Procesare Paralelă

Capitolul 2: Modele de programare: SPMD

Bogdan Dumitrescu

Facultatea de Automatică şi Calculatoare

Universitatea Politehnica Bucureşti

Cuprins

- Modelul SPMD (Single Program Multiple Data)
- Comunicaţie prin mesaje
- Message Passing Interface
- Comunicaţie globală

Modelul SPMD

- Model adecvat calculatoarelor MIMD
- Ne vom ocupa doar de cazul MIMD cu memorie distribuită
- Paradigma SPMD (Single Program Multiple Data): toate procesoarele execută acelaşi program, dar fiecare utilizează un set propriu de date
- Precizări importante:
  - Execuţia instrucţiunilor nu este sincronă, adică fiecare procesor execută instrucţiunile în ritmul său propriu
  - Nu toate procesoarele execută aceleaşi instrucţiuni; deşi programele încărcate în memoriile procesoarelor sunt identice, totuşi pot exista diferenţe la execuţie

Adresa unui procesor

- Un program se execută pe p procesoare ale unui calculator paralel
- In general, valoarea lui p este mai mică decât numărul total de procesoare al calculatorului
- Procesoarele sunt numerotate de la 0 la $p-1$
- Numerotarea nu este statică, ci se face în momentul lansării în execuţie
- Procesoarele pot să-şi afle adresa proprie printr-o primitivă a sistemului de operare sau de bibliotecă
- In descrierea algoritmilor, adresa procesorului care execută programul este utilizată ca o variabilă
- Procesoarele pot executa instrucţiuni diferite în funcţie de adresa lor

Diferenţierea la execuţie

- Notăm id adresa proprie
- Exemplu simplu de program

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

- Doar procesorul P_3 execută instrucţiunea $a \leftarrow 1$
- Toate celelalte execută $a \leftarrow 0$
- Atenţie: variabila a este în memoria locală a fiecărui procesor!

Variabile şi indici

- Aşadar, o variabilă a unui program SPMD este multiplicată: fiecare procesor deţine un exemplar, asupra căruia are control complet
- Un procesor nu poate modifica variabile din memoria altui procesor
- In exemplul de mai sus, putem interpreta variabila a ca un vector cu p elemente, fiecare procesor deţinând doar un element, anume cel al cărui indice este egal cu adresa procesorului
- Programul de mai sus iniţializează vectorul a cu zero, mai puţin elementul a_3 care devine 1
- Vom folosi indici locali în programe
- Totuşi, în prezentarea unor algoritmi vom folosi şi indici globali

Problemă

- Vrem să iniţializăm un vector a de dimensiune n cu zero, mai puţin elementul a_m, care să fie 1
- Intrebare 1: cum distribuim vectorul a celor p procesoare ?
- Intrebare 2: cum facem efectiv iniţializarea ?
- Răspunsuri: la curs

Model de comunicaţie prin mesaje

- Model exclusiv pentru calculatoare MIMD cu memorie distribuită
- Deoarece fiecare procesor are memoria sa proprie, singura modalitate de comunicare între procesoare este transmiterea de mesaje
- Operaţia de bază este cea în care un procesor sursă P_s transmite un mesaj M conţinând date din memoria sa M_s unui procesor destinaţie P_a, care stochează datele în memoria sa M_a
- Când vom discuta algoritmi de comunicaţie vom presupune că sursa şi destinaţia sunt vecine
- Vom scrie astfel algoritmi eficienţi pe anume topologii

Primitive de bază

- Sunt suficiente doar doua rutine pentru a scrie orice operaţie de comunicaţie
- Procesorul sursă transmite prin rutina **send**
- Procesorul destinaţie recepţionează prin rutina **recv**
- Sintaxa

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

- *date* este o variabilă locală, indicând locul în care se află datele transmise sau unde se memorează datele recepţionate
- Lungimea datelor va rezulta din context
- Presupunem implicit că datele ocupă o zonă contiguă în memorie

