

MPI - Message Passing Interface

1. Generalități

MPI este un **standard** pentru comunicarea prin mesaje, elaborat de MPIForum. În definirea lui au fost utilizate caracteristicile cele mai importante ale unor sisteme anterioare, bazate pe comunicația de mesaje. A fost valorificată experiența de la IBM, Intel (NX/2) Express, nCUBE (Vertex), PARMACS, Zipcode, Chimp, PVM, Chameleon, PICL.

MPI **are la bază** modelul proceselor comunicante prin mesaje: un calcul este o colecție de procese secvențiale care cooperează prin comunicare de mesaje.

MPI **este o bibliotecă** nu un limbaj. El specifică convenții de apel pentru mai multe limbaje de programare: C, C++, FORTRAN.77, FORTRAN90.

MPI a ajuns la versiunea 2, trecând succesiv prin versiunile:

- MPI 1 (1993), un prim document, orientat pe comunicarea punct la punct.
- MPI 1.0 (iunie 1994) este versiunea finală, adoptată ca standard; include comunicarea punct la punct și comunicarea colectivă.
- MPI 1.1 (1995) conține corecții și extensii ale documentului inițial din 1994, modificările fiind univoce.
- MPI 1.2 (1997) conține extensii și clarificări pentru MPI 1.1
- MPI 2 (1997) include funcționalități noi:
 - Procese dinamice
 - Comunicarea “one-sided”
 - Operații de I/E paralele

Obiectivele MPI:

- Proiectarea unei interfețe de programare a aplicațiilor
- Comunicare eficientă
- Să fie utilizabil în medii eterogene
- Portabilitate
- Interfața de comunicare să fie sigură (erorile tratate dedesubt)
- Aproximare de practici curente (PVM, NX, Express, p4)
- Semantica interfeței să fie independentă de limbaj

2. Comunicarea punct la punct

2.1. Operații de bază

Operațiile de bază pentru comunicarea punct la punct sunt **send** și **receive**. În MPI ele apar ca extensii necesare ale operațiilor similare din alte biblioteci de comunicare prin mesaje. Justificăm în cele ce urmează utilitatea acestor extensii.

Considerăm mai întâi operația de transmitere de mesaje, în forma “uzuală”:

send (adresă, lungime, destinație, tip)

unde

- **adresa** identifică începutul unei zone de memorie unde se află mesajul de transmis
- **lungime** este lungimea în octeți a mesajului
- **destinație** este identificatorul procesului căruia i se trimite mesajul (uzual un număr întreg)
- **tip (flag)** este un întreg ne-negativ care restricționează recepția mesajului. Acest argument permite programatorului să rearanjeze mesajele în ordine, chiar dacă ele nu sosesc în secvența dorită.

Acest set de parametri este un bun compromis între ceea ce utilizatorul dorește și ceea ce sistemul poate să ofere: transferul *eficient* al unei zone contigue de memorie de la un proces la altul. În particular, sistemul oferă mecanismele de păstrare în coadă a mesajelor, astfel că o operație de recepție

recv (adresă, maxlung, sursă, tip, actlung)

se execută cu succes doar dacă mesajul are tipul corect. Mesajele care nu corespund așteaptă în coadă.

În cele mai multe sisteme, **sursa** este un argument de ieșire, care indică originea mesajului. În alte cazuri este folosit pentru a restricționa recepția. Ceilalți parametri reprezintă, respectiv:

- adresă, maxlung – descrierea zonei receptoare de mesaj
- actlung – numărul de octeți efectiv recepționați

Formele prezentate impun restricții considerate actualmente inacceptabile. Analizăm pe rând aspectele.

2.1.1. Descrierea zonelor tampon

Perechea (adresă, lungime) este inadecvată pentru că:

1. În multe cazuri mesajul nu ocupă o zonă contiguă. Este dezirabilă o formă care descrie distribuția originală a datelor.
2. Mesajele conțin valori cu tip; într-un mediu eterogen, transmiterea lor ca simple șiruri de octeți este inadecvată.

Transmiterea ca date cu tip ar putea fi ușurată de rutine adecvate de conversie, făcând parte din biblioteca de comunicare. În MPI, tamponul de comunicație este definit de tripleta

(adresă, contor, tip_de_date)

care descrie **contor** valori de tipul **tip_de_date** situate începând cu adresa **adresă**. Astfel, (A, 300, MPI_DOUBLE) reprezintă un vector A de 300 valori reale, în C. O implementare MPI va asigura că *aceleași* 300 de valori sunt recepționate, chiar dacă reprezentarea lor în mașina destinatarului diferă de reprezentarea pe mașina sursă.

În plus, un utilizator poate construi (folosind rutinele de bibliotecă) propriile sale tipuri de date, care se pot referi la date din zone ne-contigue de memorie.

2.1.2. Contextul

Fiecare comunicare de mesaj se derulează într-un anumit context.. Mesajele sunt întotdeauna primite în contextul în care au fost transmise. Mesajele transmise în contexte diferite nu interferă.

Contextele sunt alocate de sistem (în timpul execuției, ca răspuns la cererea utilizatorilor) și sunt folosite, ca și câmpurile de tip (**tag**) pentru restricționarea recepției mesajelor.

2.1.3. Grupurile de procese

Contextul este **partajat** de un grup de procese. Procesele unui grup sunt ordonate și fiecare proces este identificat prin numărul său de ordine din grup. Pentru un grup de **n** procese, numerele valide au valori de la **0** la **n-1**.

2.1.4. Comunicator

Informațiile de **context** și de **grup** sunt asamblate într-un parametru suplimentar al operațiilor de comunicare, denumit **comunicator**. Mesajele poartă, în afara datelor, informații care permit diferențierea lor și recepția lor selectivă. Aceste informații sunt:

- Sursă
- Destinatar
- Tag
- Comunicator

și constituie **plicul** (anvelopa) mesajului. MPI prevede un comunicator predefinit, `MPI_COMM_WORLD`, care permite comunicarea cu orice proces accesibil după inițializarea MPI și asocierea unui număr de ordine (rank) fiecărui proces. Rolul și modalitatea de a defini noi comunicatori vor fi discutate mai târziu.

Cu aceste elemente, sintaxa operațiilor **send** și **receive** *blocante* este următoarea:

`MPI_SEND (buf, count, datatype, dest, tag, comm)`

unde parametrii sunt:

- IN buf, adresa inițială a tamponului sursă
- IN count, numărul de elemente (întreg ne-negativ)
- IN datatype, tipul fiecărui element
- IN dest, numărul de ordine al destinatarului (întreg)
- IN tag, tipul mesajului (întreg)
- IN comm, comunicatorul implicat

`MPI_RECV (buf, count, datatype, source, tag, comm, status)`

unde parametrii sunt:

- OUT buf, adresa inițială a tamponului destinatar
- IN count, numărul de elemente din tampon (întreg ne-negativ)
- IN datatype, tipul fiecărui element
- IN source, numărul de ordine al sursei (întreg)
- IN tag, tipul mesajului (întreg)
- IN comm, comunicatorul implicat
- OUT status, starea, element (structură) ce indică sursa, tipul și contorul mesajului efectiv primit

În limbajul C aceste funcții au următoarele prototipuri, valorile returnate reprezentând coduri de eroare:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status);
```

Operația **send** este blocantă. Ea nu redă controlul apelantului până când mesajul nu a fost preluat din tamponul sursă, acesta din urmă putând fi reutilizat de transmițător. Mesajul poate fi copiat direct în tamponul destinatar (dacă se execută o operație **recv**) sau poate fi salvat într-un tampon temporar al

sistemului. Memorarea temporară a mesajului *decuplează* operațiile **send** și **recv**, dar este consumatoare de resurse. Alegerea unei variante aparține implementatorului MPI.

și operația **recv** este blocantă: controlul este redat programului apelant doar când mesajul a fost recepționat. Există și alte operații MPI, care permit programatorului să controleze modul de comunicare. Vom reveni asupra acestui subiect.

2.2. Un exemplu complet

Utilizarea funcțiilor **send** și **recv** este ilustrată în următorul exemplu:

```
# include "mpi.h"
main (int argc, char **argv)
{ char message[40];
  int myrank;
  MPI_Status status;
  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
  if (myrank==0)
    { strcpy (message, "Hello, there");
      MPI_Send (message, strlen(message), MPI_CHAR, 1, 99,
                MPI_COMM_WORLD);
    }
  else { MPI_Recv (message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD,
                  &status);
        printf ("tReceived: %s \n", message);
      }
  MPI_Finalize();
}
```

În acest exemplu, procesul 0 (myrank==0) trimite un mesaj procesului 1 folosind operația **send**. Procesul 1 (myrank!=0) primește acest mesaj prin operația **recv** și îl afișează. Descrierile celor două procese coincid, iar diferențierea între acțiunile lor se face prin selecția după numărul de ordine al procesului.

2.3. Mai multe despre recepție

Recepția unui mesaj este guvernată de valorile din anvelopa sa. Un mesaj este primit de destinatarul său dacă anvelopa conține valorile specificate de operația **MPI_Recv** pentru argumentele **source**, **tag** și **comm**. Receptorul poate specifica **MPI_ANY_SOURCE**, respectiv **MPI_ANY_TAG** pentru a accepta mesaje de la orice sursă și/sau cu orice tag. El nu poate specifica valori oarecare pentru comunicatorul **comm** !! Deci mesajul nu poate fi primit decât de receptorul adresat și pentru un același comunicator

În cazul folosirii unor valori ANY pentru sursă și tag, valorile corespunzătoare unui mesaj recepționat pot fi aflate din parametrul **status**. În limbajul C, status este o structură cu trei câmpuri, **MPI_SOURCE**, **MPI_TAG**, **MPI_ERROR**, care poate conține însă și câmpuri adiționale

Lungimea mesajului recepționat poate fi aflată prin apelul operației **MPI_Get_count**.

2.4. Corespondența tipurilor și conversiile datelor

Correspondența tipurilor datelor se referă două aspecte:

- Corespondența tipurilor limbajului gazdă (C) cu tipurile specificate în operațiile de comunicare;
- Corespondența tipurilor la transmițător și receptor.

În plus, folosirea standardului în medii eterogene necesită **conversii** de tip. Aceste aspecte nu sunt tratate în lucrarea de față.

2.5. Alte operații MPI

Prezentarea exemplului anterior ne oferă prilejul să comentăm și alte operații MPI. Fiecare program MPI trebuie să conțină un apel al operației **MPI_Init**. Rolul acesteia este de a inițializa “mediul” în care programul va fi executat. Ea trebuie executată *o singură dată, înaintea* altor operații MPI. Pentru a evita execuția sa de mai multe ori (și deci provocarea unei erori), MPI oferă posibilitatea verificării dacă MPI_Init a fost sau nu executată. Acest lucru este făcut prin

MPI_INITIALIZED (flag)

- OUT flag, este **true** dacă MPI_Init a fost deja executată

În limbajul C funcția are următorul prototip:

```
int MPI_Initialized(int *flag);
```

Aceasta este, de altfel, singura operație ce poate fi executată înainte de MPI_Init.

Revenind la MPI_Init, forma sa generală este

MPI_INIT()

iar în limbajul C prototipul este

```
int MPI_Init (int *argc, char ***argv);
```

Funcția C de inițializare acceptă ca argumente **argc** și **argv**, argumente ale funcției **main**, a căror utilizare nu este fixată de standard și depinde de implementare.

Perechea funcției de inițializare este **MPI_Finalize**, care trebuie executată de fiecare proces, pentru a “închide” mediul MPI. Forma acestei operații este următoarea:

MPI_FINALIZE()

iar în limbajul C prototipul este

```
int MPI_Finalize (void);
```

Utilizatorul trebuie să se asigure că toate comunicațiile în curs de desfășurare s-au terminat înainte de apelul operației de finalizare. După MPI_Finalize, nici o altă operație MPI nu mai poate fi executată (nici măcar una de inițializare).

Un proces poate afla poziția sa în grupul asociat unui comunicator prin apelul operației

MPI_COMM_RANK (comm, rank)

- IN comm, este comunicatorul implicat
- OUT rank, este rangul procesului apelant

În limbajul C funcția are următorul prototip:

```
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

Procese din exemplul dat folosesc răspunsul acestei operații în decizia care urmează în program.

2.6. Alte moduri de comunicare

Modul standard de comunicare în MPI este blocant: controlul nu este redat apelantului până când mesajul nu a fost preluat din tamponul de emisie (într-un tampon intermediar **sau** în tamponul receptorului), astfel încât procesul transmițător poate să modifice din nou tamponul de emisie.

