in ordine, procesate si daca este necesar evenimentele rezultate sunt inserate in lista de evenimente.
- Exemplu: ctivitatile zilnice

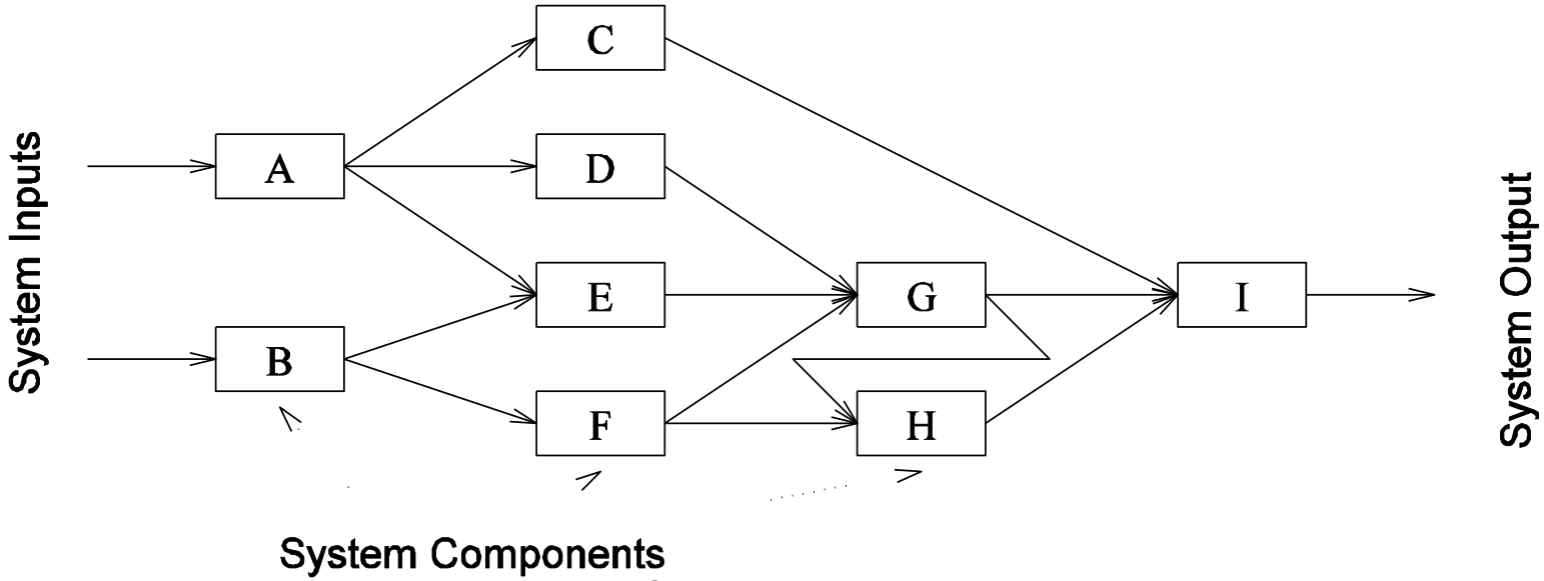
- you get up, get ready, drive to work, work, eat lunch, work some more, drive back, eat dinner, and sleep.

- Fiecare eveniment poate fi procesat independent dar in cazul aparitiei unui eveniment special (accident pe traseu catre munca) atunci evenimentele legate de job trebuie anulate (rolled back).

Speculative Decomposition: Exemplu

Simularea unei retele de noduri (linie de asamblare sau retea de comunicatie – pachete etc.)

Simularea comportamentului pt diferite valori de input si parametrii de delay (reteaua poate deveni instabila pentru anumite rate de transfer sau dimensiune a cozilor, etc.).

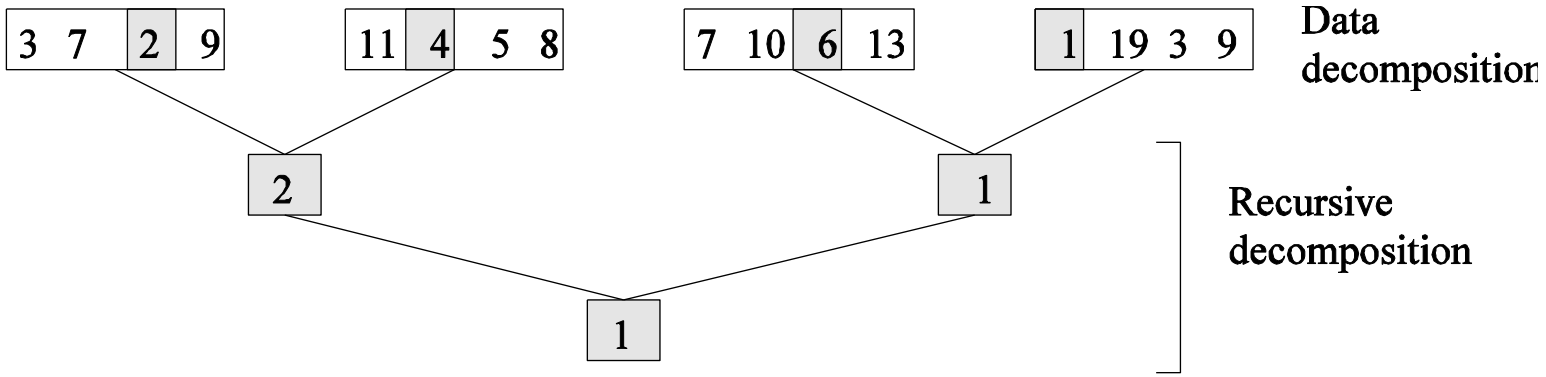


Hybrid Decompositions

Ex.