Primitive de bază

- Al doilea parametru identifică vecinul cu care se comunică
- De obicei vom nota vecinul prin adresă (ca în MPI)
- Alternativ, vom preciza direcţia în care se găseşte vecinul
- Pe inel, grilă sau tor: stânga, dreapta, sus, jos, vest, est, nord, sud
- Pentru hipercub: dimensiunea pe care se comunică, un număr între 0 şi $d-1$
- De exemplu, transmiterea liniei i a matricei A către vecinul din dreapta al procesorului ce execută operaţia se scrie

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

Corectitudinea comunicaţiei

- Orice operaţie de transmisie a unui procesor este însoţită de una de recepţie a unui vecin al său
- Primitivele **send** şi **recv** trebuie să apară în perechi, pe ansamblul procesoarelor
- Exemplu: să presupunem că procesorul P_k trebuie să transmită un acelaşi mesaj M vecinilor săi pe un inel, P_{k-1} şi P_{k+1}
- Algoritmul are forma

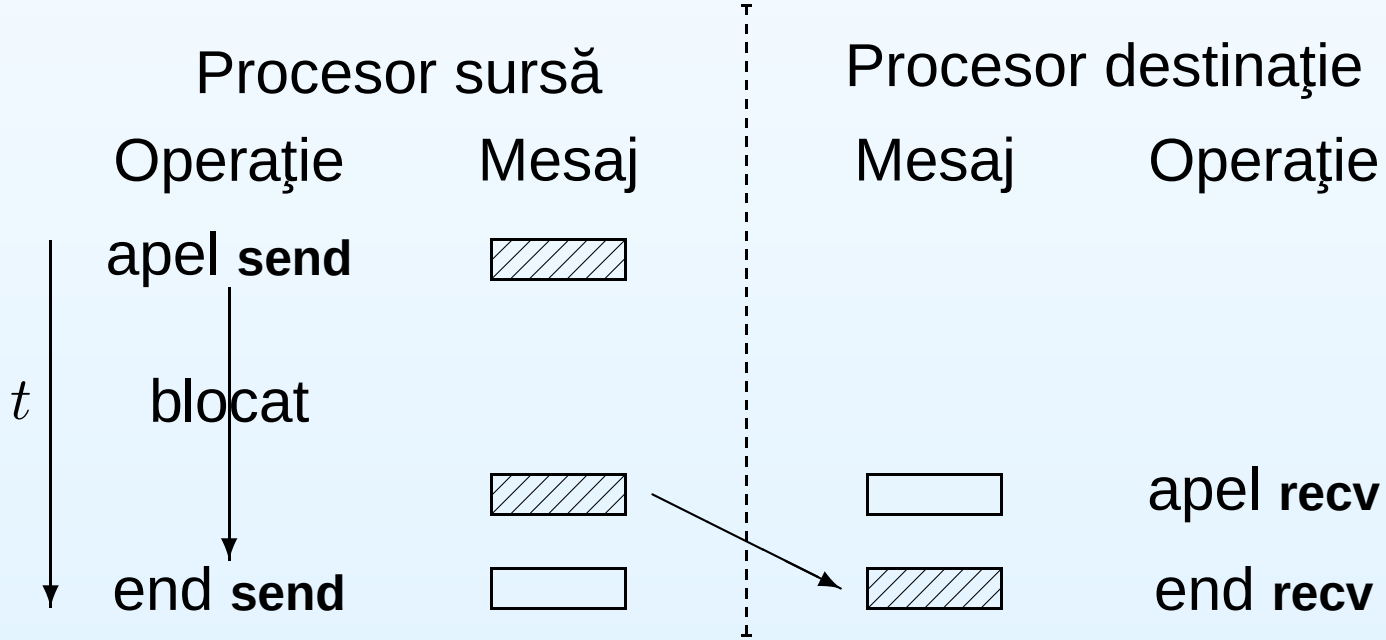
1. **dacă** $\mathrm{id} = k$ **atunci**
    1. **send**(M, dreapta)
    2. **send**(M, stânga)
2. **altfel dacă** $\mathrm{id} = (k-1) \bmod p$ **atunci** **recv**(M, dreapta)
3. **altfel dacă** $\mathrm{id} = (k+1) \bmod p$ **atunci** **recv**(M, stânga)