Memorarea temporară a mesajelor, în cazul unor operații blocante, nu este întotdeauna operațională. Dacă mesajele transmise sunt foarte lungi și sistemul nu dispune de suficientă memorie temporară, ele

trebuie transferate direct din tamponul sursă în tamponul destinatar. Aceasta presupune, însă, că transmitătorul este blocat până la execuția unei recepții corespundente. În cazul unei combinații de procese care-și transmit mesaje în lanț, mecanismul conduce la degradarea performanțelor, așa cum rezultă și din exemplul prezentat în figura 1:

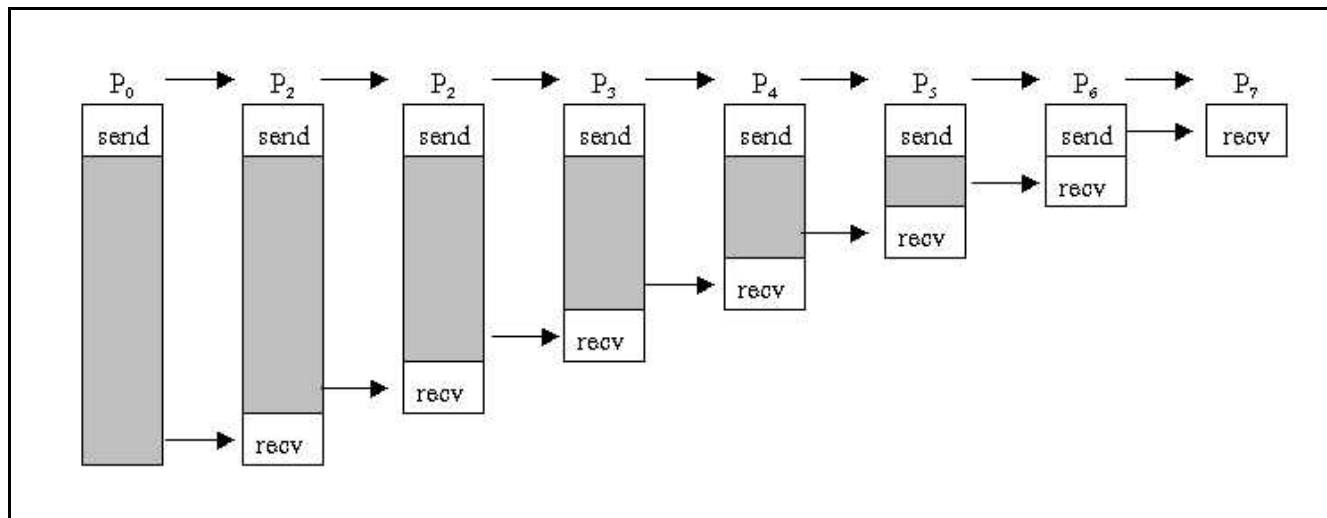


Figura 1

Fiecare proces, cu excepția primului și a ultimului, conține o pereche de operații send-recv: prima transmite un mesaj către următorul proces din secvență, a doua recepționează un mesaj de la precedentul proces. Transmiterea directă a datelor între tamponurile de emisie și recepție secvențializează execuțiile operațiilor de comunicare, procesele rămânând în așteptare (zonele colorate din figura 1) perioade lungi de timp. Există mai multe soluții pentru a controla execuția operațiilor de comunicare astfel încât programul să nu depindă de cantitatea de memorie tampon oferită de sistem.

2.6.1. Ordonarea operațiilor send-recv

O soluție este împerecherea operațiilor, de exemplu, procesele de rang par execută operațiile în ordinea send-recv, în timp ce procesele de rang impar în ordine inversă, recv-send.

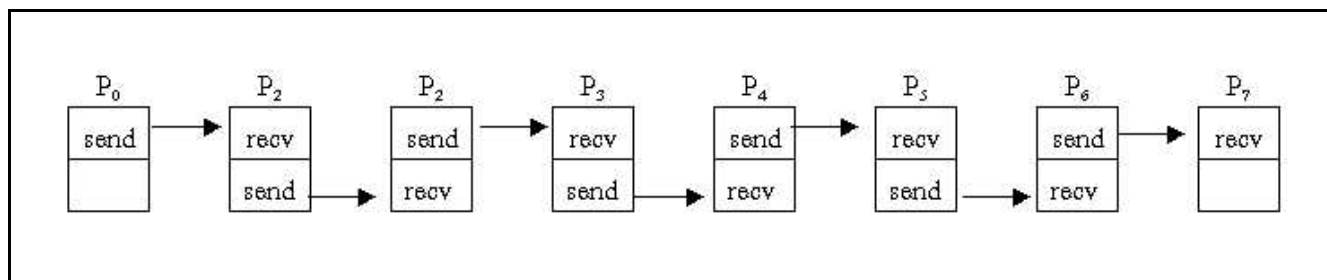


Figura 2

2.6.2. Operații send-recv combinate

Schema precedentă este greu de implementat în cazul unei combinații mai complicate de procese. O alternativă este utilizarea operației **MPI_Sendrecv**, care combină într-un singur apel transmiterea unui mesaj către o destinație cu recepția unui mesaj de la un alt proces. Subsistemul de comunicare se ocupă de combinarea operațiilor astfel încât să se evite blocarea definitivă a proceselor.

Ambele operații, send și recv, folosesc același comunicator, dar eventual diferite tag-uri. Tamponurile de emisie și de recepție trebuie să fie distincte.

Operația are și o variantă, **MPI_Sendrecv_replace**, în care același tampon este folosit atât pentru emisie cât și pentru recepție, astfel că mesajul transmis este înlocuit cu mesajul recepționat.

2.6.3. Transmitere prin tampon

MPI permite programatorului să prevadă un tampon în care datele sunt plasate până la livrarea lor. Operați **MPI_Bsend** se termină, eventual înainte de demararea unei recepții corespondente, odată cu plasarea datelor în zona tampon. Programatorul poate rezerva această zonă prin operația **MPI_Buffer_attach**, în care specifică dimensiunea dorită, suficient de mare pentru a păstra mesajele în tranzit. Când tamponul nu mai este necesar, se folosește **MPI_Buffer_detach**.

Exemplul următor ilustrează utilizarea acestora:

```
#define BuffSize 100000
int size;
char *buff;
MPI_Buffer_attach(malloc(BuffSize), BuffSize);
/* Tamponul de BuffSize octeti poate fi folosit acum */
MPI_Bsend( ... );
MPI_Buffer_detach(&buff, &size);
/* Tamponul redus la zero */
MPI_Buffer_attach(buff, size);
/* Tamponul disponibil din nou */
```

Observații:

- După detașare, utilizatorul poate reutiliza sau dealoca spațiul ocupat de tampon
- Unele operații, de atașare și de detașare, au un argument de tip **void***; ele sunt folosite diferit – un **pointer** la tampon este transmis la *attach*; **adresa pointerului** este utilizată la *detach*, astfel că acest apel întoarce valoarea pointerului. Argumentele sunt definite ambele ca **void *** (și nu **void *** respectiv **void ****) pentru a evita conversii forțate de tip (cast). În exemplul precedent, **&buff**, care este de tip **char ****, poate fi transmisă ca argument fără a folosi **cast**.

2.6.4. Transmitere sincronă

Operația de transmitere sincronă **MPI_Ssend** poate demara înaintea recepției mesajului, dar se termină doar când operația de recepție corespondentă a început să preia mesajul transmis.

2.6.5. Transmitere în modul pregătit

Operația de transmitere în modul pregătit, **MPI_Rsend** poate fi demarată **numai dacă** recepția corespondentă a fost demarată anterior.

2.7. Comunicația non-blocantă

Pentru a permite suprapunerea calculelor cu transferul de date, MPI oferă și operații **send**, **recv** **neblocante**, pentru toate formele de comunicare: standard, cu tampon, sincron, în mod pregătit. Controlul este redat procesului apelant imediat după demararea operației, împreună cu un identificator al cererii (de tip **MPI_Request**). Programatorul trebuie să specifice, prin operații explicite, terminarea transferului. Oricum, el poate programa alte operații (de prelucrare) în paralel cu transferul datelor.

MPI_Isend demarează o operație standard neblokantă de transmisie. Terminarea operației poate fi verificată prin **MPI_Test**. În plus, **MPI_Wait** pune în așteptare procesul executant până la terminarea operației. Pe parcursul desfășurării operației de transmitere, zona care conține mesajul nu trebuie modificată.

Operația **MPI_Irecv** începe o recepție neblokantă. Terminarea ei poate fi testată cu **MPI_Test** și așteptată cu **MPI_Wait**.

Pentru testarea sau așteptarea terminării uneia sau a tuturor operațiilor neblocante ale unei colecții de operații, MPI furnizează operațiile

MPI_Testall

MPI_Testany

MPI_Testsome

MPI_Waitall

MPI_Waitany

MPI_Waitsome

De exemplu, putem începe două operații neblocante, așteptând apoi terminarea ambelor operații. Aceasta ne permite realizarea unei benzi de asamblare în care fiecare proces primește date din stânga și le transmite în dreapta. În timp ce sosesc noile date, precedentele sunt folosite în calcule.

```

MPI_Request request[2];
MPI_Status status[2];
...
while (! done)
{
    MPI_Irecv (buf1, ..., &request[0]);
    MPI_Isend (buf2, ..., &request[1]);
    /* calcule cu buf2 */
    MPI_Waitall (2, request, status);
    /* schimba buf1 cu buf2 */
}

```

Observație:

- Un server cu mai mulți clienți ar trebui să folosească **MPI_Waitsome** și nu **MPI_Waitany**, pentru a nu “îngheța” nici un client.

MPI_Waitsome așteaptă până când cel puțin una din cererile active este terminată, dar întoarce numărul cererilor terminate și lista indicilor acestora (în tabloul de cereri). Se pot trata apoi toate cererile terminate.

MPI_Waitany întoarce indexul unei singure cereri, alese aleator, ceea ce poate duce la “înghețarea” unui anumit client, defavorizat de această alegere.

2.8. Cereri de comunicare persistente

Când ciclul din exemplul anterior (pipeline!) se repetă de multe ori este avantajoasă crearea unei cereri de transmitere / recepție **persistente**, folosite ulterior de oricâte ori. Aceasta simplifică dialogul dintre program și “infrastructura” de comunicare, evitând retransmiterea parametrilor operației, la fiecare apel. Programatorul poate crea o cerere persistentă (**MPI_Send_init**, **MPI_Recv_init**, și toate celelalte variante, cu tampon sincrone și în mod pregătit), poate declanșa operația (**MPI_Start**, **MPI_Startall**) de comunicare, poate aștepta terminarea (**MPI_Wait**) pentru a declanșa o nouă operație de comunicare, sau poate dealoca cererea persistentă cu **MPI_Request_free**.

```

MPI_Recv_init ( ..., &request);
MPI_Start (request); // are semantica operatiei imediate MPI_Irecv
...
MPI_Wait (&request);
MPI_Request_free (&request);

```

2.9. Procese nule

În cazul transferurilor între procesele benzii de asamblare, este convenabil ca pentru procesele de la margine să se considere ca sursă (procesul din stânga) respectiv destinație (procesul din dreapta) niște procese **fictive**.

Valoarea specială **MPI_PROC_NULL** poate fi folosită în locul unui rang de proces sursă sau destinatar. Comunicarea cu un proces fictiv nu are nici un efect. O transmitere de mesaj se termină întotdeauna cu succes. O recepție de la un proces fictiv se încheie imediat fără a modifica tamponul de recepție.

3. Tipuri de date MPI

O caracteristică importantă a MPI este includerea unui argument referitor la tipul datelor transmise / recepționate.

MPI are o mulțime bogată de tipuri predefinite: toate tipurile de bază din C (și din FORTRAN) plus **MPI_BYTE** și **MPI_PACKED**. De asemenea, MPI furnizează constructori pentru tipuri derivate și mecanisme pentru descrierea unui tip general de date.

3.1. Aspecte generale

În cazul general, mesajele pot conține valori de tipuri diferite, care ocupă zone de memorie de lungimi diferite și ne-contigue. În MPI un **tip de date** este un obiect care specifică o secvență de tipuri de bază și deplasările asociate acestora, relative la tamponul de comunicare pe care tipul îl descrie. O astfel de secvență de perechi (tip, deplasare) se numește **harta tipului**. Secvența tipurilor (ignorând deplasările) formează **semnătura tipului** general.

Typemap = {(type₀, disp₀), . . . , (type_{n-1}, disp_{n-1})}

Typesig = {type₀, . . . , type_{n-1}}

Harta tipului, împreună cu o adresă de bază **buf** descriu complet un tampon de comunicare:

- Acesta are n intrări
- Fiecare intrare i are tipul type _{i} și începe la adresa buf+ disp _{i}

Observație:

- Ordinea perechilor în Typemap nu trebuie să coincidă cu ordinea valorilor din tamponul de comunicare.

Putem asocia un **titlu** (handle) unui tip general și putem folosi acest titlu în operațiile de transmitere / recepție, pentru a specifica tipul datelor comunicate.

Tipurile de bază sunt cazuri particulare, predefinite. De exemplu, **MPI_INT** are harta {(int, 0)}, cu o intrare de tip int și cu deplasament zero.

Pentru a înțelege modul în care MPI assemblează datele, utilizăm noțiunea de **extindere** (extent). Extinderea unui tip este spațiul dintre primul și ultimul octet ocupat de intrările tipului, rotunjit superior din motive de aliniere.

De exemplu, Typemap = {(double, 0), (char, 8)} are extinderea **16**, dacă valorile double trebuie aliniate la adrese multiplu de 8. Deci, chiar dacă dimensiunea tipului este **9**, din motive de aliniere, o nouă valoare începe la o distanță de 16 octeți de începutul valorii precedente.

Extinderea și dimensiunea unui tip de date pot fi aflate prin apelurile funcțiilor

```
int MPI_Type_extent (MPI_Datatype datatype, MPI_Aint *extent);
int MPI_Type_size(MPI_Datatype datatype, int *size);
```

Deplasările sunt relative la o anumită adresă inițială de tampon. Ele pot fi substituite prin **adrese absolute**, care reprezintă deplasări relative la “adresa zero” simbolizată de constanta **MPI_BOTTOM**. Dacă se folosesc adrese absolute, argumentul **buf** din operațiile de transfer trebuie să capete valoarea **MPI_BOTTOM**. Adresa absolută a unei locații de memorie se află prin

```
int MPI_Address (void *location, MPI_Aint *address);
```

unde **MPI_Aint** este un tip întreg care poate reprezenta o adresă oarecare (de obicei este `int`) **MPI_Address** reprezintă un mijloc comod aflare a deplasărilor, chiar dacă nu se folosește adresarea absolută în operațiile de comunicare.

3.2. Tipuri derivate

MPI prevede **constructori** de tipuri derivate.

3.2.1. Tipul *contiguu*

Cel mai simplu constructor produce un nou tip de date făcând mai multe copii ale unui tip existent, cu deplasări care sunt multipli ai extensiei tipului vechi.