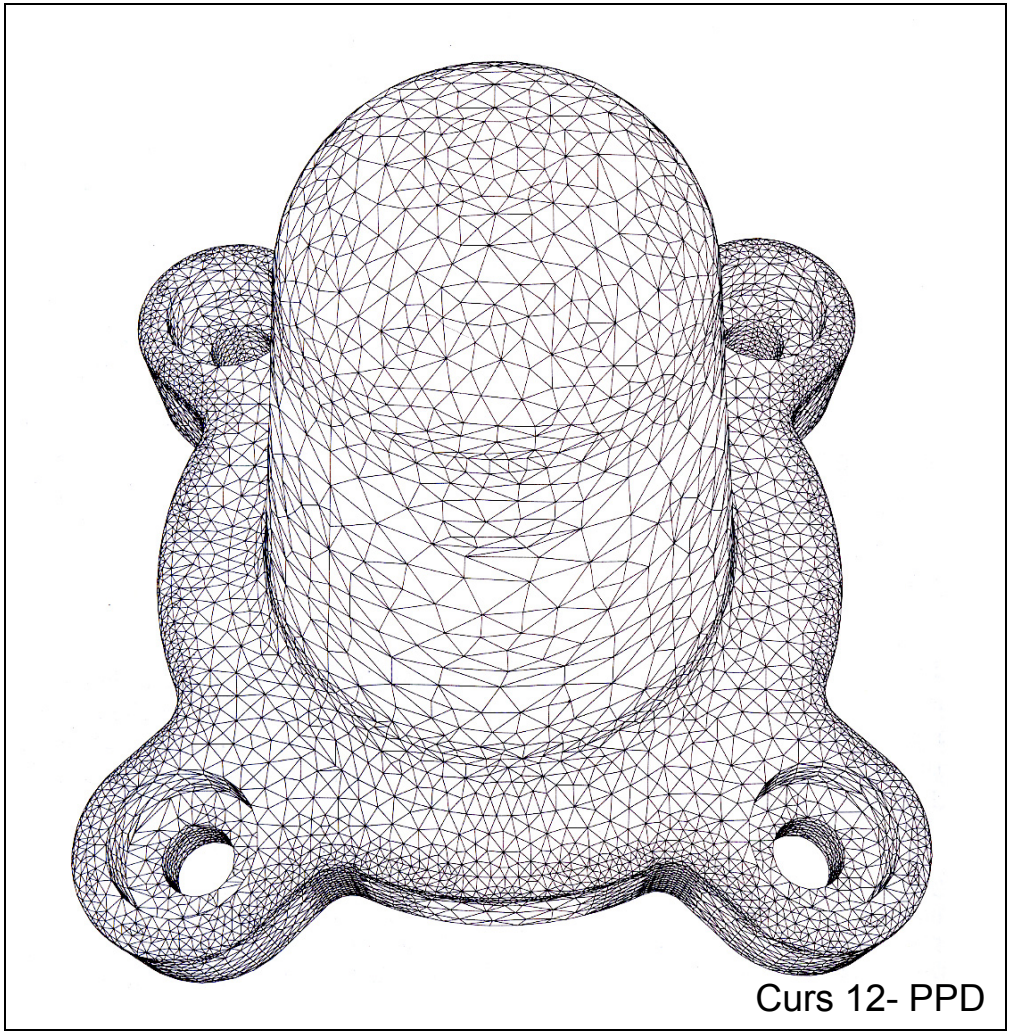
- *In quicksort, recursive decomposition alone limits concurrency . A mix of data and recursive decompositions is more desirable.*
- *In discrete event simulation, there might be concurrency in task processing. A mix of speculative decomposition and data decomposition may work well.*

- Minim



Irregular Mesh

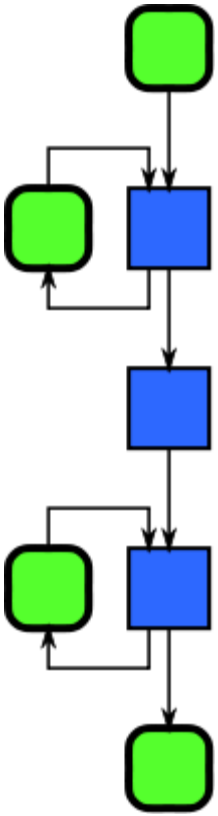
Communicatrea se face pe cai nepredictibile intr-o topologie de tip mesh.



Dificil de definit pentru ca sabloanele de comunicare difera in timp.

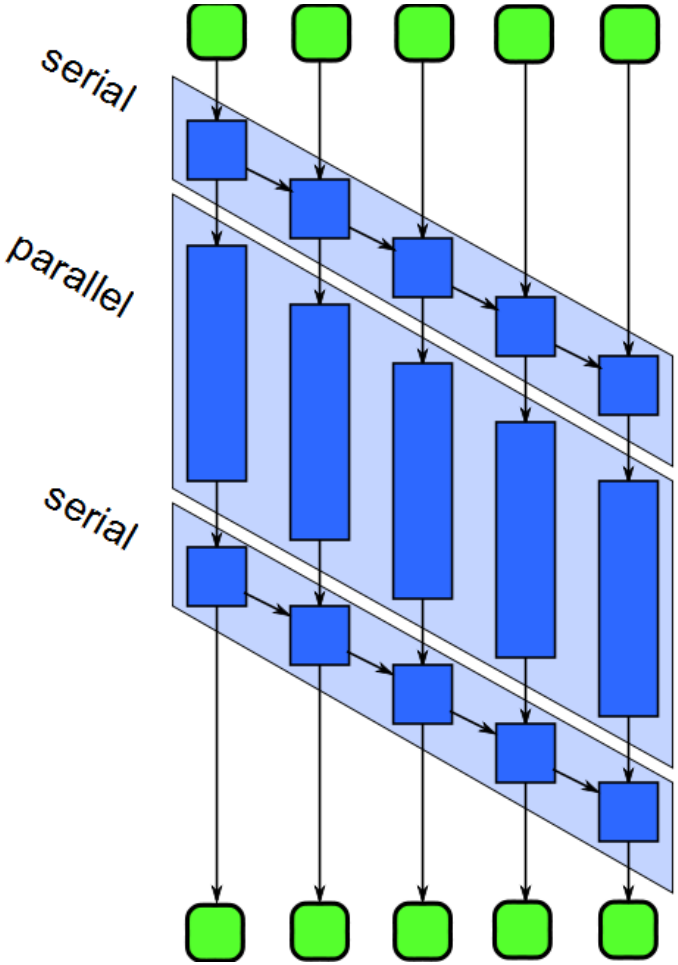
Un punct de plecare poate fi sablonul de constructie a topologiei.

Pipeline



- *Pipeline* – o secventa de stagii care transforma un flux de date
- Unele stagii pot sa stocheze stare
- Datele pot fi “consumate” si produse incremental.

