Algoritmi paraleli
Curs 2

Vlad Olaru

vlad.olaru@fmi.unibuc.ro

Calculatoare si Tehnologia Informatiei

Universitatea din Bucuresti

Modelul SPMD

- adecvat calculatoarelor MIMD
- vom considera doar cazul MIMD NORMA (memorie distribuita)
- SPMD, Single Program Multiple Data
  - toate CPU executa acelasi program, fiecare opereaza pe propriul set de date
  - executia instructiunilor nu e sincrona (nu e sistem SIMD !), sistem NORMA
  - fiecare CPU executa propriile instructiuni in ritmul propriu

Adresa unui procesor

- program executat pe *p* procesoare
- *p* <= *N*, unde N = nr procesoare calculator paralel
- numerotare: *0, …, p – 1*
- numerotare dinamica, stabilita la momentul lansarii in executie
  - un procesor isi poate afla adresa cu ajutorul unei primitive de biblioteca (sau SO)
- programele SPMD folosesc adresa de procesor ca pe o variabila
  - in particular, folosita pentru a discrimina executia diferitelor portiuni din program

Discriminarea executiei

- fie *id* variabila pt. adresa procesorului
- Ex:

```
1:      id = adresa CPU
2:      if id == 4 then
3:              x = 1
4:      else
5:              x = 0
```

- linia 3 executata doar de catre procesorul P_4
- celelalte *p -1* procesoare executa linia 5
- Obs: variabila *x* e locala fiecarui procesor !

Variabile si indici

- variabilele unui program SPMD sunt *multiplicate*
  - fiecare CPU are propria copie locala, INACCESIBILA altor procesoare !
- regula: procesoarele nu pot modifica variabile din memoria altor procesoare
- interpretare teoretica a exemplului anterior
  - x vector cu p elemente indexat dupa id-ul de procesor
  - programul initializeaza vectorul cu zero, cu exceptia elementului 4
- in programe se folosesc indici locali
- mai rar, pt anumiti algoritmi vom folosi indici globali

Model de comunicatie prin mesaje

- message passing, model pt. calculatoare MIMD NORMA
- fiecare calculator are propria memorie, coordonarea accesului la date se face prin schimb de mesaje
- operatie de baza: procesor sursa P_s trimite mesaj M cu date din memoria locala M_s catre un procesor destinatie P_d care stocheaza datele in memoria sa M_d
- daca nu se specifica altfel, se pp ca sursa si destinatia sunt adiacente (in graful aferent topologiei de interconectare a procesoarelor)
- algoritmii paraleli se vor optimiza pentru topologia folosita

Primitive de baza pt comunicatie

- doua operatii sunt suficiente: *send*, *receive* (*recv*)
  - procesorul sursa transmite mesajul folosind *send*
  - procesorul destinatie receptioneaza mesajul folosind *receive*
- sintaxa

  send(*date*, *destinatie*)

  recv(*date*, *sursa*)
- *date* = variabila locala care stocheaza datele trimise, respectiv primite
- lungimea datelor rezulta din context
- in general, pp ca datele ocupa o zona contigua de memorie

Primitive de baza pt comunicatie (cont)

- sintaxa

  send(*date*, *destinatie*)

  recv(*date*, *sursa*)

- *sursa*/*destinatie* identifica vecinul cu care se comunica
  - uzual vom folosi adresa (id-ul de procesor)
- alternativ, se poate preciza directia in care se afla vecinul
  - inel/lant liniar: stanga/dreapta
  - grid/tor: stanga/dreapta, sus/jos (sau est/vest, nord/sud)
  - hipercub: dimensiunea pe care se comunica (nr intre 0 si d – 1)
  - ex: transmiterea liniei *i* din matricea A catre vecinul din dreapta al procesorului care executa instructiunea

    send(A(i,:), dreapta)

Corectitudinea comunicatiei

- oricarei operatii de transmisie a unui processor trebuie sa-i corespunda o operatie de receptie a unui procesor vecin
  - i.e., **send** & **recv** apar in perechi pe ansamblul procesoarelor
- ex: pp. procesorul P_k trimite acelasi mesaj M vecinilor sai intr-un inel, P_{k-1} si P_{k+1}

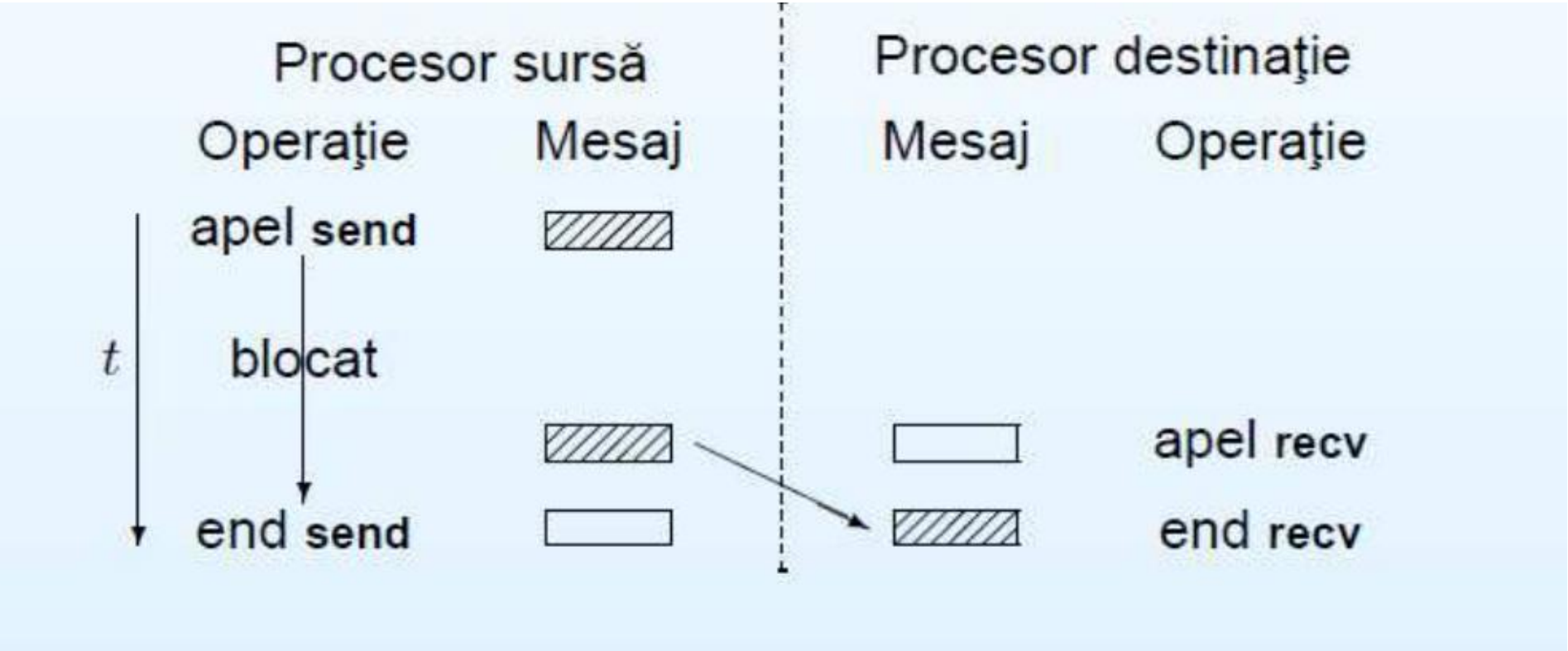
```
1:      if id == k then
2:          send(M, dreapta)
3:          send(M, stanga)
4:      else if id == (k-1) % p then recv(M, dreapta)
5:      else if id == (k+1) % p then recv(M, stanga)
```