De exemplu, dacă **oldtype** are harta $\{(int, 0), (double, 8)\}$ atunci noul tip creat prin

```
MPI_Type_contiguous (2, oldtype, &newtype);
```

are harta $\{(int, 0), (double, 8), (int, 16), (double, 24)\}$

Noul tip trebuie încredințat sistemului înainte de utilizare:

```
MPI_Type_commit (&newtype);
```

pentru ca acesta să poată face optimizările de performanță posibile. Când un tip de date nu mai este folosit, el trebuie eliberat cu:

```
MPI_Type_free (&newtype);
```

Observație:

- Utilizarea tipului contiguu este echivalentă cu folosirea unui contor mai mare ca 1 în operațiile de transfer. Astfel apelul:

```
MPI_Send (buffer, count, datatype, dest, tag, comm);
```

este similar cu:

```
MPI_Type_contiguous (count, datatype, &newtype);
```

```
MPI_Type_commit (&newtype);
```

```
MPI_Send (buffer, 1, newtype, dest, tag, comm);
```

```
MPI_Type_free (&newtype);
```

3.2.2. Tipul *vector*

Acesta permite specificarea unor date situate în zone necontigue de memorie. Elementele tipului vechi pot fi separate între ele de spații având lungimea egală cu un multiplu al extinderii tipului (deci cu un **pas constant**). Pentru exemplificare, să considerăm o matrice de 5 linii și 7 coloane (cu elementele de tip `float` memorate pe linii). Pentru a construi tipul de date corespunzător unei coloane folosim operația:

```
MPI_Type_vector (5, 1, 7, MPI_FLOAT, &newtype);
```

unde

- 5 este numărul de blocuri;
- 1 este numărul de elemente din fiecare bloc (în cazul în care acest număr este mai mare decât 1, un bloc se obține prin concatenarea numărului respectiv de copii ale tipului vechi);
- 7 este **pasul**, deci numărul de elemente între începuturile a două blocuri vecine;
- `MPI_FLOAT` este vechiul tip.

Pentru a transmite coloana a treia a matricii folosim:

MPI_Send (&a[0][2], 1, newtype, dest, tag, comm);

3.2.3. Tipul hvector

Este similar tipului **vector**, exceptând faptul că pasul se dă în număr de octeți și nu în număr de elemente.

3.2.4. Tipul indexed

În cazul acestui tip, fiecare bloc are un număr particular de copii ale tipului vechi și un deplasament diferit de ale celorlalte. Deplasamentele sunt date în multipli ai extinderii vechiului tip.

Utilizatorul trebuie să specifice ca argumente ale constructorului de tip un tablou de numere de elemente per bloc și un tablou de deplasări ale blocurilor.

3.2.5. Tipul hindexed

Este similar tipului indexat, cu diferența că deplasările sunt măsurate în octeți.

3.2.6. Tipul struct

Este o generalizare a tipului precedent, prin aceea că permite ca fiecare bloc să constea din replici ale unor tipuri de date diferite.

3.2.7. Exemplu

În multe probleme de simulare din fizică (N-Body) trebuie considerate interacțiuni între particule sau obiecte. O reprezentare posibilă a unei colecții de particule este următoarea:

```
# define MAX_PART 1000
Struct Part_struct
{ int class; /* clasa particulei */
  double d[6]; /* coordonatele particulei */
  char b[7]; /* alte informatii */
};
struct Part_struct particle[MAX_PART];
int i, dest, rank;
MPI_Comm comm;

/* constructia tipului care descrie structura MPI a unei particule, ParticleType */
MPI_Datatype ParticleType;
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR}; /* semnat */
int blocklen [3] = { 1, 6, 7}; /* lungimi blocuri */
MPI_Aint disp[3]; /* deplasari */
int base;

/* calculul deplasarilor */
MPI_Address (particle, disp);
MPI_Address (particle[0].d, disp+1);
MPI_Address (particle[0].b, disp+2);
base=disp[0];
for (i=0; i<3; i++) disp [i] -= base;
MPI_Type_struct ( 3, blocklen, disp, type, &ParticleType);
```

```
/* transmiterea intregului tablou */
```

```
MPI_Type_commit(&ParticleType);
```

```
MPI_Send (particle, MAX_PART, ParticleType, dest, tag, comm);
```

Construim **tipul** ce corespunde particulelor de clasa zero, precedate de numărul acestor intrări. Construcția se face în două etape:

- Se construiește tipul indexat al particulelor cu clasa zero, **Zparticle**; particulele succesive având clasa zero sunt tratate ca un bloc;
- Se construiește tipul struct **Ztype** cu contorul de particule “zero” și particulele propriu zise.

```
MPI_Datatype Zparticle; /* toate particulele cu clasa zero */
```

```
MPI_Aint zdisp [MAX_PART]; /* tabloul deplasarilor blocurilor */
```

```
int zblock [MAX_PART],j,k; /* tabloul numerelor de elem per bloc */
```

```
MPI_Aint zdisp [MAX_PART]; /* tabloul deplasarilor blocurilor */
```

```
MPI_Datatype Ztype; /* include contorul de particule zero */
```

```
int zzblock [2]={1,1}; /* lungimi blocuri, */
```

```
MPI_Aint zzdisp [2]; /* deplasari si */
```

```
MPI_Datatype Zztype[2]; /* tipuri pentru Ztype */
```

```
/* particule consecutive cu clasa 0 sunt tratate ca un bloc */
```

```
j=0; /* numar de blocuri */
```

```
for(i=0; i<MAX_PART; i++) /* parcurge toate particulele */
```

```
if (particle[i].class == 0)
```

```
{ for (k=i+1; (k<MAX_PART)&&(particle[k].class == 0); k++) ;
```

```
zdisp[j] = i;
```

```
zblock[j] = k-i; /* lungimea blocului */
```

```
j ++;
```

```
i = k;
```

```
}
```

```
MPI_Type_indexed(j, zblock, zdisp, ParticleType, &Zparticle);
```

```
/* pregatesc contorul de particule */
```

```
MPI_Address(&j, zzdisp);
```

```
MPI_Address(particle, zzdisp+1);
```

```
zztype[0] = MPI_INT;
```

```
zztype[1] = Zparticle;
```

```
MPI_Type_struct(2, zzblock, zzdisp, zztype, &Ztype);
```

```
MPI_Type_commit(&Ztype);
```

```
MPI_Send (MPI_BOTTOM, 1, Ztype, dest, tag, comm);
```

```
/* adrese absolute pentru valori ale unor variabile diferite */
```

Temă:

- Descrierea transmiterii primelor două coordonate ale tuturor intrărilor
- Descrierea folosind adresa absolută în tipurile de date

4. Topologii de procese

4.1. Generalități

Procesele unui grup de au ranguri de la 0 la $n-1$ (n fiind numărul de procese din grup). În multe aplicații, ordonarea liniară a proceselor, după rang, nu reflectă tiparul de comunicare între procese. Adesea, procesele trebuie aranjate în topologii **grilă** cu două sau mai multe dimensiuni. În cazul general, tiparul de comunicare între procese corespunde unui **graf**. Un astfel de aranjament al proceselor, care reflectă comunicările punct la punct dintre ele reprezintă **topologia virtuală** a grupului de procese.

Trebuie făcută distincție între topologia virtuală a proceselor și **topologia reală** a sistemului pe care acestea sunt executate. Topologia virtuală poate fi exploatată de sistem în plasarea proceselor pe procesoare, pentru a ameliora performanțele. Cum se face acest lucru este o chestiune ce iese din sfera problemelor MPI. Oricum, să reținem că topologia virtuală reflectă caracteristicile aplicației și poate fi folosită în îmbunătățirea performanțelor.

Sunt cunoscute tehnici standard de **mapare** a unor topologii grilă / tor pe hipercuburi sau grile de procesoare. Pentru topologiile graf metodele sunt mai complexe și se bazează adesea pe euristici.

În afara informației furnizate algoritmului de plasare a proceselor pe procesoare, topologiile virtuale au ca rol **facilitarea scrierii** programelor, într-o formă mai ușor de înțeles.

4.2. Topologii virtuale

Topologia virtuală poate fi modelată ca un **graf**, în care nodurile reprezintă procese, iar arcele perechile de procese care comunică.

Nu trebuie ca o comunicare să fie precedată de o **deschidere** explicită de canal. Ca urmare, absența unui arc între două noduri din graf nu înseamnă că procesele respective nu vor putea comunica între ele. Pur și simplu, înseamnă că o astfel de comunicare nu contează la maparea topologiei virtuale de procese pe o topologie reală de procesoare.

Arcele grafului de comunicare nu sunt **ponderate**. Experiențe cu tehnici similare din PARMACS arată că această informație (două noduri sunt sau nu conectate) este suficientă pentru o bună mapare. O altă variantă ar complica interfața MPI și ar face mai dificilă activitatea programatorului, care ar trebui să specifice mai multe informații la conceperea aplicațiilor.

Deși specificarea topologiei în termeni de graf este foarte generală, în multe aplicații specificarea unei **topologii mai simple, regulate**, este mai convenabilă. De exemplu inel, grile bi- sau tri-dimensionale, tor. Aceste topologii sunt complet definite prin numărul de dimensiuni și prin numărul de procese pe fiecare axă de coordonate.

Maparea unor topologii grilă sau tor este mai simplă decât maparea unui graf. De aici și interesul din MPI pentru tratarea separată, explicită a acestora.

Într-o structură **carteziană**, numărătoarea pe fiecare coordonată începe de la zero, iar variația corespunde unei numărări pe linii. De exemplu, pentru o grilă de 2×2 , corespondența între rangul proceselor și coordonatele lor este următoarea:

Coord (x,y)	Rang k
Coord (0,0)	0
Coord (0,1)	1

Coord (1,0)	2
Coord (1,1)	3

O descompunere carteziană bi-dimensională este următoarea:

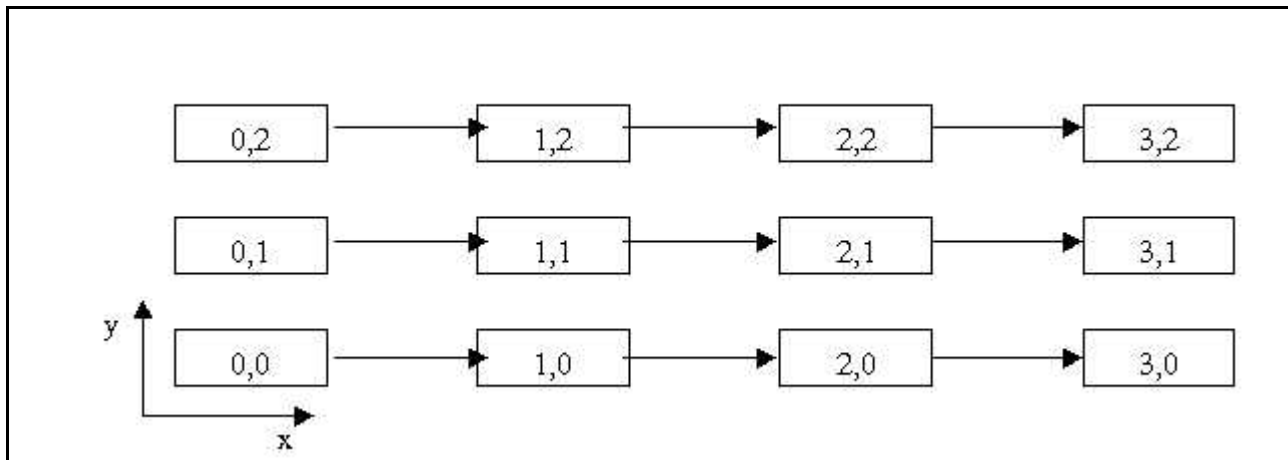


Figura 3.

În figură este schițată și o deplasare cu 1 după prima dimensiune.

4.3. Rutine pentru topologii carteziane

MPI include o colecție de rutine pentru definirea, examinarea, manipularea topologiilor carteziane. Funcția

MPI_Cart_create (comm_old, ndims, dims, periods, reorder, comm_cart);

crează o topologie carteziană a proceselor din grupul specificat de **comm_old** și întoarce un nou comunicator, **comm_cart**, la care este atașată informația de topologie carteziană (ca informație ascunsă).

Utilizatorul poate preciza numărul de dimensiuni **ndims**, și numărul de procese pe fiecare dimensiune în tabloul **dims**. Argumentul **periods** (tablou boolean de dimensiune **ndims**) precizează dacă grila este periodică sau nu, pe fiecare dimensiune (pentru o grilă periodică, ultimul ultimul element pe dimensiunea respectivă este conectat cu primul)

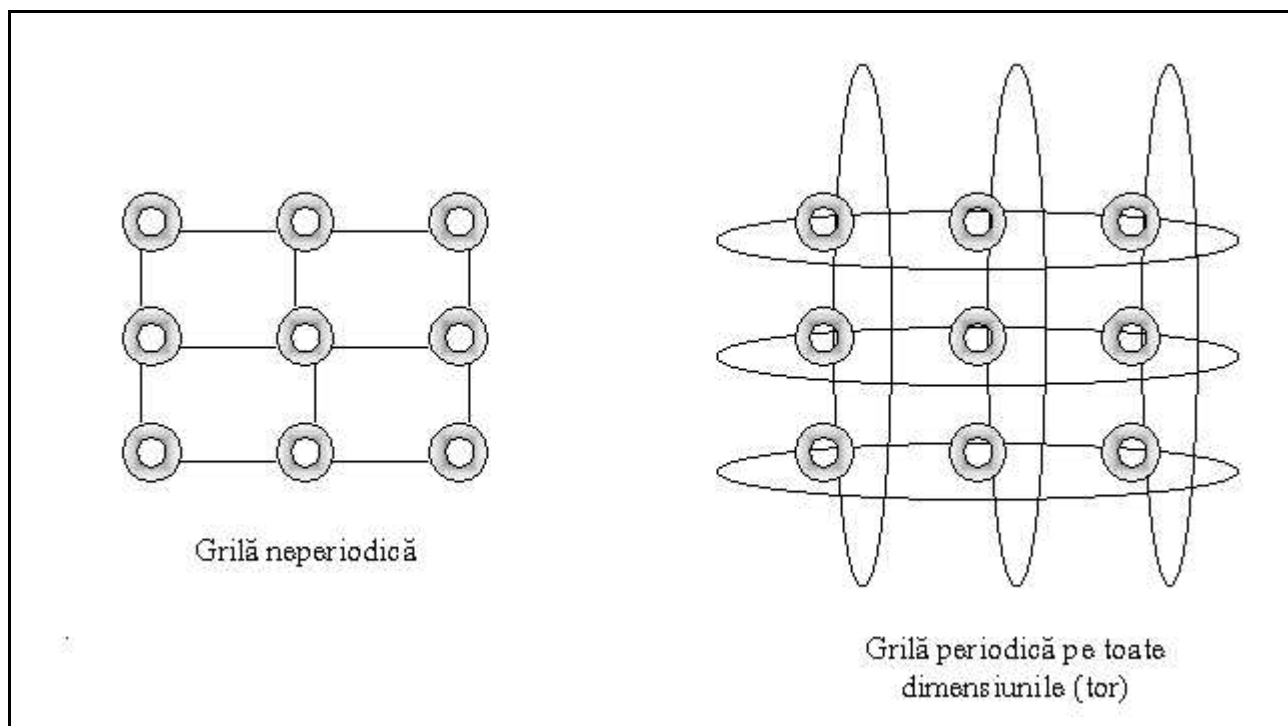


Figura 4

În fine, parametrul **reorder** precizează dacă rangurile proceselor pot fi reordonate sau nu, lăsând sau nu MPI să găsească o asignare convenabilă a elementelor topologiei. Descompunerea din figura 2 poate fi produsă de

```
dims[0] = 4; periods [0] = 0;
dims[1] = 3; periods [1] = 0;
MPI_Cart_create (MPI_COMM_WORLD,2,dims,periods,1,comm_cart);
```

Parametrul **reorder** are valoarea 1, astfel încât se permite reordonarea proceselor.