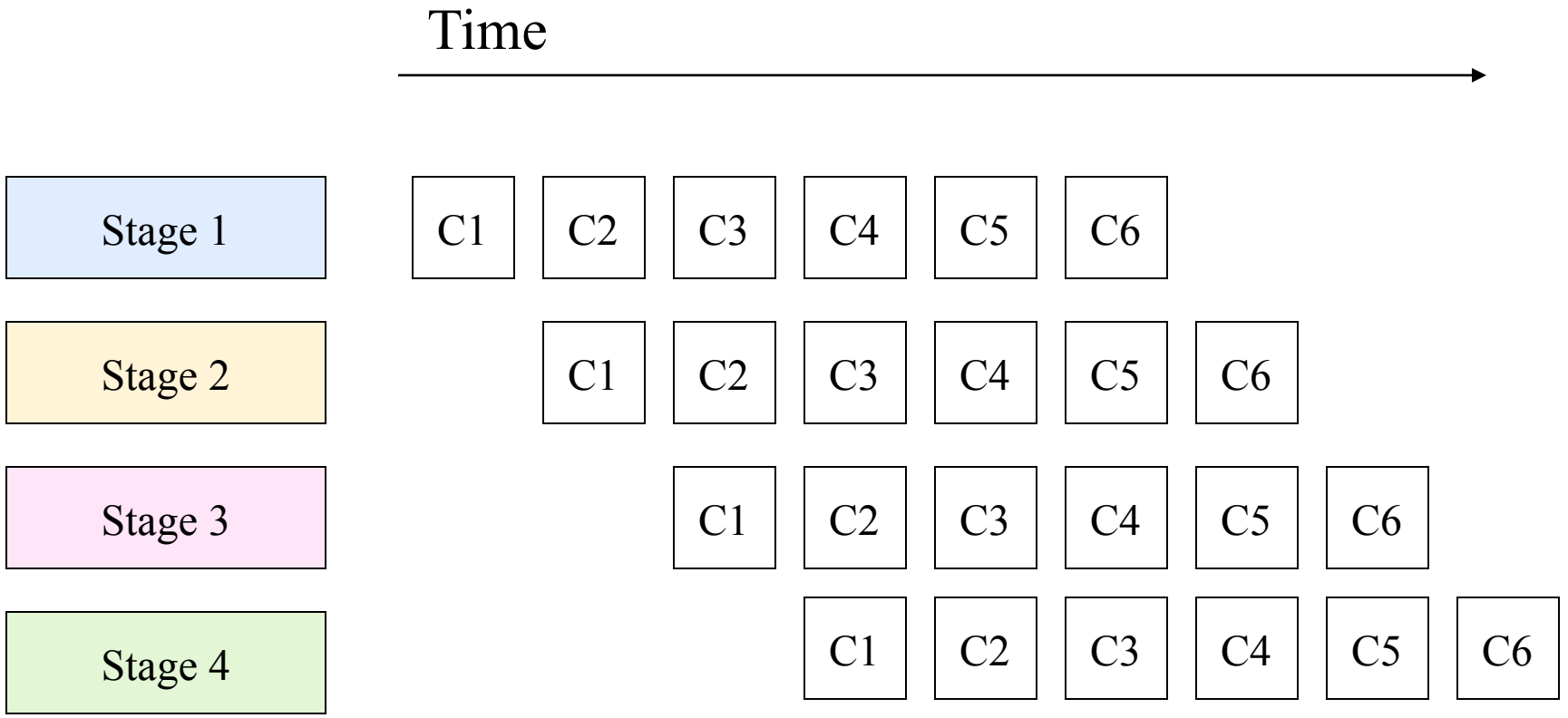
Pipeline



- Paralelizarea pipeline se face prin
  1. Executia diferitelor stagii in paralel
  2. Executia multiplelor copii ale stagiilor fara stare in paralel

Pipeline

A series of ordered but independent computation stages need to be applied on data, where each output of a computation becomes input of subsequent computation.



Sumar - descompunere

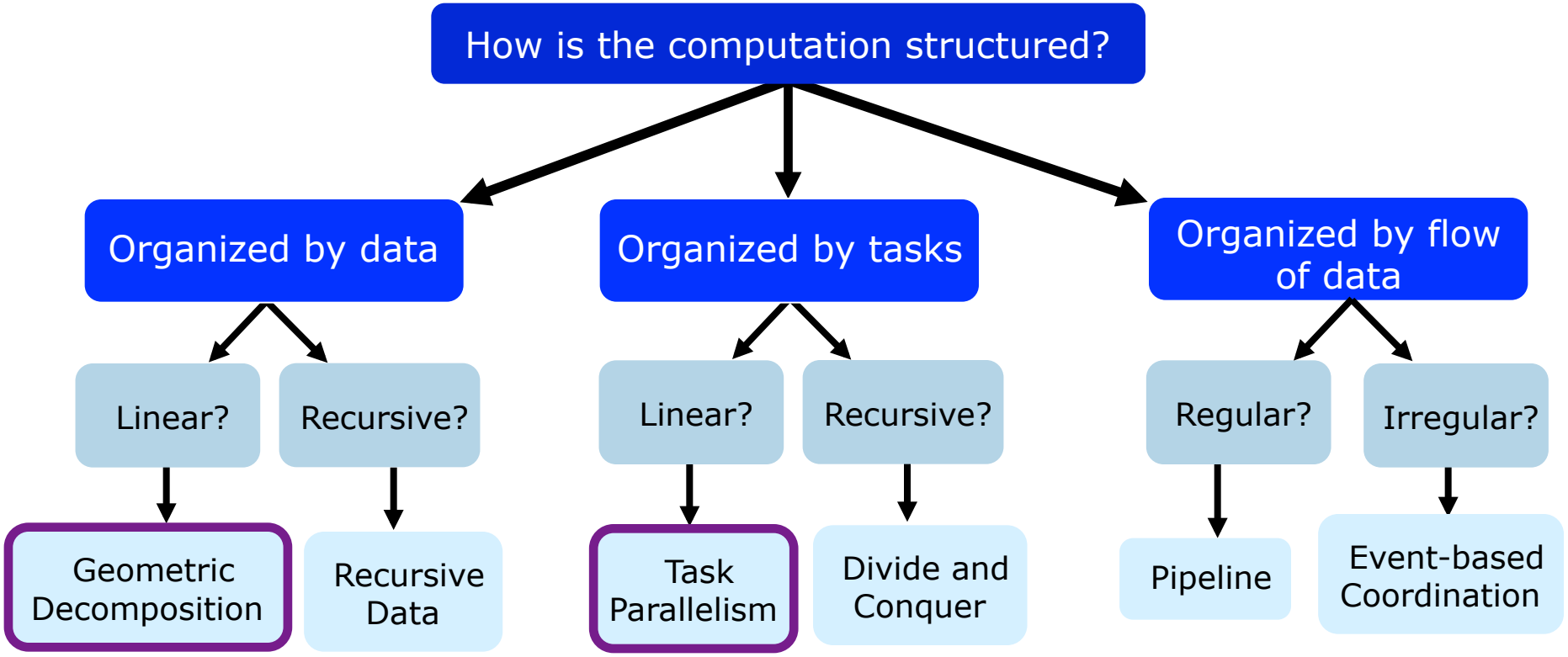
- Nu exista doar o singura reteta pentru descompunere
- Se pot aplica un set de tehnici comune pe o clasa de probleme mai vasta.