Sincronizarea comunicatiei

- pp. P_s trimite un mesaj catre P_d
- in general, momentul transmisiei e diferit de cel al receptiei
- transmisia mesajului devine efectiva doar dupa ce ambele procesoare au executat **send**, respectiv **recv**
- Q: ce face P_s dupa **send** pana cand P_d apeleaza **recv**?
  - doua raspunsuri posibile
  - asteapta (fara sa faca nimic) – *apel sincron, comunicatie blocanta*
  - poate apela alte operatii – *apel asincron, comunicatie ne-blocanta*

Comunicatie blocanta

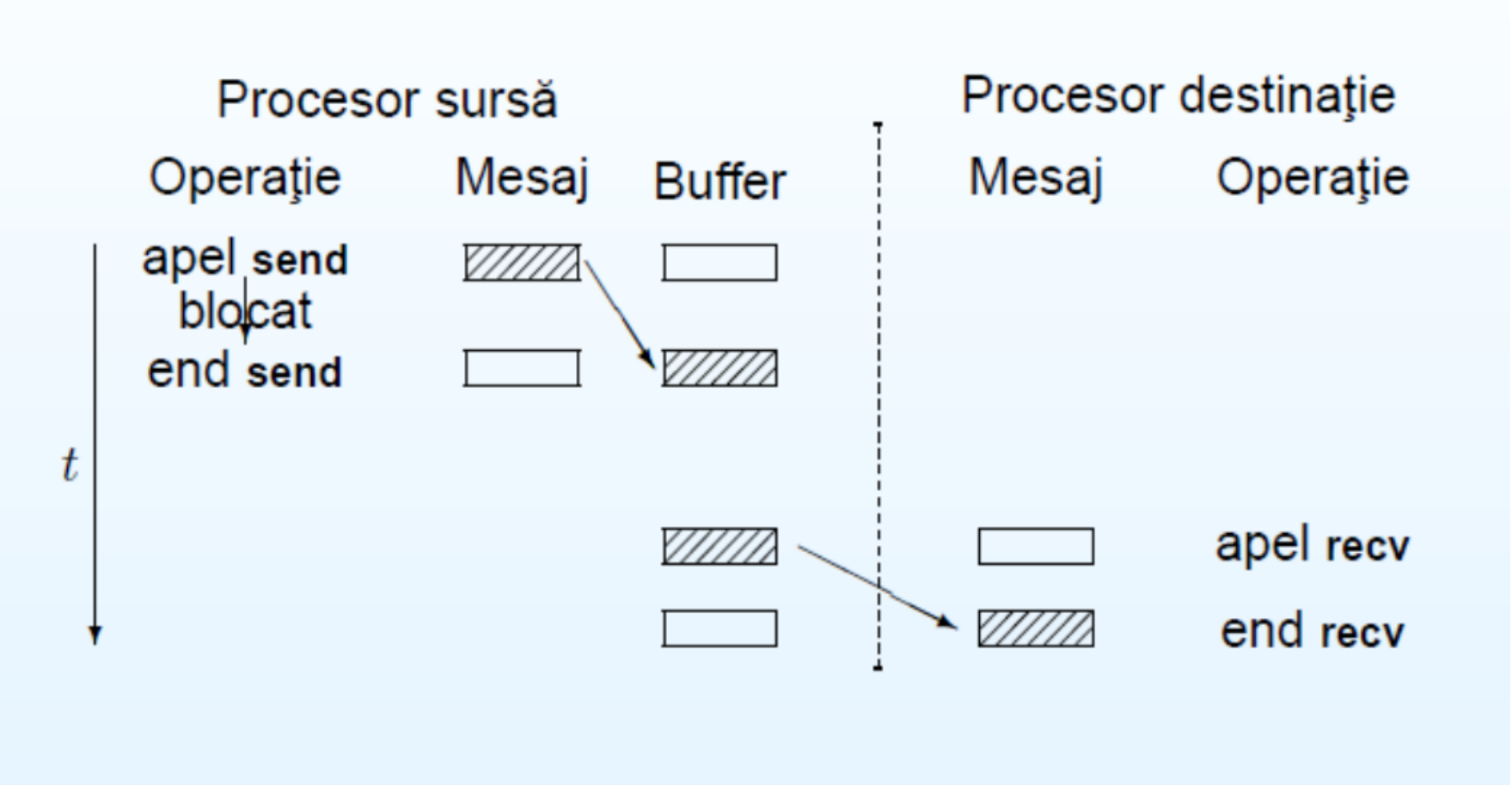
- P_d il asteapta pe P_s fara sa faca nimic
- comunicatie blocanta, sincrona



Comunicatie blocanta prin buffer

- **send** muta mesajul intr-un buffer, apoi termina executia
- CPU sursa blocat doar pe durata copierii mesajului in buffer
- terminarea **send** nu e conditionata de apelul **recv** la destinatie
- la terminarea **send** (fie sincron, fie prin buffer), zona de memorie alocata mesajului se poate refolosi
- la terminarea **recv**, mesajul e receptionat in zona de memoria alocata in acest scop