Vectorul **dims** poate fi calculat convenabil prin funcția

```
MPI_Dims_create (nrnodes, ndims, dims);
```

care realizează o distribuție echilibrată a celor **nrnodes** procese pe cele **ndims** dimensiuni.

Pentru a putea afla **poziția** și, implicit, **vecinii**, unui proces inclus într-o structură carteziană sunt două soluții. Una este

```
MPI_Cart_get (comm, maxdims, dims, periods, coords);
```

care furnizează dimensiunile **dims**, perioadele **periods** și coordonatele **coords** ale procesului apelant. Cea de-a doua

```
MPI_Cart_coords (comm, rank, maxdims, coords);
```

întoarce coordonatele procesului cu rangul **rank** din grupul **comm**. Argumentul **maxdims** dă dimensiunea vectorilor **dims**, **periods** și **coords** în programul apelant.

O altă posibilitate este legată de operația executată de proces, în general o **deplasare** de date de-a lungul unei axe de coordonate. Astfel, să considerăm o grilă bidimensională pentru calculul produsului a două matrice. Fiecare element $[i,j]$ păstrează elementele a_{ij} , b_{ij} ale celor două matrice și calculează elementul c_{ij} al produsului. În acest scop sunt necesare deplasări pe linii ale elementelor **a**, în așa fel încât $[i,j]$ să intre în posesia tuturor elementelor $a_{i1}, a_{i2}, \dots, a_{ij}, \dots, a_{in}$ și deplasări pe coloane ale elementelor **b** astfel încât $[i,j]$ să intre în posesia tuturor elementelor $b_{1j}, b_{2j}, \dots, b_{nj}$. Produsul poate fi

calculat cu formula cunoscută $c_{ij} = \sum_k a_{ik} * b_{kj}$. O operație **MPI_Sendrecv** are ca parametru rangul unui proces sursă, pentru recepție, și rangul unui proces destinație pentru transmisie. Funcția

MPI_Cart_shift(comm, direction, disp, ranksource, rankdest);

furnizează ambele informații. Utilizatorul specifică direcția de coordonate (coordonata a cărei valoare este modificată de deplasare) și dimensiunea pasului, **disp**, pozitiv sau negativ.

În funcție de periodicitatea specificată pentru coordonata respectivă, deplasarea este sau nu circulară. În a doua situație, funcția întoarce valoarea **MPI_PROC_NULL** dacă este cazul (pentru procesele de la margine).

4.3.1. Exemplu

Comunicatorul **comm** are asociată o topologie carteziană bidimensională, periodică. Un tablou bidimensional de valori float este memorat, un element per proces, în variabila **A**. Se dorește modificarea tabloului prin deplasarea după direcția 0 printr-un număr de pași dat de a doua coordonată (**coords[1]**). Mesajele vor avea tagul 0.

```
int coords[2];
MPI_Comm_rank(comm, &rank);
MPI_Cart_coords(comm, rank, 2, coords);
MPI_Cart_shift(comm, 0, coords[1], &source, &dest);
MPI_Sendrecv_replace(&A, 1, MPI_FLOAT, dest, 0, source, 0, comm, &status);
```

4.4. Rutine pentru topologii generale

Pentru o topologie generală, de graf, utilizatorul trebuie să specifice următoarele elemente definitorii:

- Numărul de noduri, apoi pentru fiecare nod
 - Gradul nodului
 - Lista vecinilor

În MPI aceste informații se găsesc într-o formă specifică. Constructorul unei topologii generale (graf) este

MPI_Graph_create(comm_old, nnodes, index, edges, reorder, comm_graph);

Unde parametri **nnodes**, **index** și **edges** definesc structura de graf:

- Parametrul **nnodes** este numărul de noduri din graf, notate 0 . . nnodes-1
- Intrarea **i**-a din tabelul **index** păstrează numărul total de vecini ai primelor **i** noduri, astfel încât **index[0]** este gradul nodului 0; **index[i]-index[i-1]** este gradul nodului **i**
- Listele vecinilor nodurilor 0 . . nnodes-1 sunt păstrate în elemente succesive în tabloul **edges**, astfel încât vecinii nodului 0 sunt în **edges[j]**, cu $0 \leq j < \text{index}[0]$; vecinii nodului **i** sunt în **edges[j]**, cu $\text{index}[i-1] \leq j < \text{index}[i]$

Topologia asociată unui comunicator poate fi aflată cu

MPI_Topo_test(comm, status);

unde **status** este un întreg cu următoarele valori posibile:

- MPI_GRAPH
- MPI_CART
- MPI_UNDEFINED

Informațiile despre topologia graf pot fi aflate cu

- `MPI_Graphdims_get`, care dă numărul de noduri și numărul de muchii din graf
- `MPI_Graph_get`, care dă tablourile **index** și **edges**
- `MPI_Graph_neighbours_count`, care dă numărul de vecini ai unui nod
- `MPI_Graph_neighbours`, care dă tabloul rangurilor vecinilor unui nod

4.4.1. Exercițiu

Fie **comm** un comunicator cu o topologie *shuffle*; grupul are 2^n membri. Fiecare proces, etichetat (a_1, a_2, \dots, a_n), are trei vecini:

- $\text{exchange}(a_1, a_2, \dots, a_n) = a_1, a_2, \dots, \underline{a_n} \ (\underline{a_n} = 1 - a_n)$
- $\text{shuffle}(a_1, a_2, \dots, a_n) = a_2, a_3, \dots, a_n, a_1$
- $\text{unshuffle}(a_1, a_2, \dots, a_n) = a_n, a_1, \dots, a_{n-1}$

De exemplu, pentru $n=3$ avem situația următoare, prezentată apoi în figura 5:

nod	exchange	shuffle	unshuffle
0	1	0	0
1	0	2	4
2	3	4	1
3	2	6	5
4	5	1	2
5	4	3	6
6	7	5	3
7	6	7	7

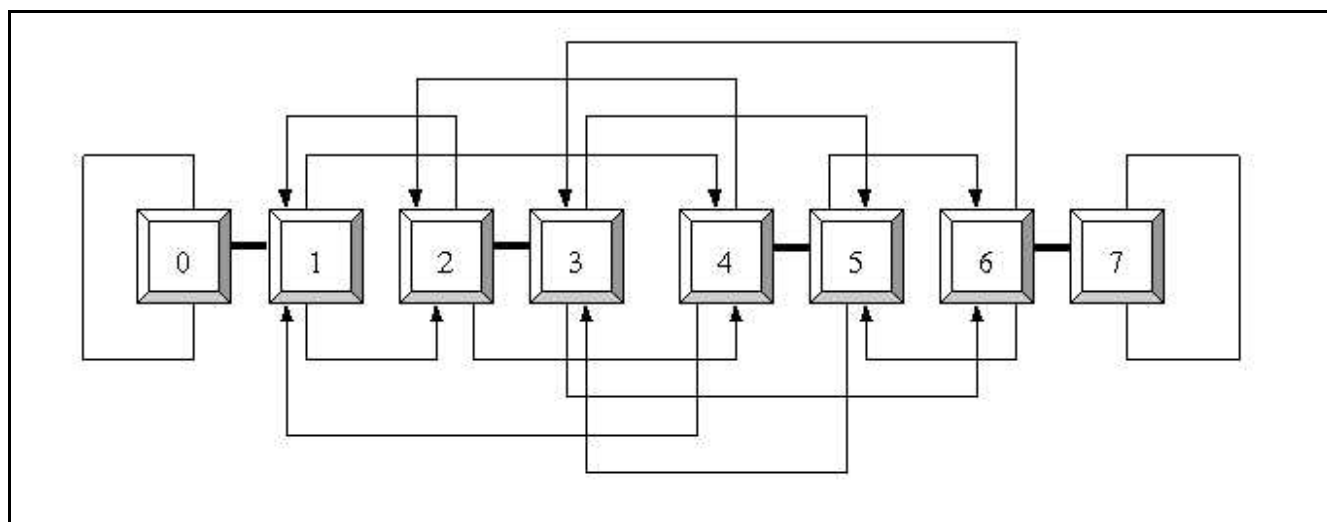


Figura 5

Presupunând că fiecare proces are o variabilă *A* de tip *float*, să se permute valorile *A*, cu înlocuire, succesiv, pentru fiecare.

```
MPI_Comm_rank(comm, &myrank);
```

```
MPI_Graph_neighbours(comm, myrank, 3, neighbours);
```

```

MPI_Sendrecv_replace(&A,1,MPI_FLOAT,neighbours[0],0,neighbours[0], 0, comm, &status);
MPI_Sendrecv_replace(&A,1,MPI_FLOAT,neighbours[1],0,neighbours[2], 0, comm, &status);
MPI_Sendrecv_replace(&A,1,MPI_FLOAT,neighbours[2],0,neighbours[1], 0, comm, &status);

```

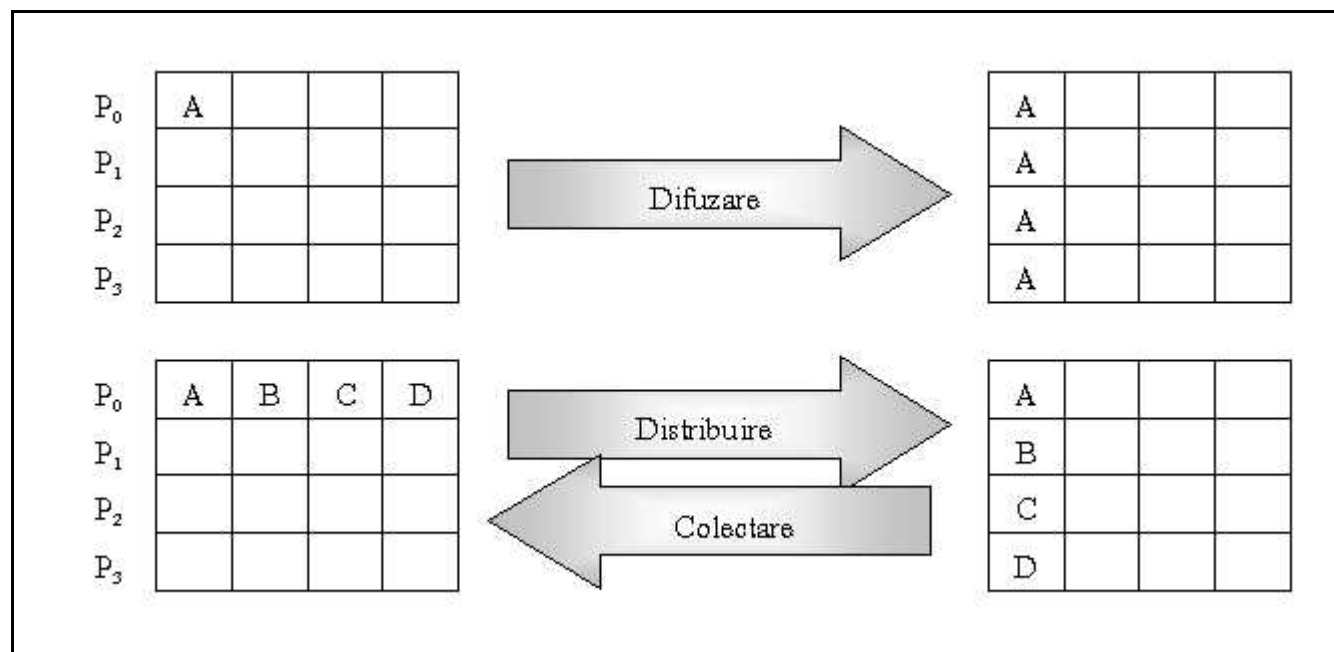
5. Comunicarea colectivă

Operațiile colective implică un grup de procese. Pentru execuție, operația colectivă trebuie apelată de **toate** procesele, cu argumente corespondente. Procesele implicate sunt definite de argumentul **comunicator**, care precizează totodată contextul operației.

Multe operații colective, cum este **difuzarea**, presupun existența unui proces deosebit de celelalte, aflat la originea transmiterii sau recepției. Un astfel de proces se numește **rădăcină**. Anumite argumente sunt specifice rădăcinii și sunt ignorate de celelalte procese, altele sunt comune tuturor.