:

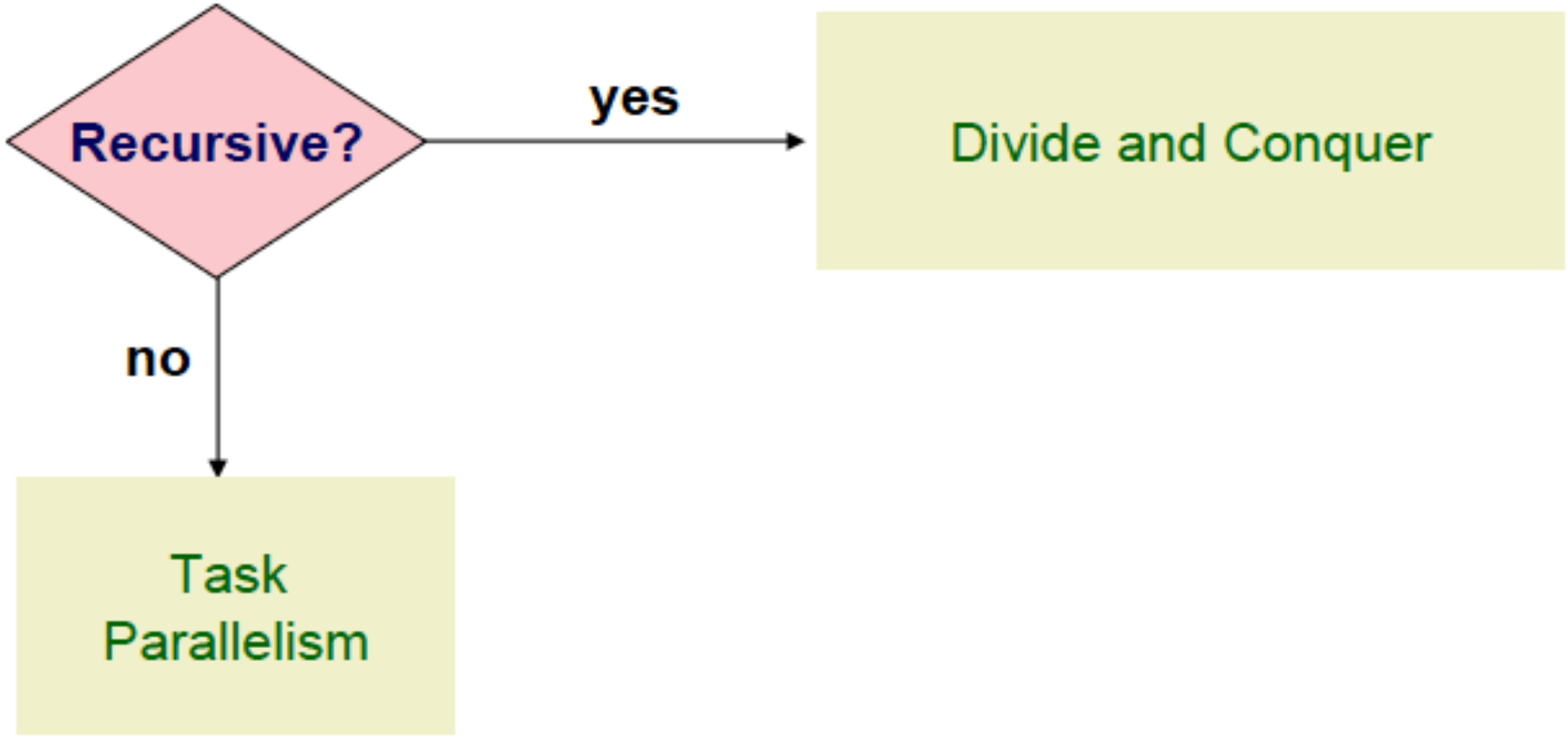
- *recursive decomposition*
- *data decomposition (geometric decomposition)*
- *exploratory decomposition*
- *speculative decomposition*