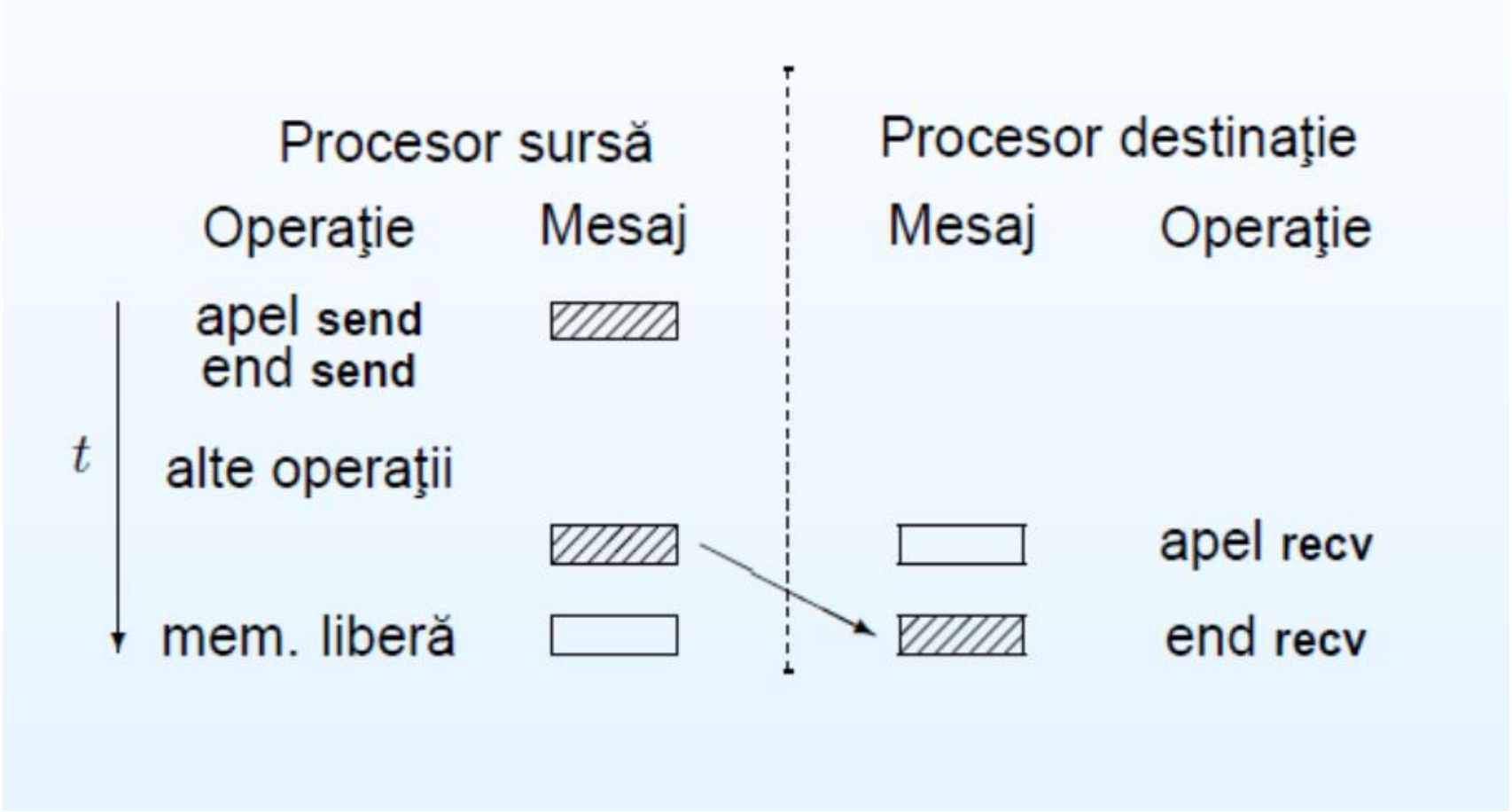
Comunicatie blocanta prin buffer



Comunicatie non-blocanta

- **send/recv** se termina *imediat* dupa apel
  - zona de memorie la **send**, respectiv zona de receptie la **recv** NU sunt libere !
- primitivele de comunicatie doar initiaza comunicatia
- transferul efectiv are loc ulterior, la nivelul sistemului de operare (sau al bibliotecii de comunicatie)
- dupa **send/recv**, procesorul poate executa *imediat* alte operatii, *in paralel sau concurent cu comunicatia* => suprapunerea calculului cu comunicatia (*computation-communication overlap*)
- exista si varianta de comunicatie non-blocanta prin buffer

Comunicatia non-blocanta



Primitive auxiliare pt. comunicatia non-blocanta

- trebuie sa existe posibilitatea de a determina cand s-a efectuat schimbul de mesaje
- primitiva **wait** asteapta terminarea comunicatiei
- scenariu tipic

```
1:    send(date, dest)
2:    executa operatii care nu implica (modifica) date
3:    wait terminare send
4:    modifica date
```

- daca **wait** se executa imediat dupa **send/recv** non-blocante, efectul este acelasi cu executia primitivelor blocante
- varianta: primitiva **test**, verifica ne-blocant terminarea comunicatiei
  - se foloseste intr-un scenariu de *polling*

Comunicatie blocanta vs ne-blocanta

- comunicatia non-blocanta faciliteaza o utilizare mai eficienta a procesoarelor, evitand timpii de asteptare
- erori de programare a comunicatiei blocante pot conduce la blocarea intregului program (*deadlock*)
- erorile de programare a comunicatiei non-blocante sunt mai subtile
  - exista posibilitatea alterarii informatiei transmise
  - eg, reutilizarea memoriei mesajului trimis, respectiv utilizarea bufferului de receptie inainte de primirea efectiva a datelor
- in general, depanarea programelor paralele e mult mai dificila decat a celor secventiale

Eroare tipica in comunicatia blocanta

- problema: fiecare procesor trimite un mesaj vecinului din dreapta pe o topologie inel
  - obs: datorita topologiei inel => fiecare procesor primeste un mesaj de la vecinul din stanga
- fie *date1* si *date2* variabilele locale folosite pentru transmisie, respectiv receptie
- program incorect in varianta blocanta sincrona

send(*date1*, dreapta)

recv(*date2*, stanga)

- fiecare procesor transmite blocant si asteapta ca procesorul din dreapta sa execute receptia, dar acesta la randul lui e blocat in propriul **send**, samd

=> deadlock !

Solutii

- solutia 1:
  - transmisie prin buffer
  - **send** se termina local cand *date1* s-a copiat in bufferul de sistem si poate incepe sa execute receptia pt *date2*
  - terminarea locala a apelului **send** se face independent de procesorul destinatie
- solutia 2:
  - o parte din procesoare incep prin a transmite, celelalte prin a receptiona

```
if id % 2 == 0 then
        send(date1, dreapta)
        recv(date2, stanga)
else
        recv(date2, stanga)
        send(date1, dreapta)
```

Solutia non-blocanta

send(*date1*, dreapta)

recv(*date2*, stanga)

wait terminare **send** si **recv**

- intre comunicatie si operatia de **wait** se pot eventual intercala alte operatii utile

Standardul MPI

- Message Passing Interface (MPI): standard de comunicatie pentru programe SPMD
- implementat atat pe calculatoare MIMD cat si pe retele de calculatoare
- implementat pe diverse sisteme de operare (Linux/Unix, Windows) si in diferite limbaje de programare (C, Fortran, python)
- avantaje
  - portabilitate, acelasi progam MPI si poate compila si executa pe multe calculatoare
  - comunicatie implementata eficient pe fiecare calculator
  - testare/depanare in medii ieftine (eg, desktop-uri + LAN), urmata de rulare pe supercalculatoare dupa declararea gold standard

Generalitati MPI

- un program MPI se executa pe un grup de procesoare numit *comunicator*
- comunicator predefinit: MPI_COMM_WORLD, contine toate procesoarele disponibile programului
- numerotare procesoare comunicator: *0, … , p-1*
- adresa unui procesor dintr-un comunicator s.n. *rang* (*rank*)
- rangul unui procesor se afla cu functia

```
int MPI_Comm_rank(MPI_Comm com, int *id)
```

- numarul de procesoare ale unui comunicator se afla cu functia

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

Generalitati MPI (cont.)