Terminarea locală a comunicaţiei

- Să presupunem că procesorul P_s transmite un mesaj procesorului P_a
- Momentele în care P_s apelează primitiva **send**, iar P_a **recv**, sunt în general diferite
- Transmiterea efectivă a mesajului se face doar după ce ambele procesoare au apelat primitivele respective
- Se pune întrebarea: ce face procesorul care a apelat primul primitiva sa de comunicaţie, din momentul apelului şi până când celălalt procesor apelează primitiva pereche ?
- Răspunsuri posibile:
  - aşteaptă (fără să facă nimic); este cazul comunicaţiei *blocante*
  - poate executa alte operaţii; comunicaţia este *non-blocantă*

Comunicaţie blocantă

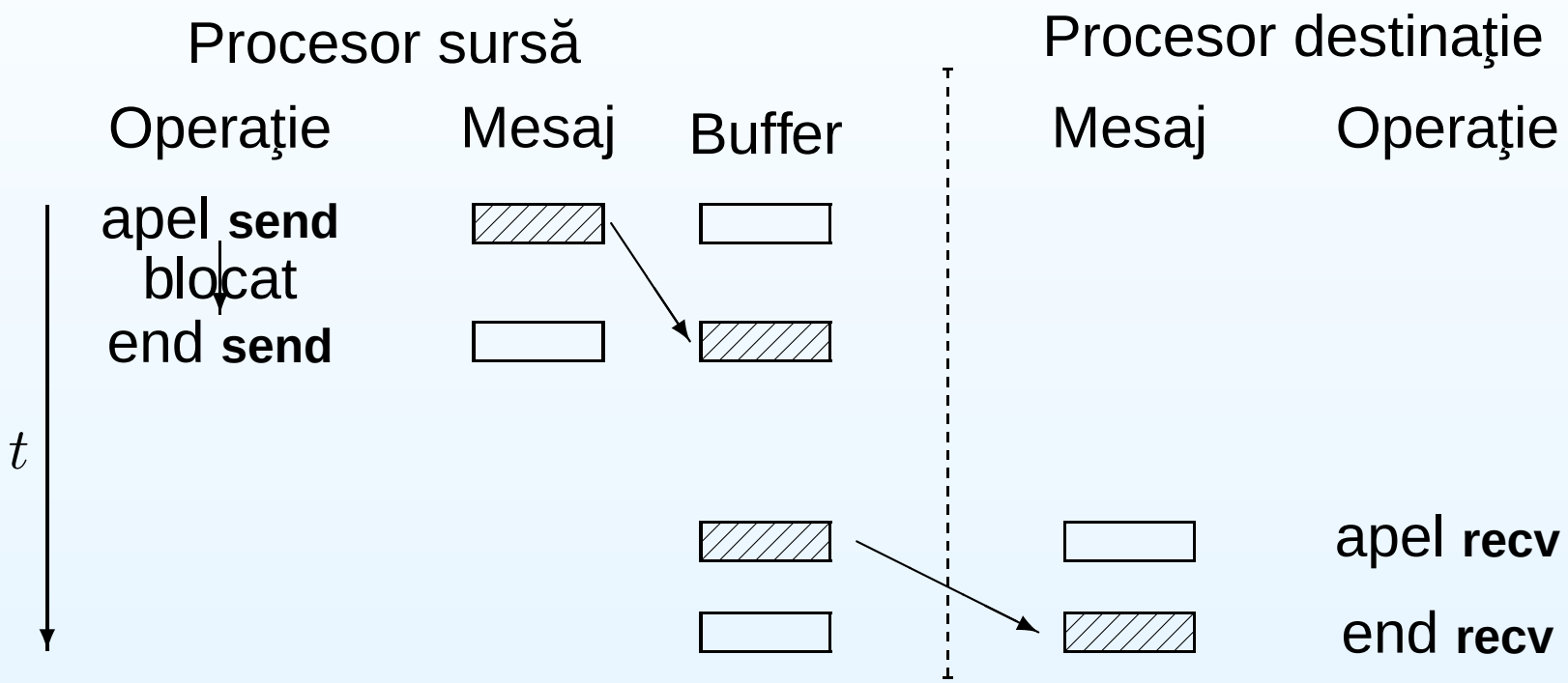
- Al doilea procesor îl aşteaptă pe primul
- Cât aşteaptă, nu face nimic
- Comunicaţia se numeşte blocantă şi sincronă (procesoarele termină simultan)