- *Pipelines…*

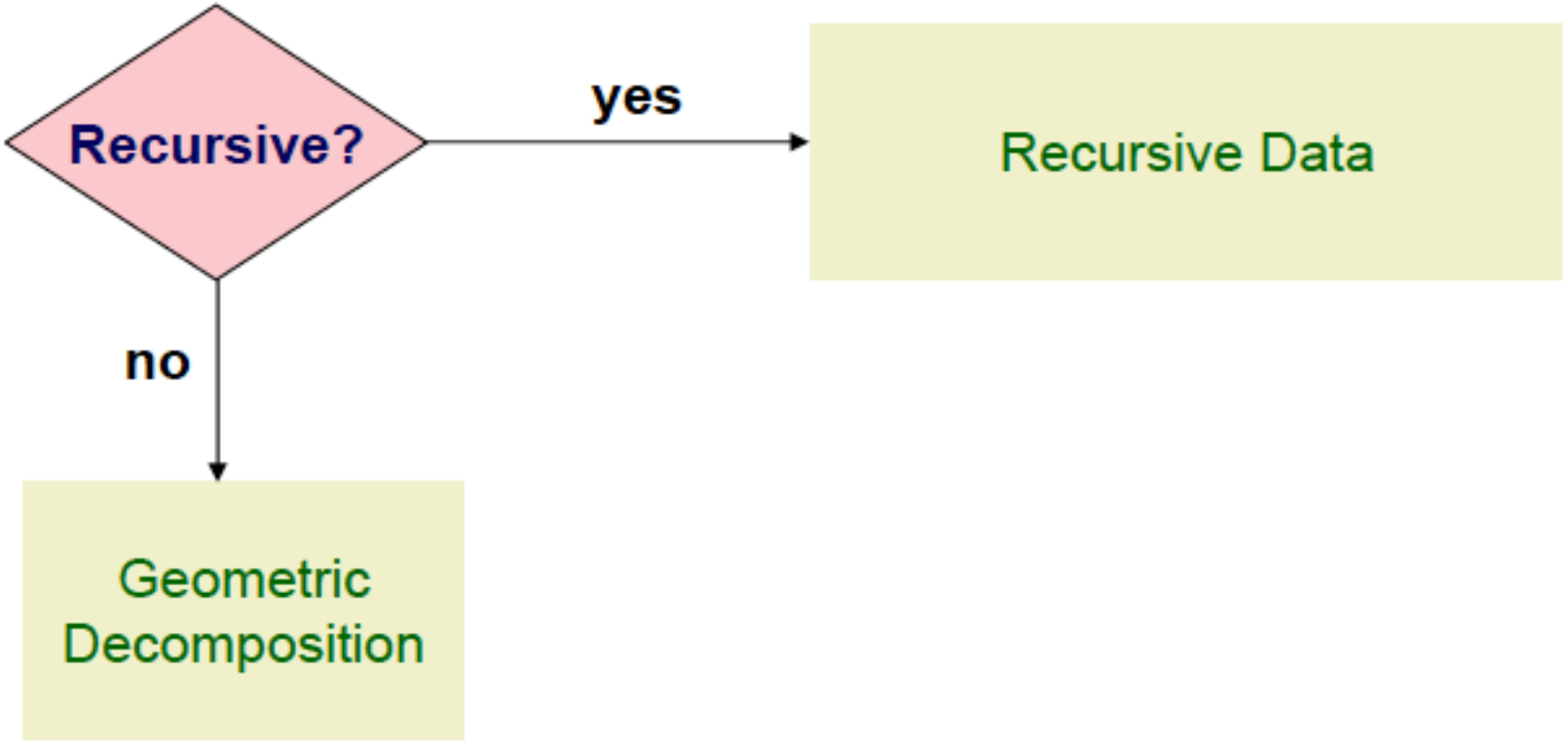
Algorithm Structure Design Space



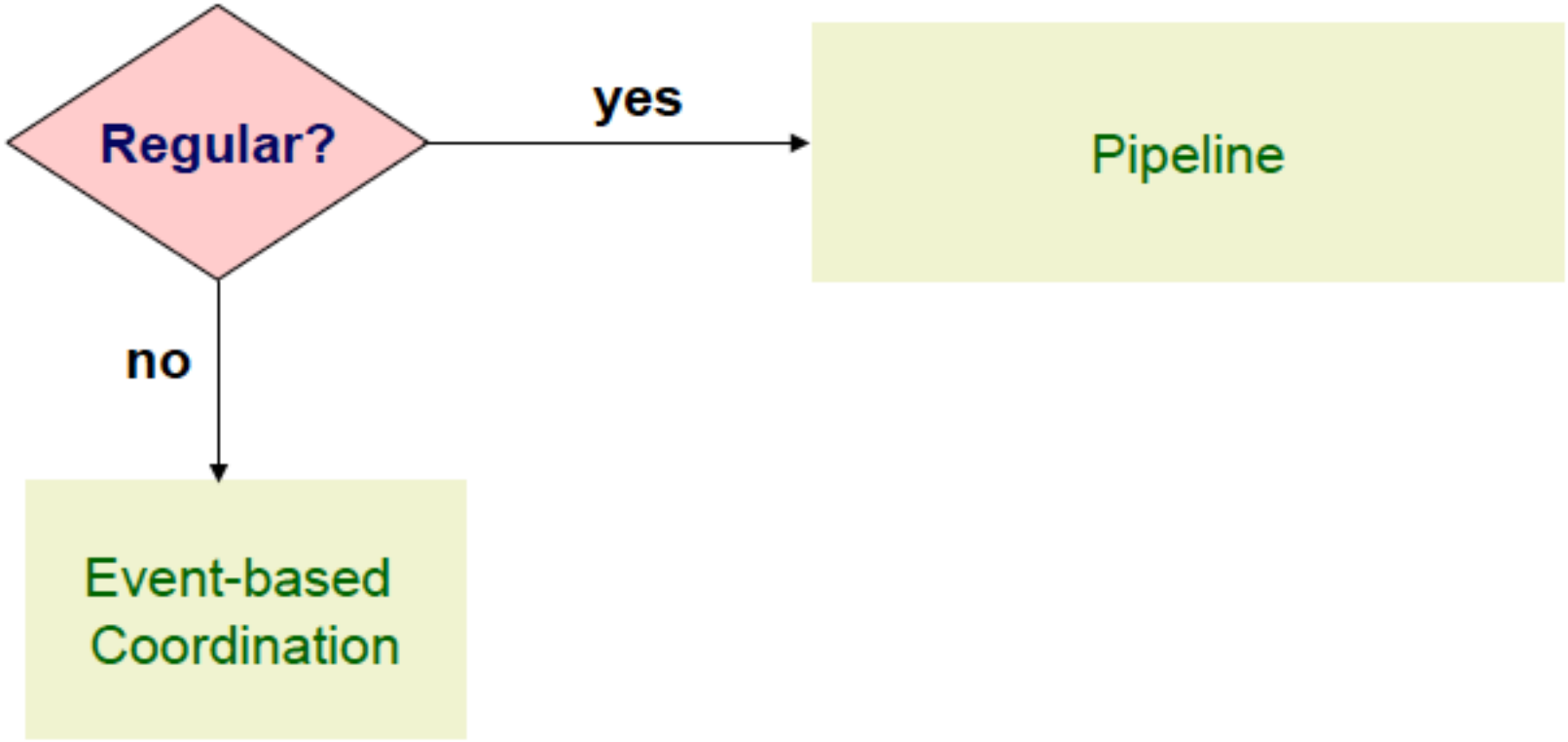
Descompunere functionala

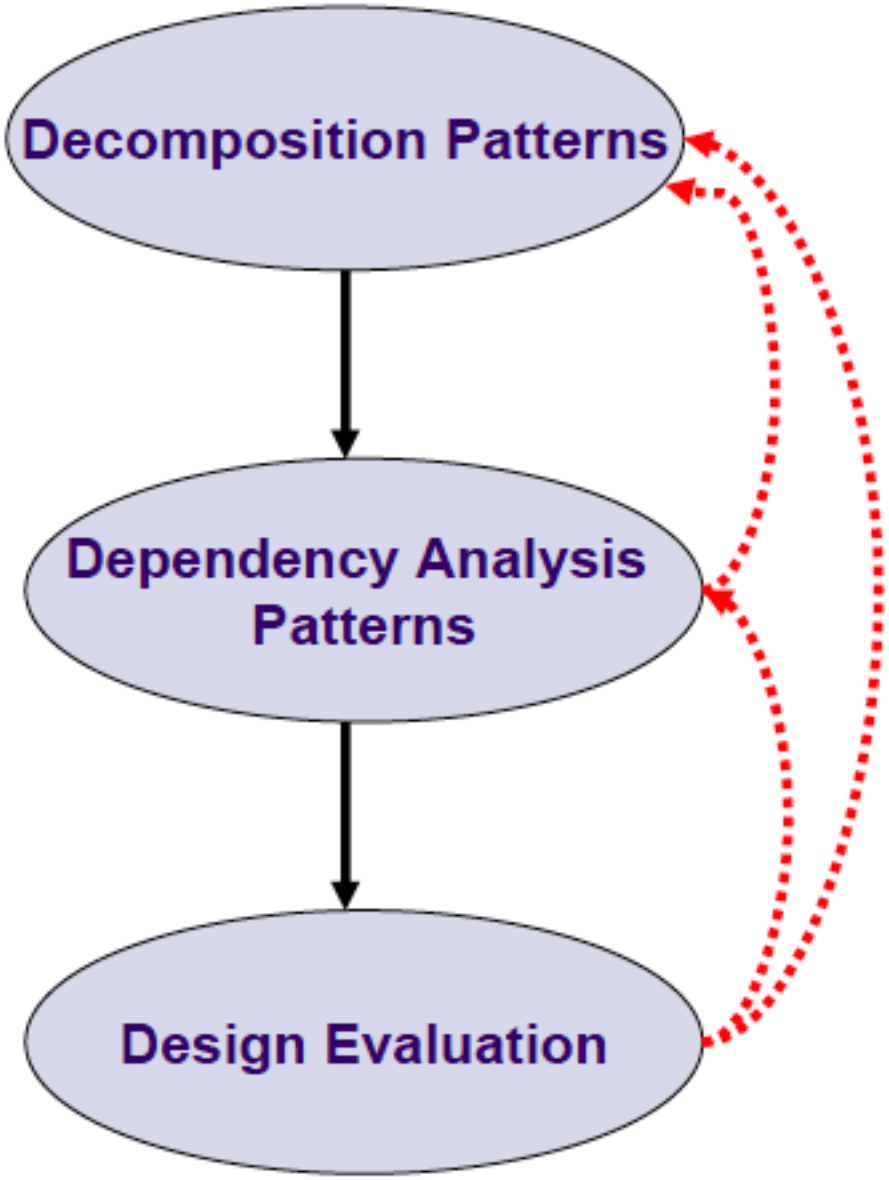


Descompunerea datelor