- MPI_Comm, tip predefinit pt comunicatoare
- *com*, comunicatorul in care se afla procesorul care executa codul
- variabilele *p* si *id* contin dupa apel nr de procesoare, respectiv adresa procesorului apelant
- toate rutinele MPI intorc un intreg care caracterizeaza succesul executiei

Structura unui program MPI in C

- model SPMD, *seamana* cu un program secvential
- orice program MPI incepe cu **MPI_Init** (inainte sa apeleze orice alte rutine MPI)
- orice program MPI trebuie sa se termine cu **MPI_Finalize** (dupa acest apel nu se mai pot apela rutine MPI)
- **MPI_Init** primeste argumente *argc* si *argv* (semnificatia depinde de implementarea MPI)
- dincolo de aceasta restrictie, programatorul are libertate totala in folosirea rutinelor MPI cf algoritmului sau

Rutine de comunicatie

- implicit, oricare doua procesoare isi pot trimite mesaje (i.e., topologia virtuala este de graf complet)

```
MPI_Send(void *mesg, int len, MPI_Datatype type,
         int dest, int tag, MPI_comm com);
```

- MPI_Send trimite un mesaj *mesg* de lungime *len* si tip *type* procesorului *dest* din comunicatorul *com*
- MPI are identificatori predefiniti pt tipurile uzuale de date; se pot construi tipuri complexe cu rutine specifice
- mesajul are asociata o eticheta (*tag*), care il personalizeaza
- comunicatorul *com* trebuie sa contina atat procesorul sursa cat si pe cel destinatie

Rutine de comunicatie (cont.)

```
MPI_Recv(void *mesg, int len, MPI_Datatype type, int source,
                                int tag, MPI_comm com, MPI_Status *status);
```

- MPI_Recv receptioneaza un mesaj de lungime maxima *len* de la procesorul *source* din comunicatorul *com* (mesajul poate fi eventual trunchiat)
- tipul de date poate eventual diferi la sursa si destinatie
- MPI_Recv foloseste *tag* pentru a face matching pe mesajele trimise de MPI_Send (MPI_ANY_TAG se poate folosi pentru a primi orice mesaj)
- *status* indica sursa mesajului si eticheta lui + informatie aditionala, gen nr de octeti efectiv primiti

Moduri de comunicatie MPI

- blocant/non-blocant, prin buffer sau sincron
- implementarea implicita a rutinelor MPI_Send/MPI_Recv nu e precizata
  - e posibil ca mesajele scurte sa fie transmise prin buffer iar cele lungi sincron
  - programatorul trebuie sa fie constient de acest aspect
- comunicatie non-blocanta
  - MPI_Wait asteapta terminarea transmisiei sau a receptiei
  - MPI_Test verifica daca transmisia sau receptia s-au terminat

Rutine MPI de comunicatie

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

Alte functii MPI

- comunicatie globala
  - implica toate procesoarele dintr-un comunicator
  - sincronizare, difuzare, distributie, etc
- alte operatii globale
  - calculul maximului unor valori distribuite tuturor procesoarelor dintr-un comunicator
  - calculul sumei, etc
- topologii virtuale: performanta algoritmilor paraleli depinde (si) de topologia HW de interconectare a procesoarelor
  - programatorul poate descrie virtual aceasta topologie
- procese
  - create dinamic, imbina paralelismul HW cu concurenta SW

Operatii de comunicare globala

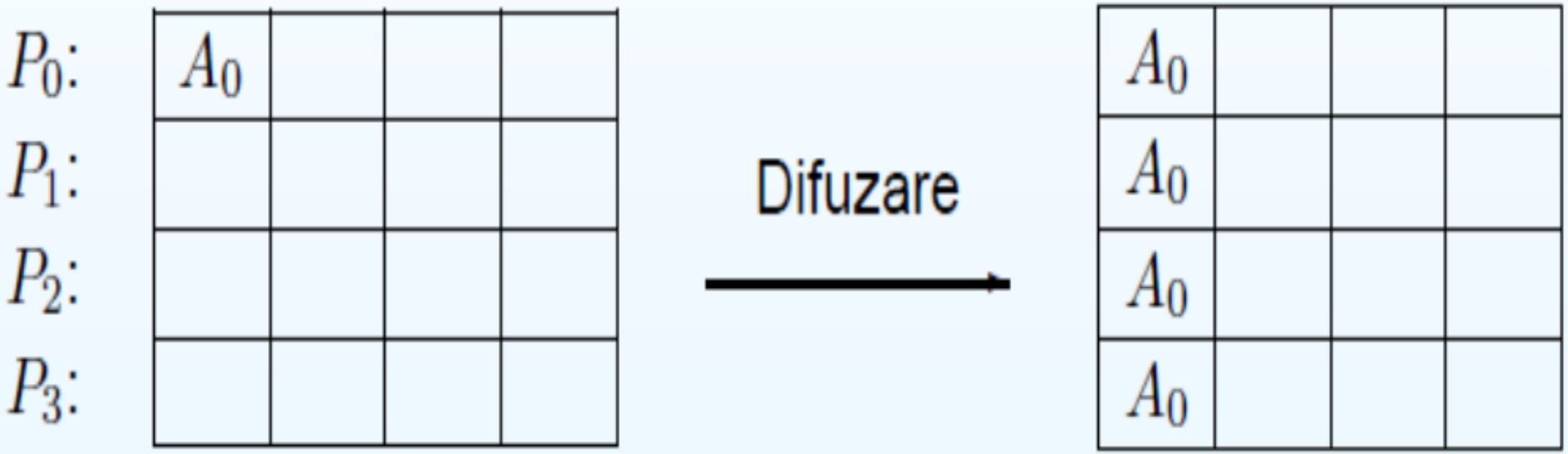
- comunicatii la care participa toate procesoarele dintr-un comunicator
  - sincronizare (bariera)
  - difuzare (broadcast, one-to-all)
  - difuzare generala (all-to-all broadcast)
  - distributie (difuzare personalizata, scattering)
  - colectare (gathering)
  - schimb complet/generalizat (total exchange, multidistributie, transpunere)

Sincronizare

- MPI_Barrier, primitiva tipica pt programele paralele
- forma implicita de comunicare, fara mesaj propriu-zis
- toate procesoarele dintr-un grup care apeleaza primitiva bariera asteapta pana cand ultimul dintre ele ajunge se execute primitiva
  - i.e. toate procesoarele asteapta sa treaca *simultan* de bariera
- dupa bariera, fiecare procesor continua individual cu executia instructiunii urmatoare