Operațiile colective se împart în mai multe categorii:

- Sincronizare de tip **barieră** a tuturor proceselor grupului
- **Comunicări** colective, în care se include:
 - Difuzarea
 - Colectarea datelor de la membrii grupului la rădăcină
 - Distribuirea datelor de la rădăcină spre membrii grupului
 - Combinații de colectare / distribuire (allgather și alltoall)
- Calcule colective:
 - Operații de reducere
 - Combinații reducere / distribuire



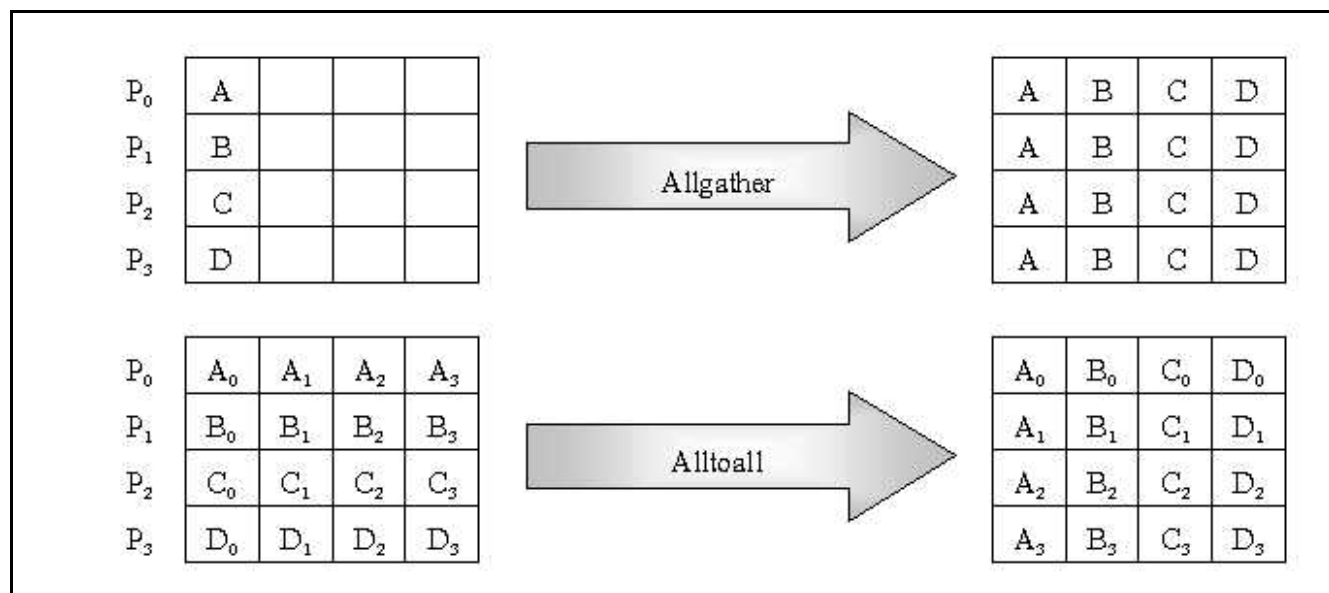


Figura 6

5.1. Caracteristici comune operațiilor colective

5.1.1. Tipul datelor

Cantitatea de date transmisă trebuie să corespundă cu cantitatea de date recepționată. Deci, chiar dacă hărțile tipurilor diferă, semnăturile lor trebuie să coincidă.

5.1.2. Sincronizarea

Terminarea unui apel are ca semnificație faptul că apelantul poate utiliza tamponul de comunicare, participarea sa în operația colectivă încheindu-se. Asta nu arată că celelalte procese din grup au terminat operația. Ca urmare, exceptând sincronizarea barieră, o operație colectivă nu poate fi folosită pentru sincronizarea proceselor.

5.1.3. Comunicator

Comunicările colective pot folosi aceiași comunicatori ca cele punct la punct. MPI garantează diferențierea mesajelor generate prin operații colective de cele punct la punct.

5.1.4. Implementare

Rutinele de comunicare colectivă pot fi bazate pe cele punct la punct. Aceasta sporește portabilitatea, față de implementările bazate pe rutine ce exploatează arhitecturi paralele.

5.2. Difuzarea și calculele colective

Exemplul următor se referă la calculul lui π prin integrare numerică cu formula:

$$\int_0^1 \frac{1}{1+x^2} dx = \arctan(x) \Big|_0^1 = \frac{\pi}{4}.$$

Pentru calculul lui π se consideră integrarea funcției $f(x) = \frac{4}{1+x^2}$, folosind metoda dreptunghiurilor.

Intervalul (0,1) se împarte într-un număr de n subintervale și se însumează ariile dreptunghiurilor având ca bază fiecare subinterval. Pentru execuția algoritmului în paralel, se atribuie, fiecăruia dintre procesele din grup, un anumit număr de subintervale. Cele **două operații colective** care apar în rezolvare sunt:

- Difuzarea valorilor lui n , tuturor proceselor

- Însurarea valorilor calculate de procese

Operația

MPI_Bcast(buffer, count, datatype, root, comm)

difuzează mesajul (buffer, count, datatype) de la procesul **root** la toate procesele grupului din **comm**.

Operația

MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm)

combină valorile din **sendbuf** ale fiecărui proces, folosind operația **op** și pune rezultatul în **recvbuf** în procesul **root**.

Tamponul de intrare este definit de (sendbuf, count, datatype), iar cel de ieșire de (recvbuf, count, datatype). Ambele tampoane au același număr de elemente, cu același tip. În cazul în care fiecare proces specifică o secvență de elemente, operația se realizează separat, pentru fiecare intrare a secvenței.

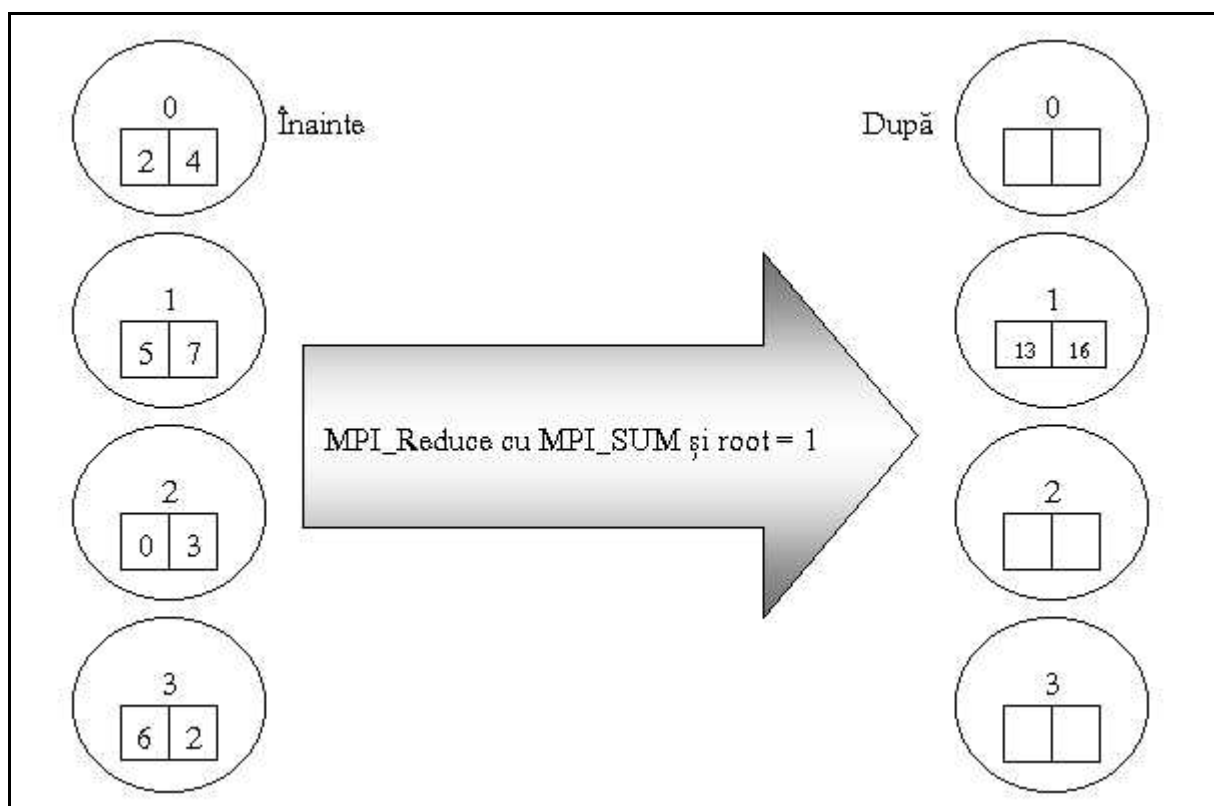


Figura 7

```
# include "mpi.h"
# include <math.h>

int main (int argc, char **argv)
{ int n, myid, numprocs, i, rc;
  double mypi, pi, h, sum, x, a;
  MPI_Status status;

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank (MPI_COMM_WORLD, &myid);
```

```

while (1)
{
    if (myid==0)
    {
        printf("Numar subintervale (0=gata): ");
        scanf("%d",&n);
    }
    MPI_Bcast (&n, 1, MPI_INT, 0, 99, MPI_COMM_WORLD);
    if(!n) break;
    h=1.0/(double)n;
    sum=0.0;
    for (i = myid + 1; i <= n; i += numprocs)
    {
        x = h*((double)i-0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    mypi = h * sum;
    MPI_Reduce (&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                MPI_COMM_WORLD);
    if (!myid) printf("pi este %f\n", pi);
}
MPI_Finalize();
}

```

MPI furnizează o mulțime de operatori de reducere asociativi și comutativi, pentru aflarea maximului, minimumului, sumei, produsului, conjuncției, disjuncției, etc.

5.3. Comunicări colective

Pentru exemplificare, considerăm transferul, de la fiecare proces la rădăcină, a 100 de valori întregi. La destinație, grupurile de 100 de valori sunt plasate la distanța **stride** între ele.

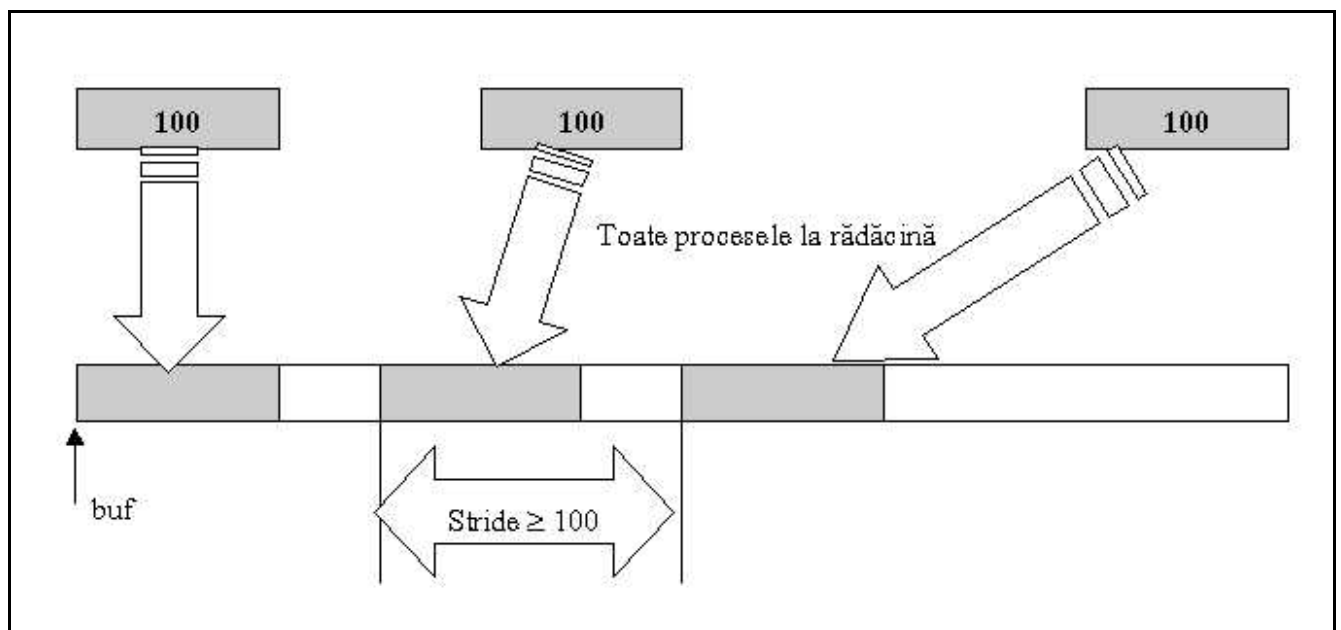


Figura 8

Se utilizează operația

```
MPI_Gatherv (sendbuf, sendcount, sendtype, recvbuf, recvcunts, displs, recvtype, root, comm);
```

care permite ca fiecare proces (inclusiv root) să transmită conținutul tamponului său (sendbuf, sendcount, sendtype) procesului **root**. Contoarele pot fi diferite, astfel că la **root** valorile care le corespund sunt reprezentate de un tablou, **recvcunts**. De asemenea, datele sunt plasate în tamponul **recvbuf** al procesului **root**, în poziții care sunt descrise de vectorul **displs**. Datele transmise de procesul *j* sunt plasate începând din poziția displs[*j*] (măsurată în elemente de **recvtype**).

Tamponul de recepție este ignorat pentru toate procesele diferite de root. Specificația contoarelor și deplasărilor nu trebuie să conducă la scrierea de mai multe ori a unei locații din root.

```
MPI_Comm comm;
int gsize, sendarray[100];
int root, *rbuf, stride;
int *displs, i, *rcounts;
...
MPI_Comm_size (MPI_COMM_WORLD, &gsize);
rbuf = (int *) malloc (gsize * stride * sizeof(int));
displs = (int *) malloc (gsize * sizeof(int));
rcounts = (int *) malloc (gsize * sizeof(int));

for (i = 0; i < gsize; i ++)
{ displs [i] = i * stride;
  rcounts [i] = 100;
}
MPI_Gatherv(sendarray, 100, MPI_INT, vbuf, rcounts, displs, MPI_INT, root, comm);
```

Observație.

- Programul este **eronat** pentru stride <100!