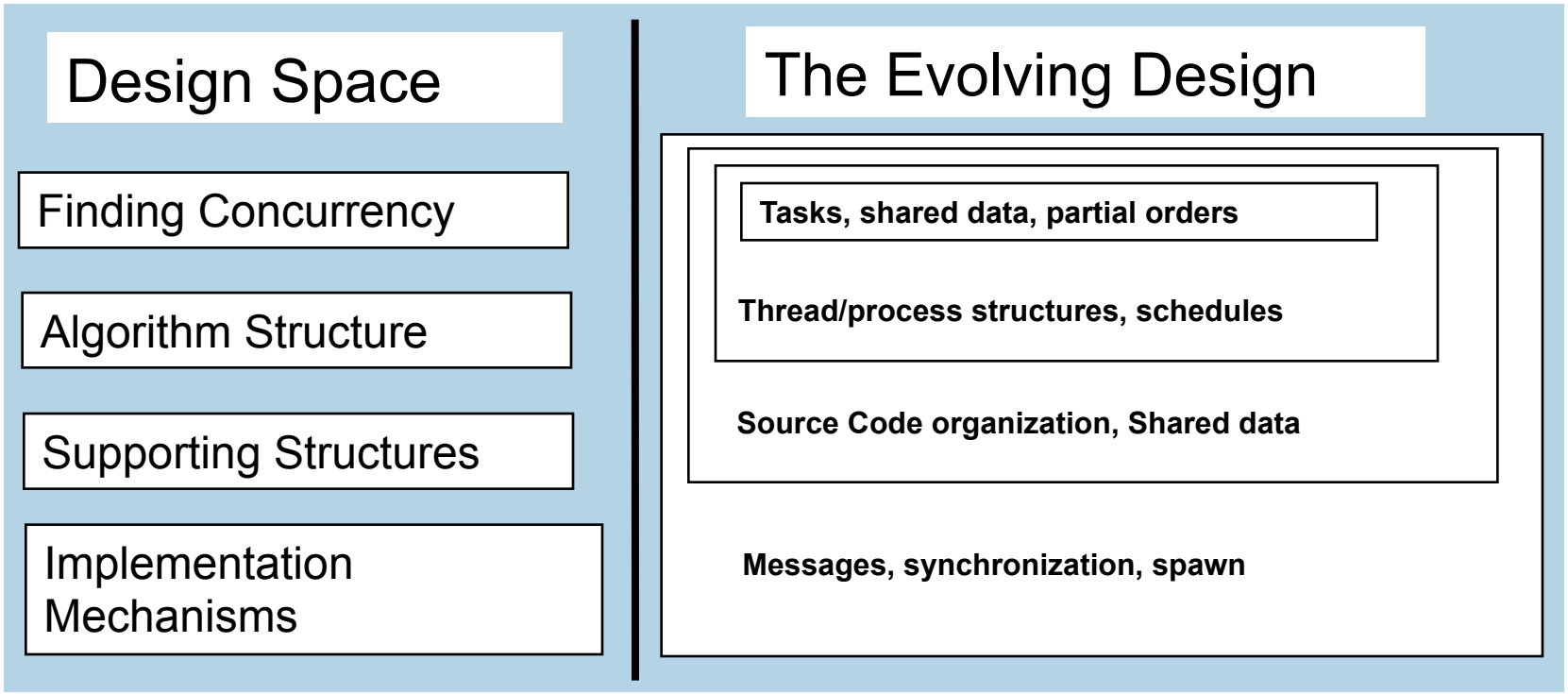


Descompunere bazata pe *Data-Flow*




Decomposition Patterns
Dependency Analysis Patterns
Design Evaluation

Patterns for Parallel Programming. Mattson, Sanders, and Massingill (2005).



4 Design Spaces