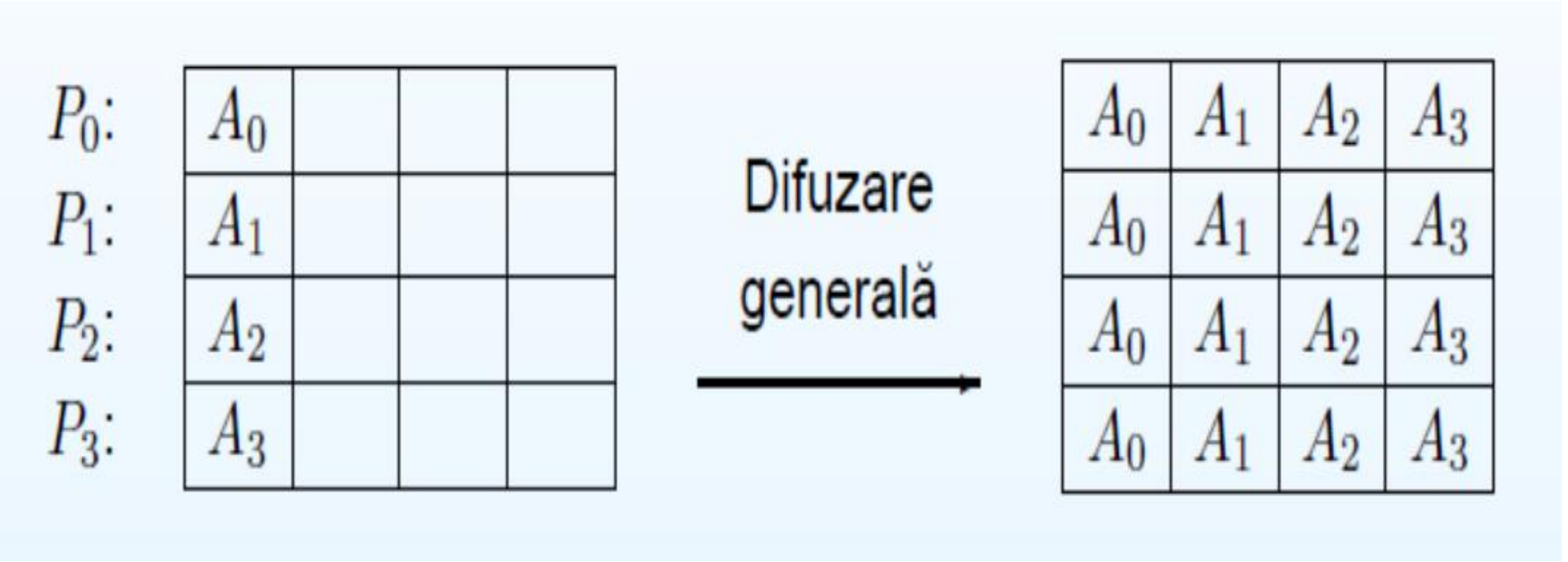
Difuzare

- un procesor trimite un singur mesaj tuturor celorlalte procesoare



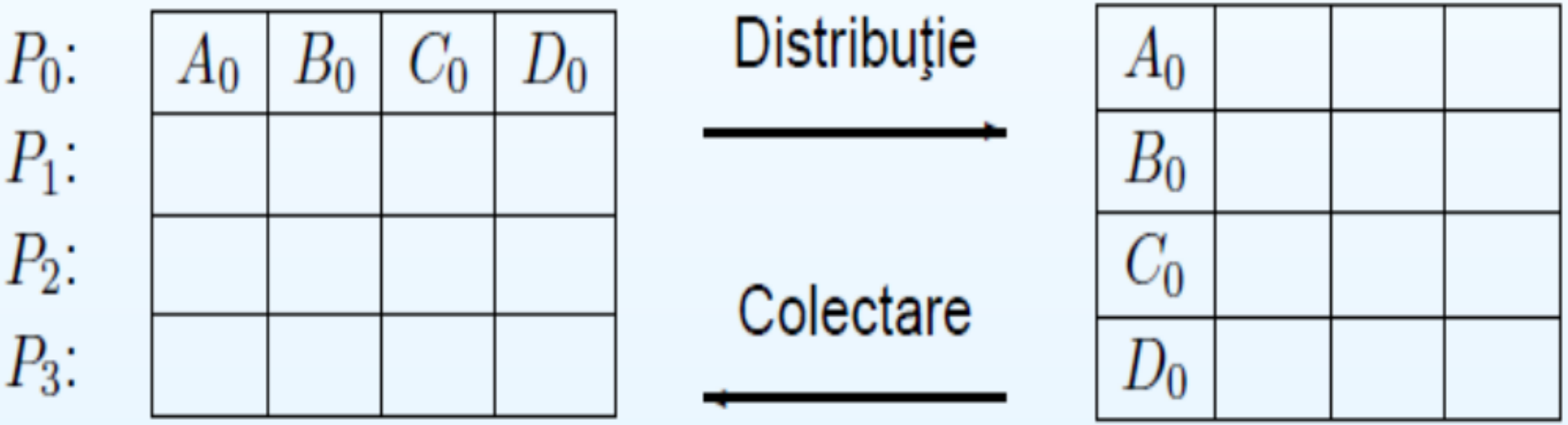
Difuzare generala

- fiecare procesor trimite acelasi mesaj tuturor celorlalte procesoare
- echivalent, fiecare procesor executa o difuzare proprie