5.4. Corectitudinea operațiilor colective

Așa cum s-a precizat mai devreme, operațiile colective pot avea sau nu efectul de **sincronizare** a tuturor proceselor din grup. Ca urmare, programele trebuie să invoce operațiile colective astfel încât să se evite blocarea definitivă, **indiferent dacă operațiile sunt sau nu sincronizate**. De aici rezultă câteva reguli:

- Operațiile colective trebuie executate în aceeași ordine de toate procesele grupului
- Nu trebuie să existe dependențe circulare între operații
- Operațiile punct la punct blocante nu trebuie să “încalece” operațiile colective

Următorul exemplu este eronat:

```
switch(rank)
{ case 0: MPI_Bcast (... , 0, comm);
  MPI_Send (... , 1, tag, comm);
  break;
case 1: MPI_Recv (... , 0, tag, comm, status);
  MPI_Bcast (... , 0, comm);
  break;
```

}

A treia regulă evidențiază un aspect foarte important: deși contextele operațiilor punct la punct și colective, asociate unui comunicator sunt distincte și, deci, traficul operațiilor punct la punct este independent de traficul generat de operațiile colective o programare necorespunzătoare a operațiilor poate conduce la interblocarea proceselor, datorită posibilelor restricții impuse de sincronizarea operațiilor.

5.5. Determinismul operațiilor

Modelele de programare bazate pe comunicarea de mesaje sunt nedeterminate: ordinea sosirii a două mesaje transmise de două procese A și B unui al treilea proces C este nedefinită. (din contră, MPI asigură că două mesaje transmise de un același proces A ajung în aceeași ordine la B). Este **rolul programatorului** de a asigura că un calcul este determinist, acolo unde el trebuie să fie așa.

Determinismul poate fi garantat prin definirea unui canal pentru fiecare pereche de procese care comunică. În acest caz, procesul C poate face diferența între mesajele primite de la A și cele primite de la B, deoarece ele sosesc pe canale diferite. **MPI nu acceptă conceptul de canal**, dar are mecanisme **similare**, care permit unui receptor să specifice **sursa** mesajului, **tag-ul** și **contextul**. Oricum, deoarece MPI permite specificarea oricărei surse, MPI_ANY_SOURCE și a oricărui tag, MPI_ANY_TAG, folosirea acestor parametri trebuie făcută cu grijă.

Să considerăm următorul fragment de program, în care un proces transmite și primește de **max** ori mesaje de **date** și apoi un mesaj rezultat.

```
for(i=0;i<max;i++)
{ MPI_Send (buff, 600, MPI_FLOAT, r1, 1, MPI_COMM_WORLD);
  MPI_Recv (buff, 600, MPI_FLOAT, MPI_ANY_SOURCE, MPI_ANY_TAG,
    MPI_COMM_WORLD, &status);
    .
    .
    .
}
MPI_Send (buff, 300, MPI_FLOAT, r2, 2, MPI_COMM_WORLD);
MPI_Recv (buff, 300, MPI_FLOAT, MPI_ANY_SOURCE, MPI_ANY_TAG,
  MPI_COMM_WORLD, &status);
```

Deoarece nu se specifică nici o sursă și nici un tag în operațiile de recepție, ordinea între ultimul mesaj de date și cel de rezultate se poate inversa, conducând la un rezultat eronat. Eroarea poate fi îndreptată prin specificarea unor valori precise pentru tag-uri și, eventual, pentru procesele sursă.

6. Dezvoltarea bibliotecilor paralele peste MPI

Bibliotecile construite peste MPI și orientate către aplicații (de exemplu de algebră liniară) oferă un grad sporit de abstractizare și de portabilitate. Creare unor astfel de biblioteci necesită următoarele caracteristici, pe care MPI le posedă:

- Un spațiu de comunicare **sigur**, care garantează că bibliotecile pot comunica fără a avea conflicte cu comunicări din exteriorul său;
- Limitarea operațiilor colective la nivelul grupului de procese, evitând sincronizări inutile, cu procese ce execută alte operații, independente de cele ale grupului;
- Un mecanism de **denumire** abstractă a proceselor, care să permită bibliotecilor să descrie comunicările în termeni convenabili propriilor algoritmi și structuri de date;

- Posibilitatea de a îmbogăți o mulțime de procese comunicante cu **attribute** definite de utilizator, ceea ce permite extinderea operațiilor de comunicare.

Așa cum s-a arătat anterior, tag-urile mesajelor furnizează un mecanism pentru deosebirea mesajelor folosite în scopuri diferite. Ele nu sunt însă suficiente pentru modularizarea programelor. Să considerăm o aplicație care apelează o rutină de bibliotecă (ce face, de exemplu, transpunerea unei matrici). Este important să se asigure că tag-urile mesajelor folosite în bibliotecă sunt distincte de cele folosite în restul aplicațiilor. Altfel pot apare erori. În figură sunt reprezentate patru procese, care apelează o rutină de bibliotecă (reprezentată prin zona dreptunghiulară). Unul din procese termină mai devreme rutina, după care generează un mesaj (linia punctată) interceptat de bibliotecă. Utilizarea unor contexte diferite evită această situație (figura din dreapta).

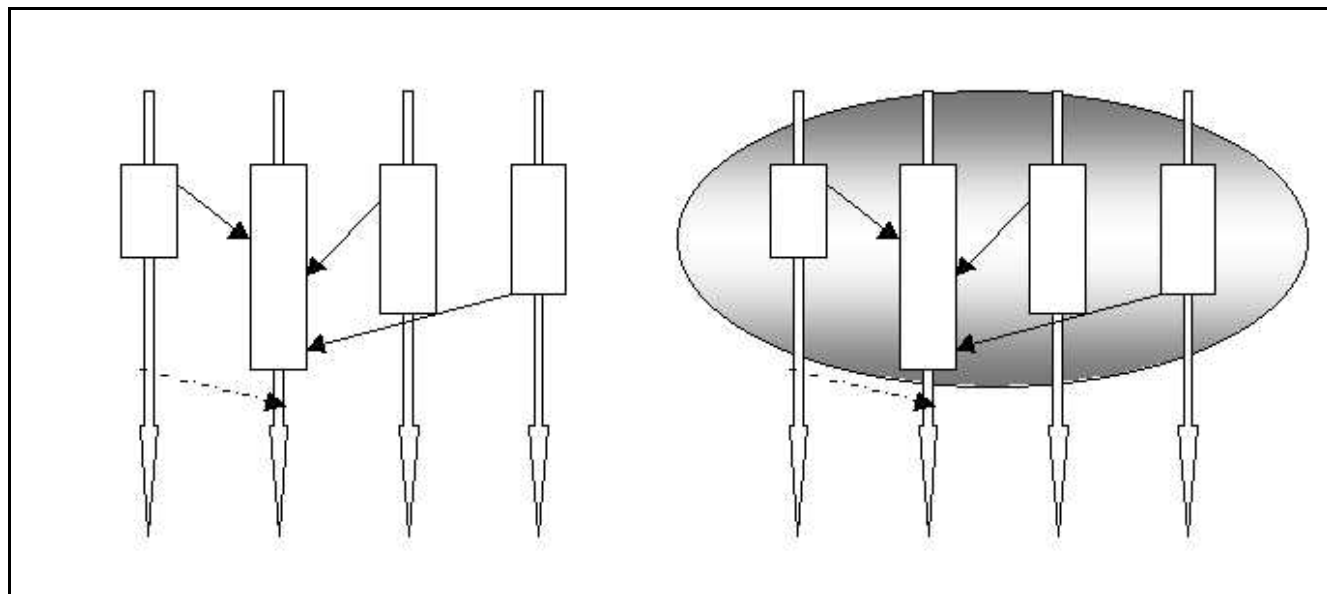


Figura 9

Apelul funcției

MPI_Comm_dup (comm, &newcomm)

permite crearea unui nou comunicator care include același grup de procese, dar cu un nou context. Pentru a rezolva problema anterioară, generată de **compunerea secvențială** a unor programe paralele, se generează un nou comunicator, care este trimis ca parametru rutinei de bibliotecă.

MPI_Comm_dup (comm, &newcomm)

Transpose (newcomm, A)

MPI_Comm_free (&newcomm)

Rutina transpose va fi creată astfel încât să facă uz de comunicatorul transmis ca argument.

Compunerea paralelă denotă execuția paralelă a două sau mai multor componente de program pe mulțimi disjuncte de procesoare. Abordarea prin task-uri paralele (figura stânga), se crează dinamic task-uri care execută componente diferite de program. În figură sunt reprezentate patru noi task-uri, de două “tipuri” diferite, pentru a executa două componente diferite de program.

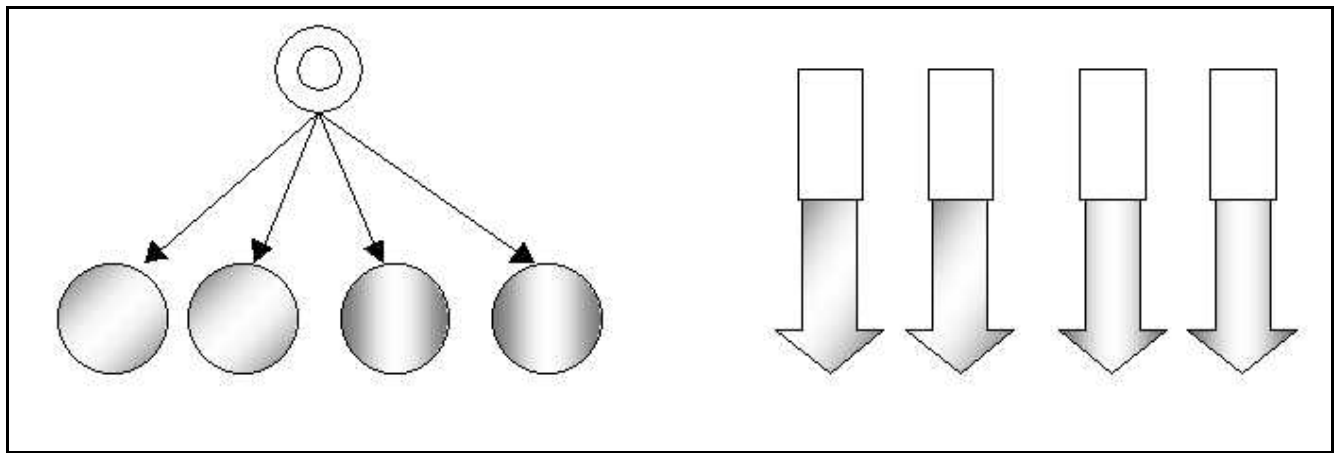


Figura 10

În abordarea MPI, procesele își schimbă “caracterul”: procesele disponibile sunt partiționate în două seturi, corespunzător celor două componente de program de executat. Pentru aceasta se folosește apelul

MPI_Comm_split (comm, color, myid, &newcomm)

care este o operație colectivă, executată de procesele din grupul **comm**. Ea produce comunicatoare noi, câte unul pentru fiecare valoare distinctă argumentului **color**. De exemplu secvența următoare produce trei comunicatoare noi, dacă grupul inițial conține cel puțin trei procese.

MPI_Comm_rank (comm, &myid);

color=myid%3;

MPI_Comm_split (comm, color, myid, &newcomm)

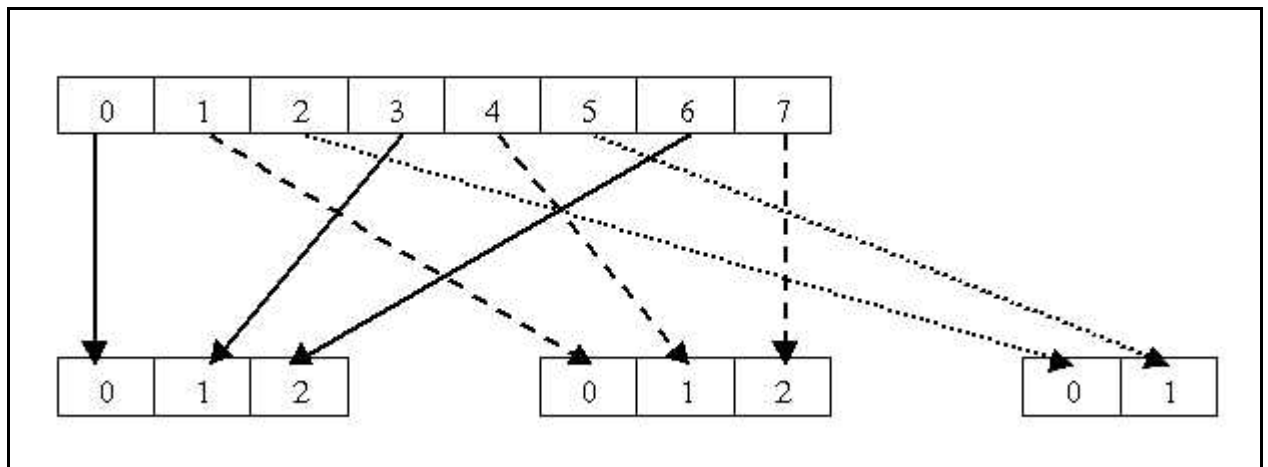


Figura 11

Pentru a permite crearea unor biblioteci robuste, MPI furnizează următoarele mecanisme:

- Contexte
- Grupuri
- Topologii virtuale
- Atribute ascunse
- Comunicatori

În MPI comunicarea poate avea loc în cadrul unui grup (**intra-comunicare**), sau între două grupuri distincte (**inter-comunicare**). Corespunzător, comunicatorii se împart în două categorii: **intra_comunicatori** și **inter_comunicatori**.

Un intra-comunicator descrie:

- Un grup de procese
- Contexte pentru comunicare punct la punct și colectivă (aceste două contexte sunt disjuncte, astfel că un mesaj punct la punct nu se poate confunda cu un mesaj colectiv, chiar dacă au același tag)
- O topologie virtuală (eventual)
- Alte attribute

MPI are serie de operații pentru manipularea grupurilor.