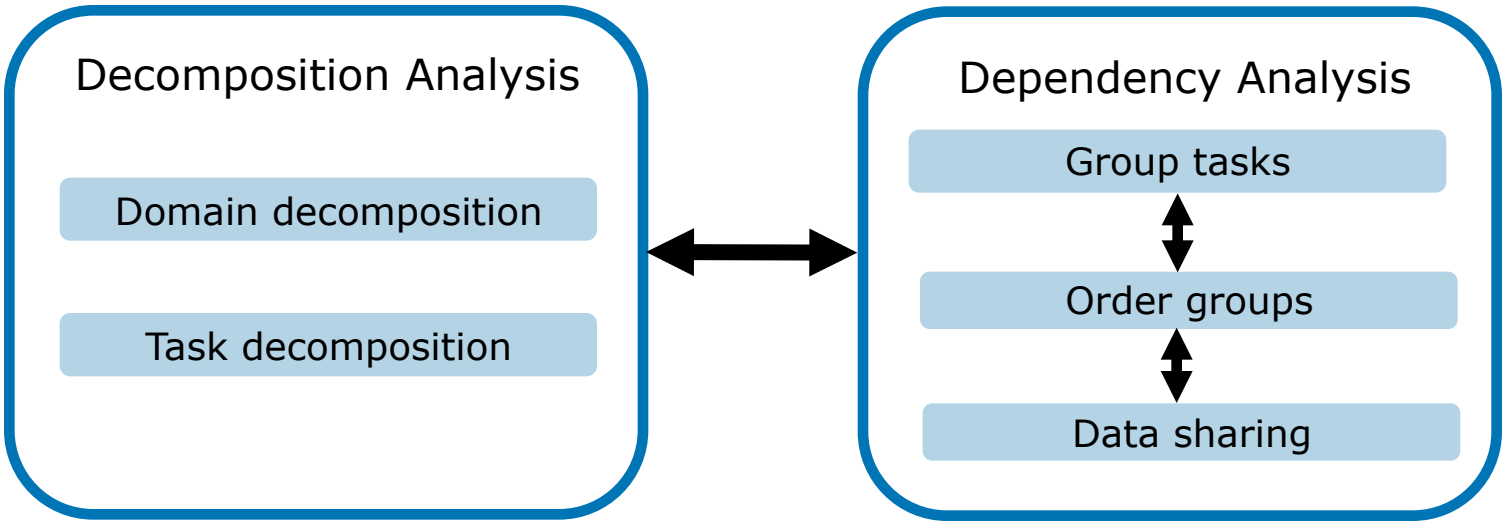
Algorithm Expression

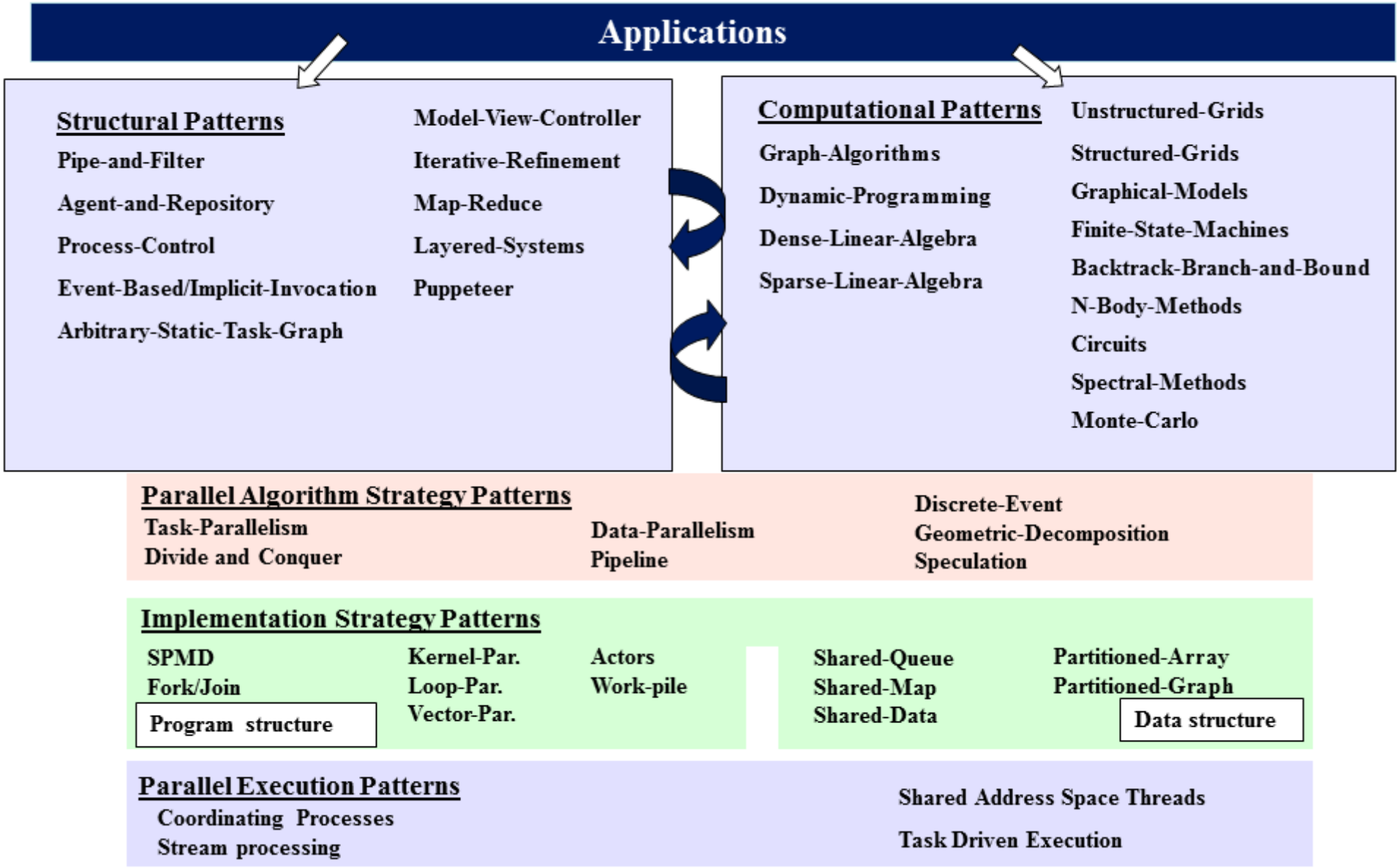
- Finding Concurrency
  - Expose concurrent tasks
- Algorithm Structure
  - Map tasks to processes to exploit parallel architecture