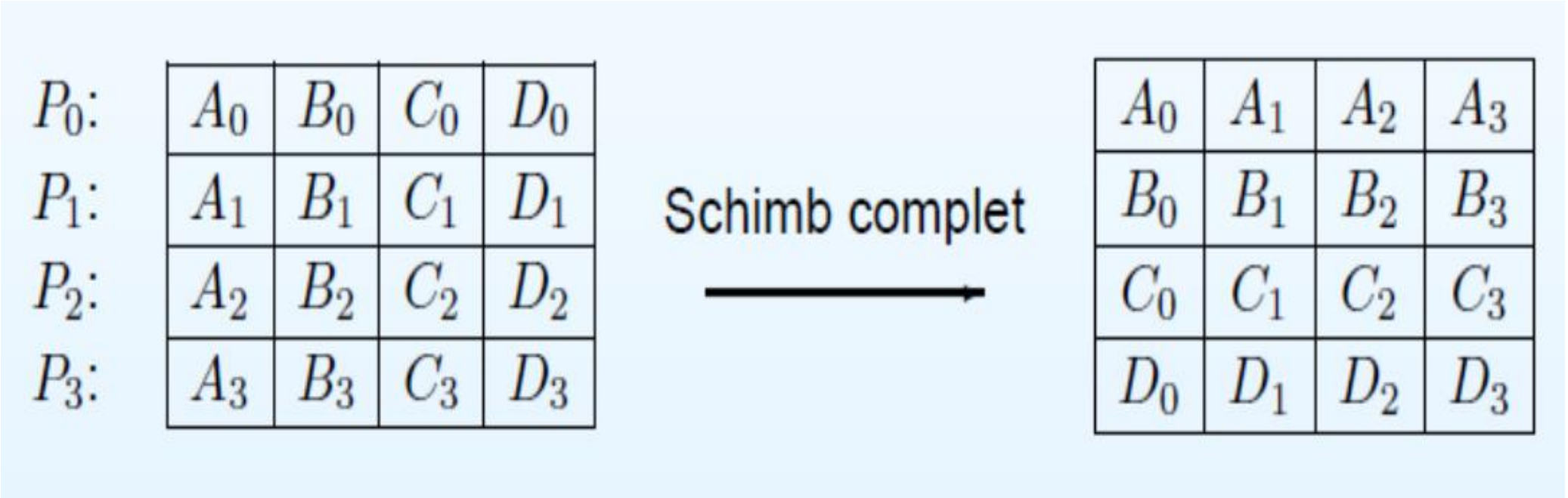
Distributie si colectare

- distributie: un procesor trimite cate un mesaj fiecarui alt procesor
- colectare: operatia inversa, un procesor primeste cate un mesaj de la toate celelalte procesoare



Schimb complet

- fiecare procesor trimite un mesaj tuturor celorlalte procesoare
- echivalent, fiecare procesor executa o distributie (sau o colectare)
- daca un procesor are initial o linie dintr-o matrice $p \times p$, in final primeste o coloana (motiv pt. care operatia se mai numeste si *transpunere*)



Exemplu send/recv

```
 3 #include <mpi.h>
 4 #include <stdio.h>
 5 #include <stdlib.h>
 6
 7
 8 int main(int argc, char** argv) {
 9
10   MPI_Init(NULL, NULL);
11
12 // Find out rank, size
13 int rank,world_size;
14 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
15 MPI_Comm_size(MPI_COMM_WORLD, &world_size);
16
17 int number;
18
19 if (rank == 0)  // procesorul 1 ignora acest bloc de instructiuni
20         {
21             number = 200;
22             MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
23         }
24         else if (rank == 1) // procesorul 0 ignora acest bloc de instructiuni
25         {
26              MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
27              MPI_STATUS_IGNORE);
28              printf("Procesor 1 primeste numarul %d de la procesorul 0\n",  number);
29         }
30
31 MPI_Finalize();
32 }
```

Tipuri de date elementare

MPI datatype	C equivalent
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	char

Detalii MPI_Recv

- mesajul se receptioneaza cand corespund urmatoarele argumente
  - id-ul (rangul) procesorului sursa
  - tag-ul (eticheta)
  - comunicatorul (eg, MPI_COMM_WORLD)
- valori predefinite
  - MPI_ANY_TAG
  - MPI_ANY_SOURCE

Problema

- se dau doua procesoare P_0 si P_1
- implementati o comunicatie alternanta intre cele doua procesoare
  - t_0: P_0 trimite date lui P_1
  - t_1: P_1trimite date lui P_0
  - t_2: P_0 trimite date lui P_1
  - samd

Comunicatie ping-pong

```
12 // Find out rank, size
13 int rank,world_size;
14 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
15 MPI_Comm_size(MPI_COMM_WORLD, &world_size);
16
17 int ping_pong_count = 0;
18 int partner_rank = (rank + 1) % 2;
19
20 while (ping_pong_count < PING_PONG_LIMIT) {
21
22     if (rank == ping_pong_count % 2)
23     {
24
25         // Increment the ping pong count before you send it
26         ping_pong_count++;
27         MPI_Send(&ping_pong_count, 1, MPI_INT, partner_rank, 0, MPI_COMM_WORLD);
28         printf("%d sent and incremented ping_pong_count "
29                "%d to %d\n", rank, ping_pong_count, partner_rank);
30     } else
31         {
32         MPI_Recv(&ping_pong_count, 1, MPI_INT, partner_rank, 0,
33                  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
34         printf("%d received ping_pong_count %d from %d\n",
35                rank, ping_pong_count, partner_rank);
36         }
37      }
38 }
```

Deadlock

```
 8 int main(int argc, char** argv) {
 9
10   MPI_Init(NULL, NULL);
11
12   int world_size,rank,n=0;
13   MPI_Comm_size(MPI_COMM_WORLD, &world_size);
14   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
15   MPI_Status stat;
16
17 // Varianta functionala
18
19   if (rank == 0) {
20         MPI_Send(&n, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
21         MPI_Recv(&n, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &stat);
22         }
23   else { // rank==1
24         MPI_Recv(&n, cnt, MPI_INT, 0, tag, MPI_COMM_WORLD, &stat);
25         MPI_Send(&n, cnt, MPI_INT, 0, tag, MPI_COMM_WORLD);
26         }
27
28 // Varianta deadlock
29
30   if (rank == 0) {
31         MPI_Recv(&n, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &stat);
32         MPI_Send(&n, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
33         }
34   else { // rank==1
35         MPI_Recv(&n, cnt, MPI_INT, 0, tag, MPI_COMM_WORLD, &stat);
36         MPI_Send(&n, cnt, MPI_INT, 0, tag, MPI_COMM_WORLD);
37         }
38   MPI_Finalize();
39 }
```

Comunicatie non-blocanta

- MPI_Isend (buf, count, datatype, dest, tag, comm, request)
- MPI_Irecv(buf, count, datatype, src, tag, comm, request)
- similare cu MPI_Send/MPI_Recv, apare in plus ultimul argument
- *request* este un handle care contine starea transmisiei/receptiei
  - se foloseste impreuna cu MPI_Wait

Ex. comunicatie non-blocanta (1)

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


int main(int argc, char** argv) {

  MPI_Request request;
  MPI_Status status;
  double s_buf[100], r_buf[100];

  MPI_Init(NULL, NULL);

  int world_size,rank,recv_count;
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank==0){
        MPI_Isend(s_buf, 100, MPI_DOUBLE, 1, 10, MPI_COMM_WORLD, &request);
        MPI_Recv(r_buf, 100, MPI_DOUBLE, 1, 22, MPI_COMM_WORLD, &status);
        MPI_Wait(&request, &status);
        }
        else if(rank == 1){
        MPI_Isend(s_buf, 100, MPI_DOUBLE, 0, 22, MPI_COMM_WORLD, &request);
        MPI_Recv(r_buf, 100, MPI_DOUBLE, 0, 10, MPI_COMM_WORLD, &status);
        MPI_Wait(&request, &status);
        }

        MPI_Get_count(&status, MPI_DOUBLE, &recv_count);
        printf("proc %d, source %d, tag %d, count %d\n", rank, status.MPI_SOURCE,
                                                 status.MPI_TAG, recv_count);

  MPI_Finalize();
}
```

Ex. comunicatie non-blocanta (2)