Comunicaţie blocantă prin buffer

- Comunicaţia blocantă are loc printr-un *buffer*
- Primitiva **send** mută mesajul din zona sa originală într-un buffer; după aceasta, execuţia **send** se termină, zona de memorie a mesajului putând fi refolosită
- Procesorul sursă este blocat doar în timpul în care mesajul este copiat în buffer
- Terminarea rutinei **send** la sursă nu este condiţionată de apelul **recv** la destinaţie
- Atât în varianta sincronă cât şi în cea prin buffer, la terminarea execuţiei primitivei **send**, zona de memorie în care era memorat mesajul poate fi refolosită, iar la terminarea execuţiei **recv**, mesajul este recepţionat în zona de memorie alocată acestui scop

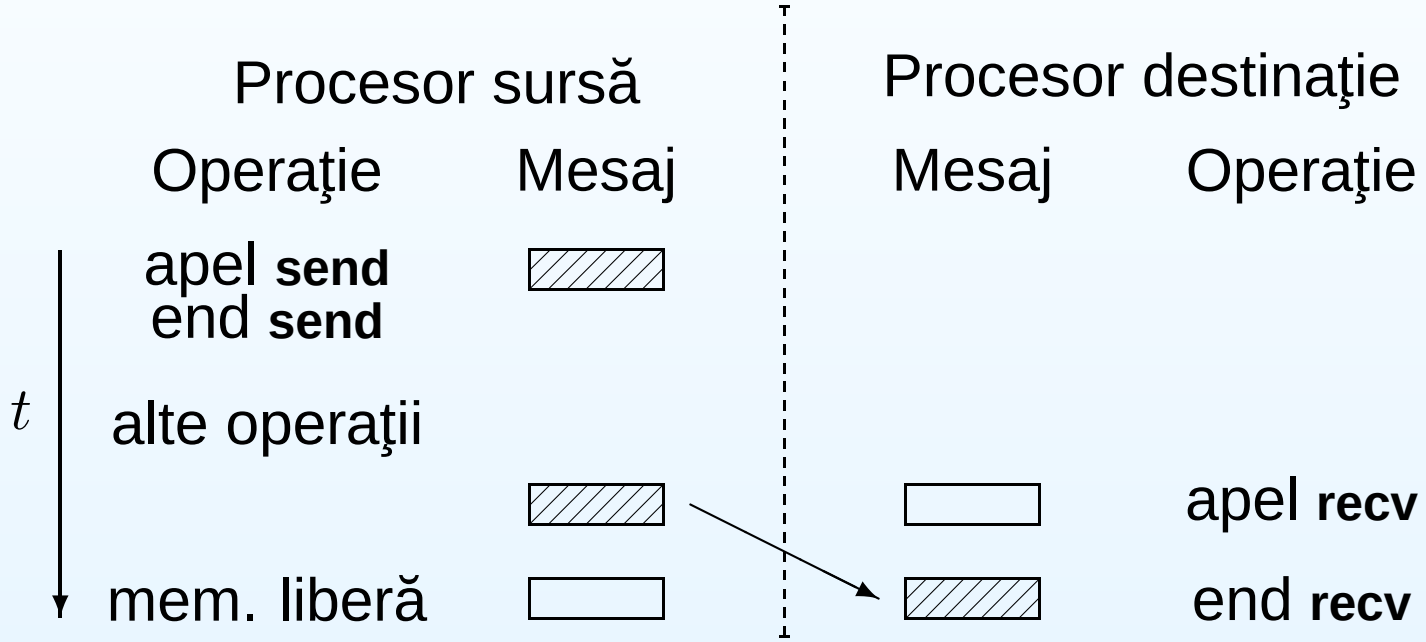
Comunicaţie blocantă prin buffer—ilustraţie



Comunicaţie non-blocantă

- Primitivele **send** şi **recv** se termină *imediat* după apel, fără ca zona de memorie a mesajului să fie liberă la **send** sau să conţină mesajul la **recv**
- Primitivele au doar rolul de a iniţia comunicaţia, transferul efectiv fiind realizat mai târziu la un nivel inferior
- După terminarea execuţiei **send** sau **recv**, procesorul poate executa alte operaţii, în paralel sau concurent cu comunicaţia
- Şi în modul non-blocant comunicaţia se poate desfăşura sincron sau prin buffer