Software Construction

- Supporting Structures
  - Code and data structuring patterns
- Implementation Mechanisms
  - Low level mechanisms used to write parallel programs

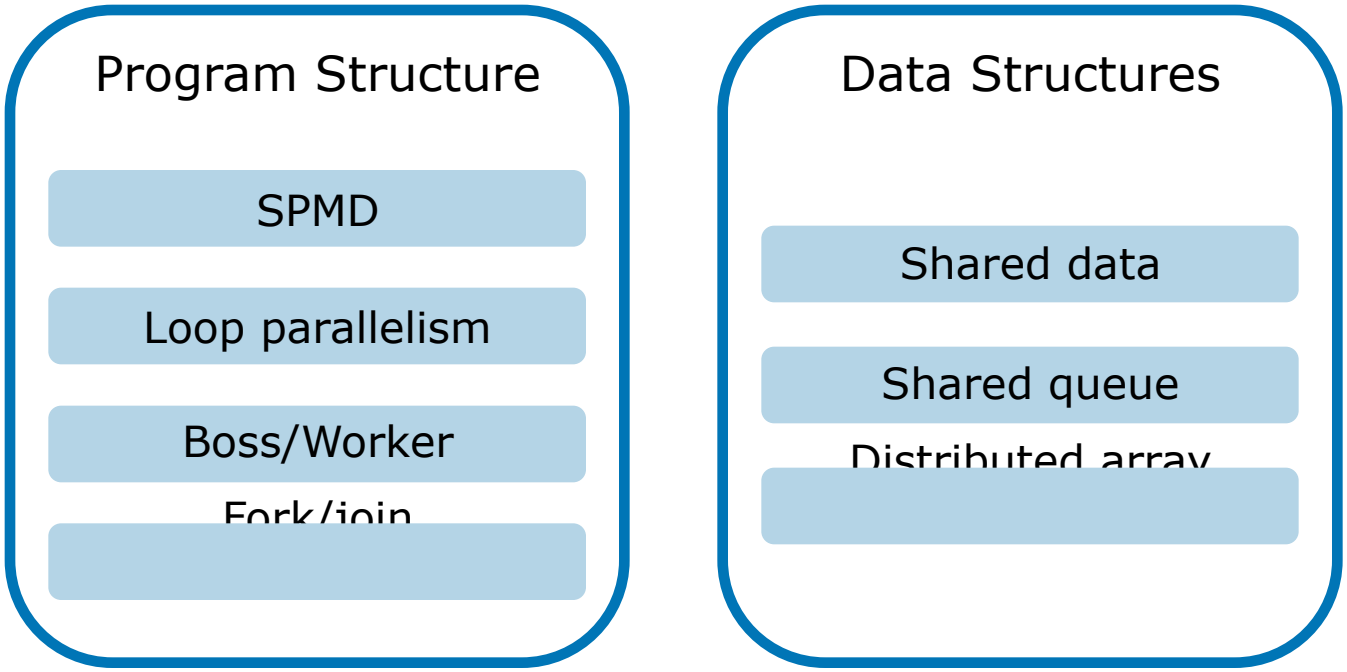
Cautare in spatiul de proiectare concurenta




Applications
Structural Patterns
Pipe-and-Filter
Agent-and-Repository
Process-Control
Event-Based/Implicit-Invocation
Arbitrary-Static-Task-Graph
Model-View-Controller
Iterative-Refinement
Map-Reduce
Layered-Systems
Puppeteer
Computational Patterns
Graph-Algorithms
Dynamic-Programming
Dense-Linear-Algebra
Sparse-Linear-Algebra
Unstructured-Grids
Structured-Grids
Graphical-Models
Finite-State-Machines
Backtrack-Branch-and-Bound
N-Body-Methods
Circuits
Spectral-Methods
Monte-Carlo
Parallel Algorithm Strategy Patterns
Task-Parallelism
Divide and Conquer
Data-Parallelism
Pipeline
Discrete-Event
Geometric-Decomposition
Speculation
Implementation Strategy Patterns
SPMD
Fork/Join
Program structure
Kernel-Par.
Loop-Par.
Vector-Par.
Actors
Work-pile
Shared-Queue
Shared-Map
Shared-Data
Partitioned-Array
Partitioned-Graph
Data structure
Parallel Execution Patterns
Coordinating Processes
Stream processing
Shared Address Space Threads
Task Driven Execution

Supporting Structures Design Space

- High-level constructs used to organize the source code
- Categorized into program structures and data structures



Referinte:

``Introduction to Parallel Computing''
Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar
2003

Ian Foster
Designing and Building Parallel Programs, Addison Wesley, 2, Addison-Wesley Inc., Argonne National Laboratory (http://www.mcs.anl.gov/~itf/dbpp/)

Parallel Programming Patterns
Eun-Gyu Ki
2004

Patterns for Parallel Programming. Mattson, Sanders, and Massingill (2005).