```
13 int rank,world_size;
14 MPI_Request req;
15 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
16 MPI_Comm_size(MPI_COMM_WORLD, &world_size);
17 int *v = calloc(20,sizeof(int));
18 int i;
19
20 if (rank == 0)  // procesorul 1 ignora acest bloc de instructiuni
21         {
22             for (i=0;i<20;i++) v[i]=1;
23             MPI_Isend(v, 20, MPI_INT, 1, 0, MPI_COMM_WORLD,&req);
24         }
25         else if (rank == 1) // procesorul 0 ignora acest bloc de instructiuni
26         {
27              MPI_Irecv(v, 20, MPI_INT, 0, 0, MPI_COMM_WORLD,&req);//req);
28             for (i=0;i<20;i++) printf("%d ",  v[i]);
29         }
30 MPI_Barrier(MPI_COMM_WORLD);
31
32 if (rank == 1)
33         {
34             printf("\n");
35             for (i=0;i<20;i++) printf("%d ",  v[i]);
36             printf("\n");
37         }
38
39 MPI_Finalize();
40 }
```

Ex. comunicatie non-blocanta (3)

```c
#include "mpi.h"
#include <stdio.h>

main(int argc, char *argv[])  {
int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
MPI_Request reqs[4];    // required variable for non-blocking calls
MPI_Status stats[4];    // required variable for Waitall routine

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

// determine left and right neighbors
prev = rank-1;
next = rank+1;
if (rank == 0)  prev = numtasks - 1;
if (rank == (numtasks - 1))  next = 0;

// post non-blocking receives and sends for neighbors
MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

    // do some work while sends/receives progress in background

// wait for all non-blocking operations to complete
MPI_Waitall(4, reqs, stats);

    // continue - do more work

MPI_Finalize();
}
```

Comunicatie pe inel

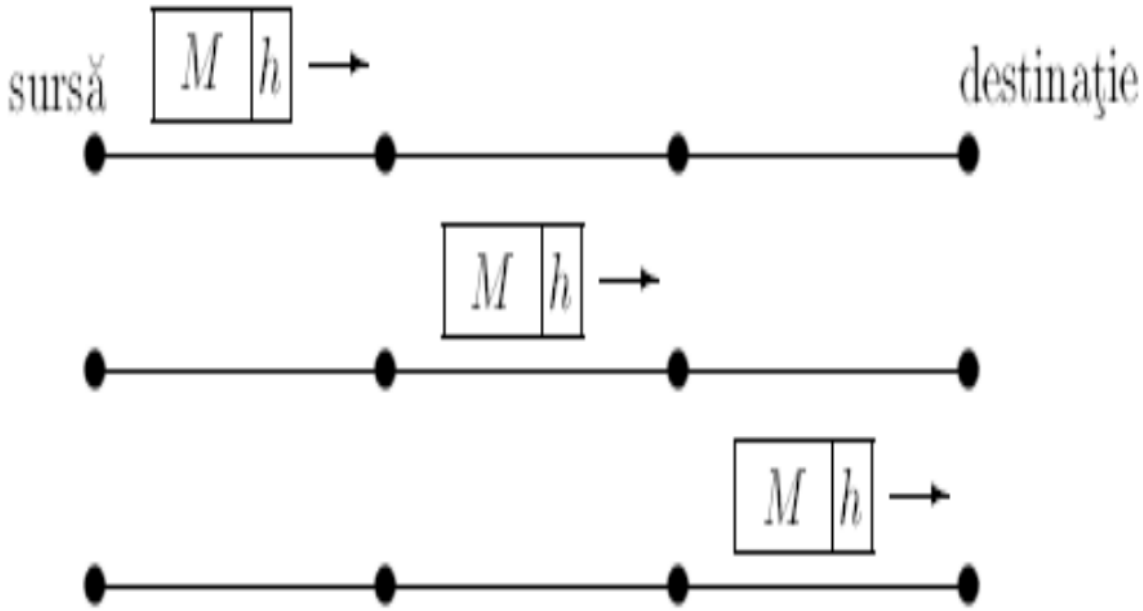
```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>


int main(int argc, char** argv) {

  MPI_Init(NULL, NULL);
  int world_size,rank;
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  int token;
  // Receive from the lower process and send to the higher process. Take care
  // of the special case when you are the first process to prevent deadlock.
  if (rank != 0) {
        MPI_Recv(&token, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Procesorul %d a primit %d de la procesorul %d\n", rank, token, rank - 1);
  } else {
    // Set the token's value if you are process 0
        token = -1;
  }
  MPI_Send(&token, 1, MPI_INT, (rank + 1) % world_size, 0, MPI_COMM_WORLD);
  // Now process 0 can receive from the last process. This makes sure that at
  // least one MPI_Send is initialized before all MPI_Recvs (again, to prevent
  // deadlock)
  if (rank == 0) {
        MPI_Recv(&token, 1, MPI_INT, world_size - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Procesorul %d a primit %d de la procesorul %d\n", rank, token, world_size - 1);
  }