Comunicaţie non-blocantă—ilustraţie



Primitive auxiliare pentru comunicaţia non-blocantă

- Comunicaţia non-blocantă trebuie să fie însoţită de posibilitatea de a afla când se termină comunicaţia efectivă
- În acest scop se introduce primitiva **aşteaptă**, de aşteptare a terminării comunicaţiei
- Utilizare tipică

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

- Dacă **aşteaptă** este executată imediat dupa **send** sau **recv**, efectul este acelaşi ca în cazul primitivelor blocante
- Altă variantă: primitivă care verifică terminarea comunicaţiei

Comparaţii

- Comunicaţia non-blocantă permite în general o mai bună ocupare a procesoarelor, evitând unii timpi morţi
- Erorile de programare în modul blocant se manifestă de obicei prin blocarea programului
- Erorile de programare în modul non-blocant pot fi mai subtile, deoarece se poate altera informaţia transmisă, prin rescrierea zonei de memorie la **send** sau prin utilizarea datelor înainte de a fi recepţionate efectiv
- (In general, depanarea programelor paralele este mult mai dificilă decât în cazul secvenţial)

Eroare tipică în comunicaţia blocantă

- Problemă: fiecare procesor transmite vecinului din dreapta un mesaj, pe un inel de procesoare
- Echivalent, fiecare procesor recepţionează un mesaj de la vecinul din stânga
- Notăm $d1$ şi $d2$ variabilele locale alocate mesajului propriu, respectiv celui recepţionat
- Program incorect în modul blocant sincron:

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

- Fiecare procesor începe prin a transmite şi apoi aşteaptă procesorul din dreapta să execute recepţia
- Acesta însă aşteaptă şi el, etc.
- Toate procesoarele se blochează

Soluţie

- Soluţie 1: transmisia se face prin buffer
- Operaţia **send** se termină local la transferul mesajului în buffer (independent de acţiunile procesorului destinaţie), deci recepţia poate să înceapă
- Soluţie 2: o parte din procesoare încep prin a transmite, celelalte prin a recepţiona, de exemplu

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

Soluţia în cazul non-blocant

- În modul non-blocant, programul unui procesor poate fi

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

- Eventual alte operaţii pot fi intercalate înainte de **aşteaptă**, pentru a folosi timpul posibil mort în care are loc transferul efectiv al mesajului

Standardul MPI

- Message Passing Interface (MPI) este un standard descriind primitive de comunicatie—în contextul unui model de programare SPMD
- Este implementat pe toate calculatoarele MIMD şi pe multe alte arhitecturi (de exemplu, reţele de calculatoare)
- Pentru Windows: Open MPI, MPICH2 şi altele (WMPI)
- Avantaje
  - Portabilitate (acelaşi program MPI) se poate compila şi executa pe multe calculatoare
  - Comunicaţia este implementată eficient pe fiecare calculator
- Se pot testa şi depana programele în medii "ieftine" (pe un desktop sau o mică reţea), rulându-le pe supercalculatoare în momentul în care sunt aproape sigur funcţionale

MPI—generalităţi

- O rutină MPI se execută în cadrul unui grup de procesoare, numit *comunicator*
- Iniţial, există doar comunicatorul `MPI_COMM_WORLD`, conţinând toate procesoarele pe care se execută programul
- În cadrul unui comunicator sunt p procesoare, numerotate de la 0 la $p-1$
- Un procesor poate afla numărul de procesoare din comunicator şi adresa proprie (numită *rang* în MPI) prin

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

MPI—generalităţi

- `MPI_Comm` este tipul predefinit pentru comunicatoare
- `com` este comunicatorul în care se află procesorul
- În variabilele `p` şi `my_id`, rutinele returnează valorile numărului de procesoare, respectiv a adresei procesorului apelant
- Toate rutinele MPI întorc un întreg caracterizând succesul execuţiei