- Aflarea grupului asociat unui comunicator
- Găsirea numărului de procese dintr-un grup și a rangului procesului apelant
- Compararea a doua grupuri
- Reuniunea, intersecția, diferența a două grupuri
- Crearea unui nou grup din altul existent, prin includerea sau excluderea unor procese
- Desființarea unui grup

Funcții similare sunt prevăzute pentru manipularea intra-comunicatorilor:

- Găsirea numărului de procese sau a rangului procesului apelant
- Compararea a doi comunicatori
- Duplicarea, crearea unui comunicator
- Partiționarea grupului unui comunicator în grupuri disjuncte
- Desființarea unui comunicator

Un **inter-comunicator** leagă două grupuri, împreună cu contextele de comunicare partajate de cele două grupuri. Contextele sunt folosite doar pentru operații punct-la-punct, neexistând comunicare colectivă intergrupuri. Nu există topologii virtuale în cazul intercomunicării.

O inter-comunicare are loc între un proces **inițiator**, care aparține unui **grup local** și un proces țintă, care aparține unui **grup distant**. Procesul țintă este specificat printr-o pereche (comunicator, rang) relativă la grupul distant.

În implementarea MPI un comunicator este reprezentat în fiecare proces printr-un tuplu care conține:

- Grupul
- Contextul_send
- Contextul_receive
- Sursa

Pentru intercomunicări, discuția trebuie să se refere la ambele procese, inițiator și țintă. Fie P un proces în grupul GP, care are un inter-comunicator C_{GP} și un proces Q în GQ cu un inter-comunicator C_{GQ} . Atunci:

- $C_{GP}.group$ descrie grupul Q, iar $C_{GQ}.group$ descrie grupul P
- $C_{GP}.context_send = C_{GQ}.context_receive$ și contextul este unic în GQ
- $C_{GQ}.context_receive = C_{GP}.context_send$ și contextul este unic în GP
- $C_{GP}.sursa$ este rangul lui P în GP și $C_{GQ}.sursa$ este rangul lui Q în GQ.

Când P transmite un mesaj lui Q folosind inter-comunicatorul, P folosește tabela grupului pentru a afla adresa absolută a lui Q. El adaugă mesajului informația despre **sursă** și **contextul_send**. Q are o recepție cu un argument **sursă** explicit, folosind inter-comunicatorul. Atunci Q compară **contextul_receive** cu contextul din mesaj și argumentul **sursa** cu **sursa** din mesaj.

MPI permite următoarele operații relative la inter-comunicatori:

- Testul unui comunicator pentru a afla dacă este intra sau inter
- Aflarea numărului de procese din grupul distant
- Aflarea grupului distant
- Legarea a doi **intra**-comunicatori într-un **inter**-comunicator

MPI_Intercomm_create (local_comm, local_leader, peer_comm, remote_leader, tag, newintercommunicator);

- Crearea unui intra-comunicator prin unirea grupurilor local și distant ale unui inter-comunicator

MPI_Intercomm_merge (inter_comm, high, newintracomunicator);

- Duplicarea și desființarea unui inter-comunicator

Crearea unui inter-comunicator presupune că doi membri, selectați câte unul din fiecare grup (**leader**), pot comunica între ei. Deci există un comunicator căruia cei doi lideri îi aparțin. În plus:

- Fiecare leader știe rangul celuilalt în **peer_comm**
- Membrii fiecărui grup cunosc rangul leader-ului lor

Un inter-comunicator este creat printr-o **operație colectivă** executată în cele două grupuri ce urmează a fi conectate. Procesele trebuie să specifice un **intra-comunicator** local, care identifică procesele incluse în grupul lor; trebuie să convină asupra unui **leader** în fiecare grup și asupra unui comunicator “părinte” care leagă liderii și prin care conexiunea poate fi stabilită. Cei doi lideri sunt identificați prin (local_comm, local_leader) respectiv (peer_comm, remote_leader).

6.1.1. Exemplu

Un inel de trei grupuri.

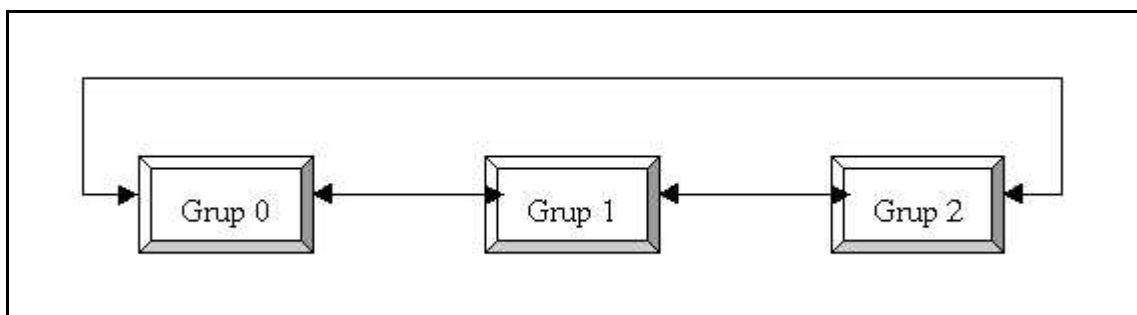


Figura 12

Fiecare grup necesită doi inter-comunicatori. Mai întâi se partiționează grupul de procese MPI_COMM_WORLD în trei subgrupuri, utilizându-se drept “culoare” restul modulo 3 al rangului, păstrat în variabila **membershipkey**. Se folosește operația

MPI_Comm_split (comm, color, key, newcomm)

În noul grup, procesele sunt ordonate după cheie, coliziunile fiind rezolvate prin utilizarea rangului proceselor din grupul inițial. Apoi se construiesc inter-comunicatorii, câte doi pentru fiecare grup. **MPI_Intercomm_create** este o operație colectivă peste grupurile local și distant. În fiecare grup procesele trebuie să prevadă argumente **local_comm** și **local_leader identice**. Apelul folosește comunicarea punct la punct cu comunicatorul **peer_comm** și cu **tag**, între liderii grupurilor.

```

int main (int argc, char **argv)
{
    MPI_Comm myComm;           /* intra-comm grup local */
    MPI_Comm myFirstComm;      /* inter-comm */
    MPI_Comm mySecondComm;
    MPI_Status status;
    int rank, membershipkey;

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    ...
    /* generez membership key in domeniul {0, 1, 2} */
    membershipkey = rank % 3;
    /* construiesc intra-comunicator pentru subgrup local */
    MPI_Comm_split (MPI_COMM_WORLD, (color=membershipkey), (key=rank), &mycomm)
    /* construiesc inter-comunicatorii, taguri fixe */
    if (membershipkey==0) /* grupul 0 comunica cu 1 si 2 */
    { MPI_Intercomm_create (myComm, 0, MPI_COMM_WORLD, 1, 1,
        &myFirstComm);
      MPI_Intercomm_create (myComm, 0, MPI_COMM_WORLD, 2, 2,
        &mySecondComm);
    }
    else if (membershipkey==1) /* grupul 1 comunica cu 0 si 2 */
    { MPI_Intercomm_create (myComm, 0, MPI_COMM_WORLD, 0, 1,
        &myFirstComm);
      MPI_Intercomm_create (myComm, 0, MPI_COMM_WORLD, 2, 12,
        &mySecondComm);
    }
    else if (membershipkey==2) /* grupul 2 comunica cu 0 si 1 */
    { MPI_Intercomm_create (myComm, 0, MPI_COMM_WORLD, 0, 2,
        &myFirstComm);
      MPI_Intercomm_create (myComm, 0, MPI_COMM_WORLD, 1, 12,
        &mySecondComm);
    }
    /* alte operatii */
    ...
    /* eliberare comunicatorilor si terminare */
    MPI_Comm_free (&myFirstComm);
    MPI_Comm_free (&mySecondComm);
    MPI_Comm_free (&myComm);

    MPI_Finalize();
}

```

Parametrii subliniați au în ordine semnificațiile următoare:

- Leader

- Comunicator comun al leaderilor
- Rangul leaderului din grupul distant în comunicatorul comun
- Tag sigur

Deoarece funcțiile relative la topologie sunt aplicabile doar intra-comunicatorilor, nu și inter-comunicatorilor, **MPI_Intercomm_merge** este o soluție pentru a construi un intra-comunicator și a induce apoi pe el o topologie carteziană sau graf.

7. Facilități MPI 2

7.1. Procese dinamice

MPI presupune existența unui mediu de execuție a proceselor, cu care interacțiunea este păstrată la un nivel scăzut pentru a evita compromiterea portabilității. Interacțiunea se limitează la următoarele aspecte:

- Un proces poate porni *dinamic* alte procese prin **MPI_Comm_spawn** și **MPI_Comm_spawn_multiple**
- Poate comunica printr-un argument **info** informații despre unde și cum să pornească procesul
- Un atribut **MPI_UNIVERSE_SIZE** al **MPI_COMM_WORLD** precizează câte procese pot rula în total, deci câte pot fi pornite dinamic, în plus față de cele în execuție

Comanda

```
MPI_Comm_spawn(command, argv, maxprocs, info, root, comm, intercomm, array_of_errcodes);
```

Încearcă să pornească **maxprocs** copii ale unui program MPI specificat de comanda **command**; **argv** conține argumentele transmise programului, în forma unui tablou de șiruri de caractere. Parametrul **info** conține informații adiționale pentru mediul de execuție în forma unor perechi de șiruri de caractere (cheie, valoare). Informația comunicată prin **info** este transparentă pentru MPI. Operația este **executată în colectiv** de procesele grupului **comm**, dar argumentele anterioare sunt examinate doar de procesul **root** din acest grup. Procesele puse în execuție sunt considerate fii ai proceselor care execută comanda. Aceștia au propriul lor **MPI_COMM_WORLD**, distinct de acela al părinților. Apelul întoarce un inter-comunicator pentru comunicare între părinți și fii. Acesta conține procesele părinte în grupul local și procesele copii în grupul distant. Inter-comunicatorul poate fi obținut de copii prin funcția **MPI_Comm_get_parent**.

MPI permite stabilirea unor “canale” de comunicare între procese, chiar dacă acestea nu împart un comunicator comun. Aceasta este utilă în următoarele situații:

- Două părți ale unei aplicații, pornite independent, trebuie să comunice între ele
- Un instrument de vizualizare vrea să se atașeze la un proces în execuție
- Un server vrea să accepte conexiuni de la mai mulți clienți; serverul și clienții pot fi programe paralele.

7.2. Accesul distant la memorie RMA (Remote Memory Access)

RMA extinde mecanismele de comunicare MPI, permițând unui proces să specifice parametrii de comunicare pentru ambele capete implicate într-un tampon de date. Prin aceasta, un proces poate citi, scrie sau actualiza date din memoria altui proces, fără ca al doilea proces să fie explicit implicat în transfer. Cele trei operații sunt reprezentate în MPI prin **MPI_Put**, **MPI_Get**, **MPI_accumulate**.

În afara acestora, MPI furnizează operații de *inițializare*, **MPI_Win_create** care permit fiecărui proces al unui grup să specifice o “fereastră” în memoria sa, pusă la dispoziție celorlalte pentru RMA, și

terminare, **MPI_Win_free** pentru eliberarea ferestrei, ca și operații de *sincronizare* a proceselor care fac acces la datele altui proces.

Comunicările RMA intră în două categorii:

- Cu țintă **activă**, în care toate datele sunt mutate din memoria unui proces în memoria altuia și ambele sunt implicate în mod implicit. Un proces prevede toate argumentele pentru comunicare, al doilea participa doar la sincronizare
- Cu țintă **pasivă**, unde datele sunt mutate din memoria unui proces în memoria altui proces, și numai unul este implicat în mod implicit în comunicație – două procese pot comunica făcând acces la aceeași locație într-o fereastră a unui al treilea proces (care nu participă explicit la comunicare)

MPI 2 prevede și alte facilități, cum ar fi operații de intrare/iesire paralele, la care nu ne referim aici.

8. Implementarea MPI

MPI este implementat pe o mare diversitate de sisteme, de la cele de înaltă performanță la colecții de stații de lucru (Workstation Clusters). Interesul pentru această din urmă a crescut odată cu introducerea noilor tehnologii de comunicare de mare viteză: ATM, FDDI, Myrinet, FastEthernet, etc. Există deja implementări MPI construite pe baza protocolului TCP/IP, care au performanțe reduse.

Dintre multiplele implementări MPI am ales **MPICH**, dezvoltată la Argonne National Laboratory & Mississippi State University. Portabilitatea acestei implementări derivă din construcția ei pe baza unui număr restrâns de funcții care crează independența de hardware, formând împreună o interfață de echipament abstract, ADI (Abstract Device Interface). Pentru a porta MPI pe alt sistem este necesară implementarea funcțiilor ADI.

ADI încapsulează detaliile și complexitatea suportului hardware al comunicației într-un modul separat. El se rezumă la servicii de **comunicare punct la punct**. Peste el, restul MPICH conține implementarea întregului standard, incluzând:

- Gestiunea comunicatorilor
- Tipuri de date derivate
- Operații colective

Rutinele **ADI de bază** se rezumă la următoarele funcții:

- Inițializare și terminare ADI **MPID_Init**, **MPID_End**
- Inițializarea unei transmisii / recepții de mesaj, **MPID_Port_send** (**send_ready**, **send_sync**), **MPID_Recv**. . .
- Testul terminării unei operații sau a sosirii unui anume mesaj, **MPID_Test_send**, **MPID_Test_recv**, **MPID_Iprobe**
- Terminarea unei transmisii / recepții, **MPID_Complete_send**, **MPID_Complete_recv**
- Anularea unei operații, **MPID_Cancel**
- Verificarea unei operații în așteptare, **MPID_Check_device**
- Aflarea unor informații legate de rang sau număr de procese, **MPID_Myrank**, **MPID_Mysize**.

Ideea de bază ADI este împărțirea aplicațiilor de comunicare în **două faze distincte**, inițierea și terminarea. Funcțiilor care realizează aceste faze le sunt adăugate alte câteva, pentru acumularea unor cereri sau testul terminării execuției lor.