  MPI_Finalize();
}
```

Algoritmi de comunicatie

- comunicatia e parte intrinseca, f. importanta, a algoritmilor paraleli
- modelele de comunicatie
  - specifica modul de transmitere al unui mesaj intre procesoare
  - estimeaza timpul necesar pentru comunicarea unui mesaj
- modele principale
  - store-and-forward
  - comutare de circuite
  - wormhole

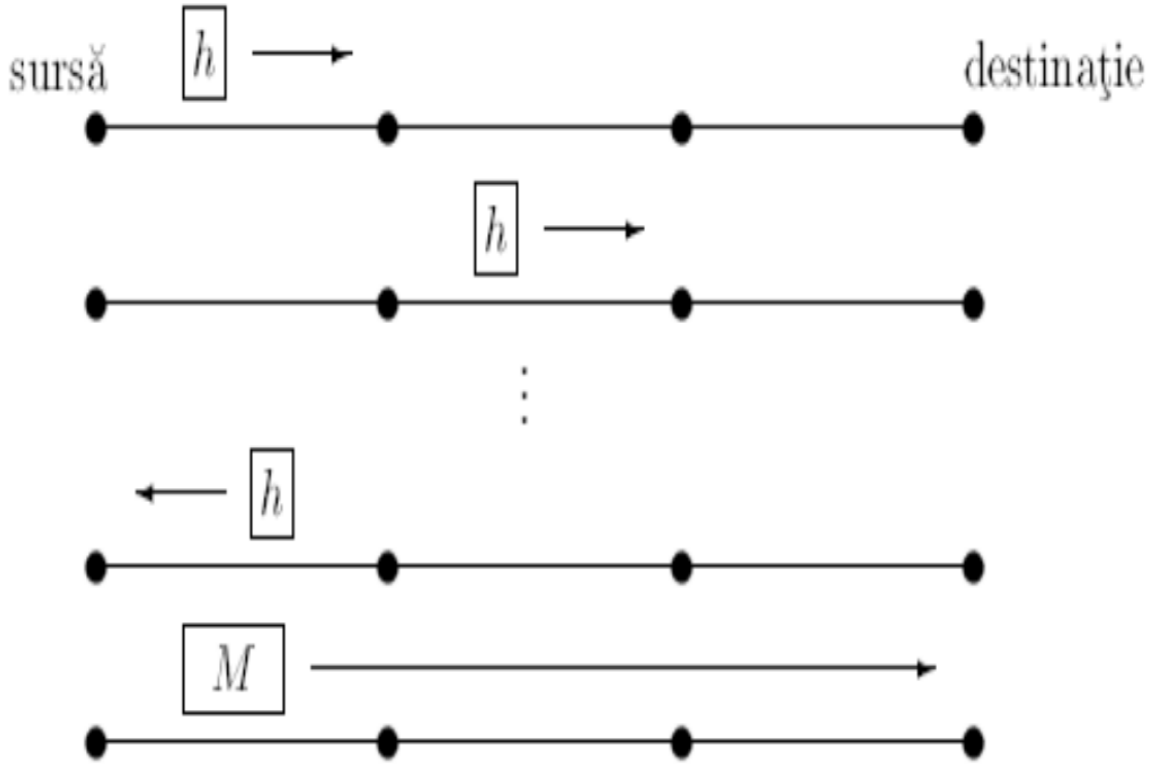
Store-and-forward

- mesajul transmis de sursa e receptat de vecin intr-o zona de memorie (*store*)
- apoi e prelucrat si eventual trimis mai departe catre destinatie (*forward*)
- procesoarele memoreaza integral mesajul



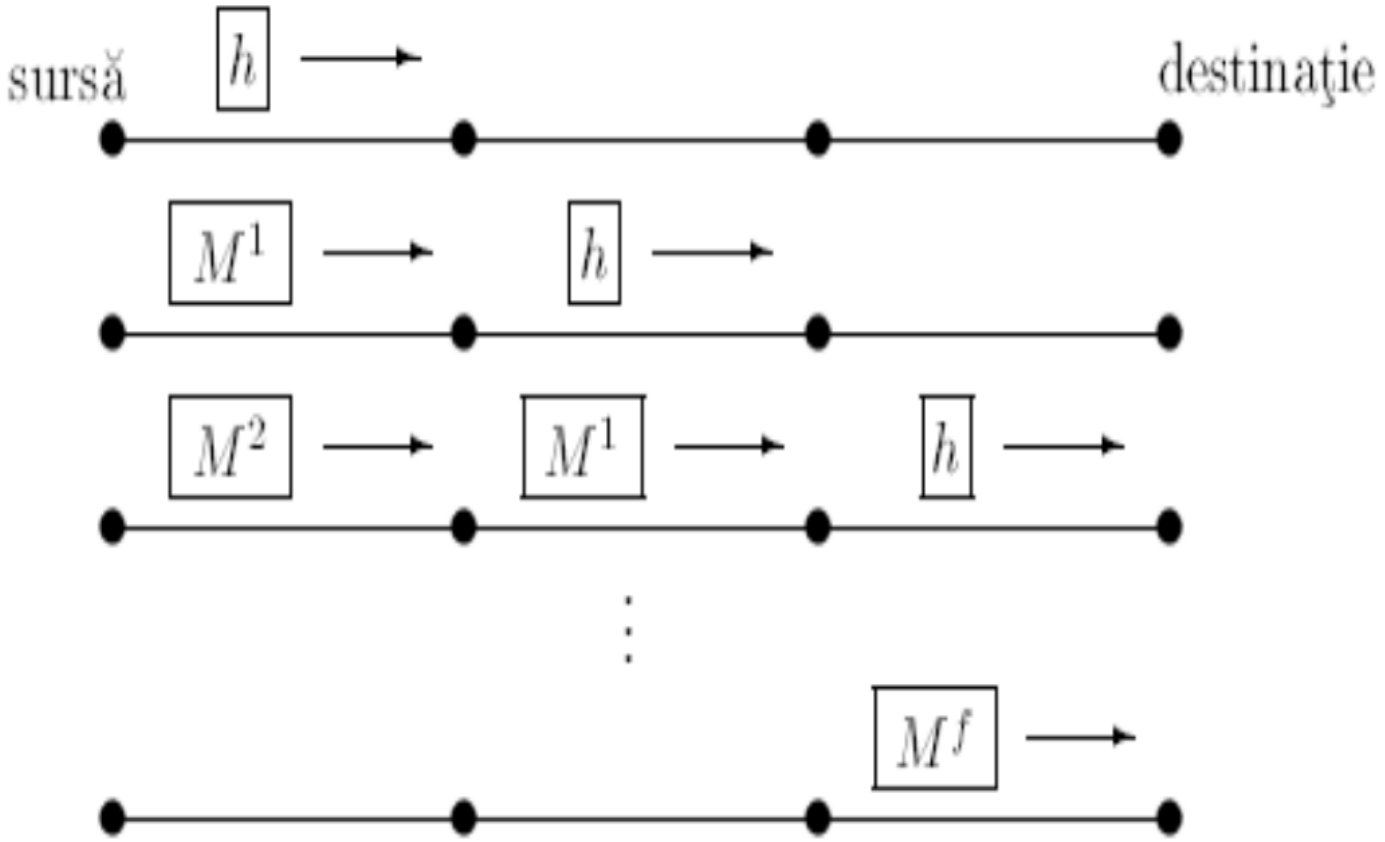
Comutare de circuite

- stabilirea unei cai fizice intre sursa si destinatie: antetul (header-ul) *h* al mesajului *rezerva* legaturile pe intreaga cale
- procesorul sursa e informat de rezervare
- mesajul este trimis direct destinatarului



Modelul wormhole

- mesajul M este impartit in pachete M^i
- dimensiunea pachetului = dimensiunea bufferului canal
- antetul avanseaza si restul pachetelor il urmeaza
- se elimina etapa de confirmare a caii spre destinatie



Modelarea timpului de comunicatie

- consideram modelul store-and-forward si un mesaj de *m* elemente
- modele posibile
  - durata de transmisie a unui mesaj este constanta
  - model proportional

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

β = durata transmisiei unui element

  - model liniar

$$t_c(m) = \sigma + m\beta$$

σ = durata start-up-ului transmisiei

Operatii de comunicatie globala

- One-to-One: transmisie mesaj de la un procesor la altul (nu neaparat vecini)
- One-to-All: difuzare, transmiterea unui mesaj de la un procesor la toate celelalte
  - MPI_Bcast, MPI_Scatter
- All-to-All: difuzare generala, transmiterea unui mesaj de la fiecare procesor catre toate celelalte
  - MPI_Alltoall
- difuzare personalizata (scatter): transmiterea de la un procesor a cate unui mesaj pentru fiecare dintre celelalte procesoare
  - MPI_Scatter
- colectare (gather): operatie inversa, MPI_Gather