Structura unui program MPI

- Încadrându-se în modelul SPMD, un program MPI are forma obişnuită a unui program secvenţial
- Rutinele `MPI_Init` şi `MPI_Finalize` trebuie apelate înaintea, şi respectiv după, orice alte rutine MPI
- `MPI_Init` primeşte argumentele `argc` şi `argv`, acestea având sau nu o semnificaţie în funcţie de implementarea MPI
- Aceasta este singura constrângere, în rest programatorul este liber sa folosească rutinele MPI potrivite algoritmului pe care îl implementează

Rutine de comunicaţie

- Oricare două procesoare îşi pot transmite mesaje
- Deci topologia virtuală este graful complet conectat
- Principalele rutine de comunicaţie sunt

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

- Prin `MPI_Send` un procesor trimite procesorului `dest` mesajul aflat în memoria locală la adresa `mes`, de lungime `lung` şi având tipul `tipdate`

Rutine de comunicaţie

- MPI are identificatori predefiniţi pentru practic toate tipurile uzuale de date; alte tipuri se pot construi cu rutine speciale
- Mesajului îi este asociată o etichetă (tag), care îl personalizează
- Transmisia mesajului se face în cadrul comunicatorului `com`, care trebuie să conţină şi procesorul sursă, şi cel destinaţie
- La `MPI_Recv` parametri similari
  - se recepţionează un mesaj de lungime cel mult `lung`, trunchiind eventual mesajul transmis
  - tipul de date poate diferi la sursă şi destinaţie
  - după recepţie, variabila `stare` dă informaţii suplimentare despre mesajul primit, ca de exemplu lungimea efectivă a mesajului

Moduri de comunicaţie

- MPI prevede rutine pentru toate tipurile de comunicaţie descrise anterior: blocantă sau non-blocantă, prin buffer sau sincron
- Modul de implementare a rutinelor de bază (standard) `MPI_Send`, `MPI_Recv` nu este precizat
- De exemplu, este posibil ca mesajele scurte să fie transmise prin buffer, iar cele lungi sincron; de aceea programatorul trebuie să-şi ia precauţiile necesare
- Pentru comunicaţia non-blocantă
  - rutina `MPI_Wait` aşteaptă terminarea transmisiei sau recepţiei
  - `MPI_Test` verifică dacă transmisia sau recepţia s-au terminat sau nu

Rutine MPI de comunicaţie

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

Exemple de programe MPI

- Vezi pag.41–43 din carte
- Programele vor fi discutate la calculator

Alte funcţii MPI

- Standardul MPI conţine numeroase alte rutine:
  - Comunicaţie globală: operaţii de comunicaţie implicând toate procesoarele dintr-un grup (comunicator): sincronizare, difuzare, distribuţie, etc.
  - Alte operaţii globale: calculul maximului unor valori distribuite tuturor procesoarelor unui grup, calculul sumei, etc.
  - Topologii virtuale. Pentru creşterea performanţelor unui program MPI pe un anume calculator, programatorul poate descrie o topologie virtuală care să corespundă topologiei reale a calculatorului
  - Procese: se pot crea dinamic, deci se pot îmbina paralelismul şi concurenţa

Operaţii de comunicaţie globală

- Comunicaţii la care participă toate procesoarele (toate sau cele dintr-un grup bine precizat)
  - sincronizare (barieră)
  - difuzare (broadcast, one-to-all)
  - difuzare generală (all-to-all broadcast)
  - distribuţie (difuzare personalizată, scattering)
  - colectare (gathering)
  - schimb complet (multidistribuţie, transpunere)

Sincronizare

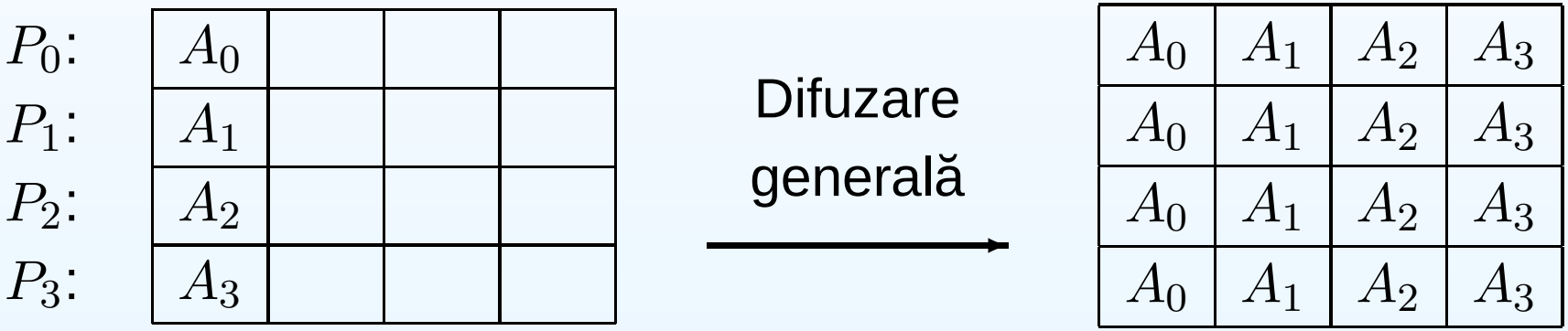
- Nu este o operaţie de comunicaţie propriu-zisă
- Dacă fiecare procesor dintr-un grup apelează primitiva de sincronizare, atunci toate procesoarele aşteaptă până la apelul ultimului dintre ele
- Apoi toate continuă execuţia cu instrucţiunea următoare
- Cu alte cuvinte, procesoarele aşteaptă la barieră până când pot trece toate (aproximativ) simultan
- In MPI funcţia este `MPI_Barrier`

Difuzare

- Un procesor trimite un (singur) mesaj tuturor celorlalte procesoare

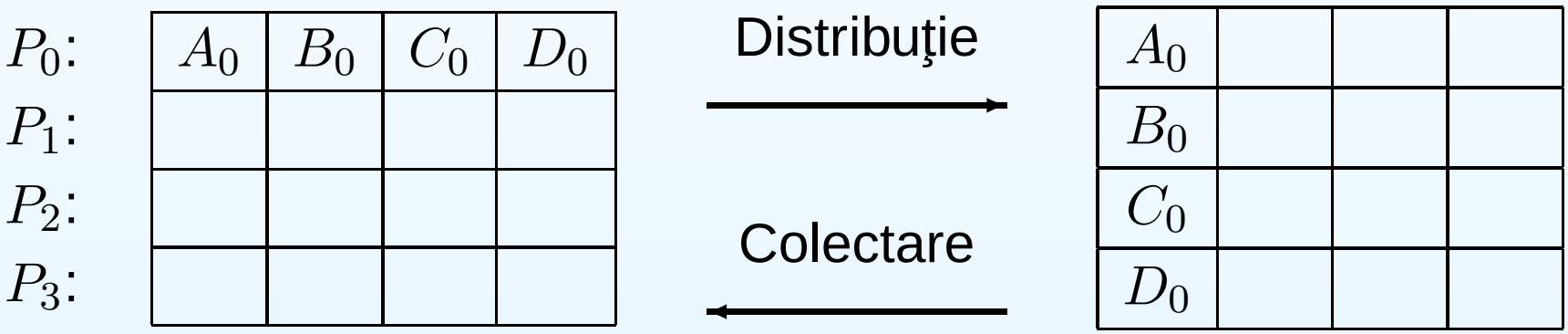
Difuzare generală

- Fiecare procesor trimite un acelaşi mesaj tuturor celorlalte procesoare
- Echivalent, fiecare procesor efectuează o difuzare



Distribuţie şi colectare

- Distribuţie: un procesor trimite câte un mesaj fiecărui alt procesor
- Colectare: operaţia inversă, prin care un procesor primeşte câte un mesaj de la toate celelalte procesoare



Schimb complet

- Fiecare procesor trimite câte un mesaj fiecărui alt procesor
- Echivalent, fiecare procesor efectuează o distribuţie (sau o colectare)
- Dacă fiecare procesor deţine iniţial o linie a unei matrice $p \times p$, în final are o coloană, de unde numele de transpunere

P_0:	A_0	B_0	C_0	D_0
P_1:	A_1	B_1	C_1	D_1
P_2:	A_2	B_2	C_2	D_2
P_3:	A_3	B_3	C_3	D_3

Schimb complet

A_0	A_1	A_2	A_3
B_0	B_1	B_2	B_3
C_0	C_1	C_2	C_3
D_0	D_1	D_2	D_3