Desigur, implementarea ADI pe o anumită platformă trebuie să țină seama de primitivele de comunicare oferite de aceasta. Ne referim în continuare la implementarea pe o colecție de stații de

lucru conectate prin **Myrinet** folosind o bibliotecă FM (**Fast Messages**). Biblioteca FM are trei **primitive**:

- FM_send_4 (dest, handler, i₀,i₁,i₂,i₃), care transmite un mesaj de 4 cuvinte
- FM_send (dest, handler, buff, size), care transmite un mesaj lung
- FM_extract (), care prelucrează mesaje primite

FM diferă de alte metode de transfer de mesaje prin aceea că are două operații de transmitere asincronă, dar nu există operații corespunzătoare de recepție. În schimb, fiecare mesaj include numele unui **handler**, care este o funcție definită de utilizator, ce va fi invocată la sosirea mesajului și care va prelucra datele comunicate. Când este invocată, **FM_extract()** prelucrează mesajele sosite (aflate în așteptare), le scoate din coada mesajelor sosite și pune în execuție handler-ul corespunzător. FM garantează recepția **sigură și ordonată** a mesajelor cu o degradare minimă de performanțe.

Întreaga implementare are trei niveluri. Figura următoare arată modul de compunere a primitivelor **MPI_send** și **MPI_Recv** în termenii primitivelor diferitelor niveluri. Prefixul MPID denotă funcțiile ADI. Funcția **FM_extract** și rutina **async_handler** sunt conectate prin linii punctate pentru a arăta că ele răspund de evoluția funcțiilor ADI corespunzătoare, deși nu sunt apelate direct de acestea.

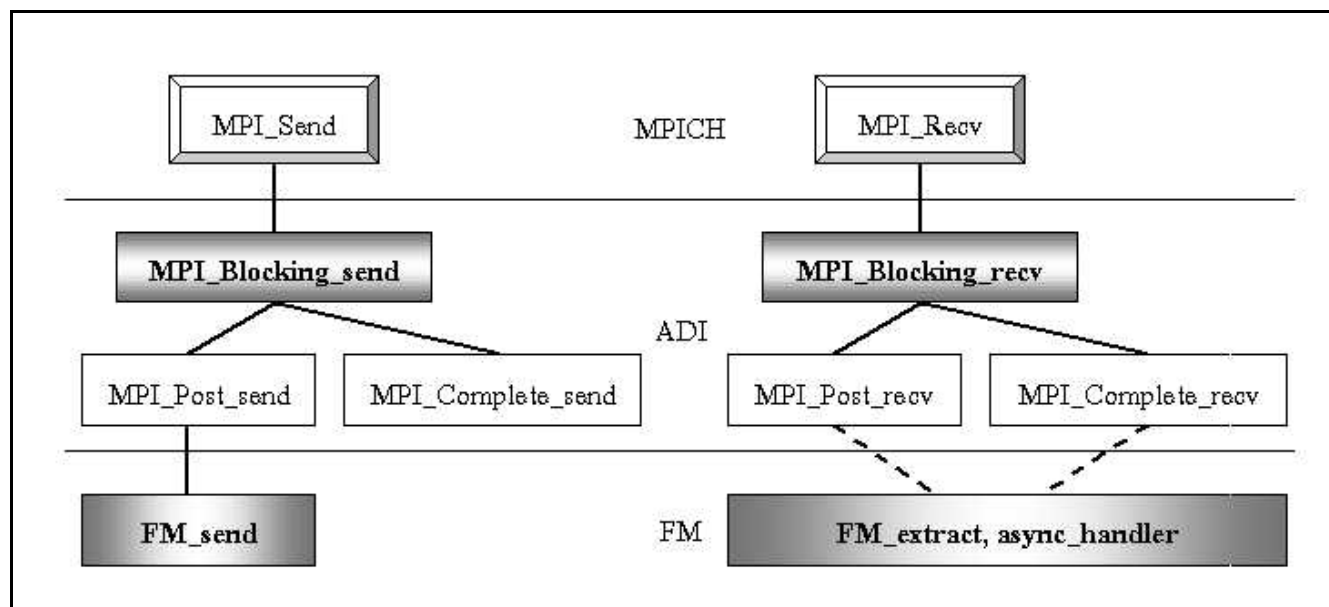


Figura 13

Primitivele ADI sunt, semantic, mai apropiate de MPI decât de FM, astfel că fiecare primitivă de comunicare punct la punct MPI își găsește descriere în câteva linii de cod în termenii ADI. Diferențe mai mari există față de FM:

- Nu există o corespondență unu la unu între primitivele ADI și FM
- Primitivele ADI sunt orientate pe faze (start – terminare)
- FM nu are o operație de recepție explicită

Orientarea pe faze cere ca **starea** să fie gestionată pentru fiecare operație în curs. Pe de altă parte, comunicarea în FM se bazează pe un modul fără stări. Pentru a suplini diferența, se păstrează două cozi, **receive** și **send**, fiecare păstrând o descriere a cererilor și a stărilor lor. În plus, la recepție, o coadă **unexpected** păstrează evidența mesajelor sosite, pentru care nu există o cerere de recepție corespondentă.

Fiecare element al cozilor memorează un **pointer** spre spre un tampon temporar și descrierea mesajului corespunzător. Folosirea tampoanelor temporare accelerează recepția mesajelor, față de soluția păstrării acestora la sursă până la apariția unei cereri de recepție. În locul unei operații de **recepție** implicite, FM se bazează pe funcții definite de utilizatori (**handles**). Mesajele sosite sunt puse

în coadă în memoria calculatorului gazdă și sunt extrase cu **FM_extract**, care execută **handler**-ul corespunzător. Deci, implementarea primitivei ADI de recepție înseamnă scrierea **handler**-ului și plasarea corespunzătoare a apelurilor **FM_extract**. Sarcina **handler**-ului este să copieze mesajul sosit în zona al cărei pointer este găsit în cererea de recepție.

Handler-ul diferă în cazul unei operații sincrone, el având rolul suplimentar de a transmite o confirmare către emițător.

O abordare diferită este **MPI-CCL** (Collective Communication Library), care extrage avantaje din faptul că rețelele locale ce conectează stațiile de lucru sunt medii de comunicare cu *difuzare*. Ca urmare, în locul construcției comunicațiilor colective pe baza transferurilor punct la punct, care înrăutățește nejustificat performanțele operațiilor colective, această abordare optimizează operațiile colective pentru rețelele de stații de lucru, NOW (Network Of Workstations), care furnizează un serviciu nesigur de transport cu difuzare (Ethernet, ATM, etc.,).

Arhitectura sistemului include două niveluri, situate între nivelul legătură de date al rețelei locale și nivelul aplicațiilor MPI (vezi figura). Nivelul inferior, URTP (User-level Reliable Transport Protocol), oferă servicii de transport punct la punct și cu difuzare, sigure, nivelului superior. El realizează interfața cu nivelul legătură de date și convertește în servicii sigure funcționalitatea acestui nivel.

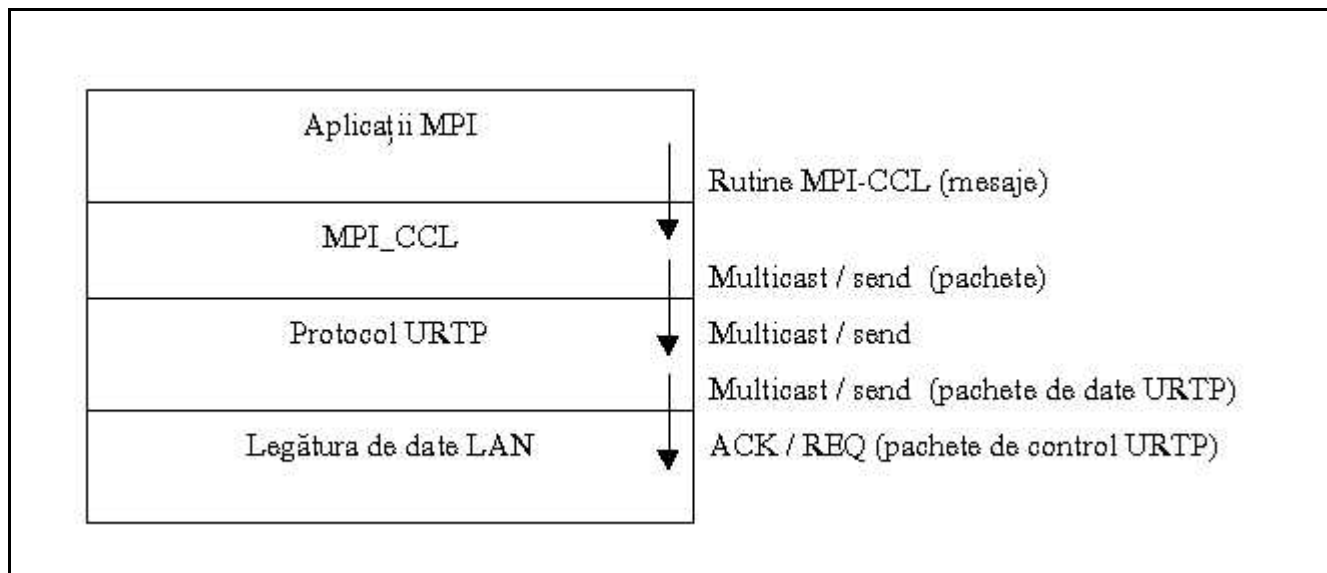


Figura 14

Nivelul superior, **MPI-CCL** oferă funcționalitatea operațiilor colective ale Mpi. Totodată, el face corespondența necesară, cu nivelul inferior, pentru operațiile punct la punct. În figură sunt precizate câteva noțiuni folosite în prezentare:

- Unitățile de date la nivelul aplicațiilor sunt **mesajele**, de lungime nelimitată; unitățile de date la nivelurile inferioare sunt **pachetele**, limitate superior la 1.5 Kocteți;
- Operația **multireceive** este definită ca recepția următorului pachet de la fiecare **procesor** dintr-o colecție specificată.

8.1.1. Programul global

Înainte de descrierea principalelor elemente relative la cele două niveluri, reluăm aici discuția relativă la corectitudinea programelor MPI, într-o formă mai detaliată.

Un program MPI poate fi văzut ca un singur **program global** executat pe mai multe procesoare SPMD (Single Program Multiple Data). Modelul garantează două proprietăți:

- Fiecare nod cunoaște ordinea pachetelor așteptate de la fiecare din celelalte noduri;
- Lipsa blocării datorată spațiului tampon pentru pachete.

Programele generate de MPI-CCL **se supun acestui model**. În speță, un astfel de program ar trebui să includă, la fiecare pas:

- Cel mult o operație **multicast** cu anumite procesoare țintă
- Câte o operație **receive** corespundătoare pentru fiecare procesor țintă

O variantă mai relaxată (denumită **k-buffer correct**) specifică următoarele condiții:

- Există o grupare de instrucțiuni ale programului, în care fiecare grup are cel mult k instrucțiuni;
- În fiecare grup, toate operațiile **multicast** corespund consistent și corect unor operații **receive**;
- Un **multicast** nu poate corespunde unui **receive** anterior.

Consecința: se poate obține o implementare corectă a unui program “k-buffer correct” folosind tamponare-sistem pentru k pachete per procesor.

De exemplu, pentru o operație all-to-all cu p procesoare, fiecare procesor poate genera un **multicast** urmat de $p-1$ apeluri **receive**, într-o ordine arbitrară. Pentru fiecare procesor sunt suficiente p tamponare pentru p pachete.

8.1.2. Protocolul URTP

URTP a fost proiectat pentru a furniza servicii sigure de transport, punct la punct și multicast, pornind de la ideea că, într-un LAN, pierderile de pachete se datorează în special lipsei spațiului de memorare la receptor. **Cerințele** avute în vedere sunt următoarele:

- Garantarea ordinii FIFO a pachetelor, indiferent dacă sunt punct la punct sau multicast
- Păstrarea pachetului la transmițător până la confirmarea recepției la toate procesoarele țintă
- Deoarece anumite pachete pot fi pierdute, se memorează la receptor toate pachetele de date, chiar dacă se detectează o pauză
- Detecția pierderilor de pachete se poate face când MPI-CCL face o cerere **receive** sau la violarea ordinii FIFO

URTP folosește o versiune modificată a protocolului cu **fereastră glisantă**. Fiecare pereche (sender, receiver) are asociat un contor. La transmiterea punct la punct a unui pachet, valoarea curentă a contorului este copiată într-un câmp al pachetului. Contorul este apoi incrementat. La un pachet multicast, se introduce un contor pentru fiecare pereche (sender, receiver_i). Contoarele sunt apoi incrementate.

La **transmisie**, MPI-CCL dă URTP **adresa** tamponului care conține un pachet și o funcție “call-back”. Pachetul este transmis, după care se așteaptă confirmările. Când toate sunt recepționate, URTP apelează funcția, al cărei rol este eliberarea tamponului. Numărul recomandat de tamponare gestionate de MPI-CCL este $(p-1)*w$, unde p este numărul de procesoare din configurație, iar w este lărgimea ferestrei.

La **recepție**, MPI-CCL primește, ca urmare a unei cereri de recepție, un **pointer** la pachetul primit și o funcție “call-back” de golire a tamponului. Zonele tampon sunt gestionate de URTP și sunt în număr de $(p-1)*w+c$, unde zona c este rezervată pentru pachete de control (ACK, REQ).

Un pachet este confirmat imediat ce ajunge într-un tampon URTP. Dacă o aplicație execută o operație **receive** și pachetul corespunzător nu este în tamponul URTP, acesta transmite un REQ către sursă. ACK și REQ sunt pachete punct la punct. De asemenea, unui pachet REQ i se răspunde cu un pachet de date punct la punct, chiar dacă pachetul original a fost multicast. Când REQ se repetă la un pachet încă netransmis, el este ignorat. În fine, pachetelor REQ le este asociat un mecanism time-out. Pachetele REQ acționează și cu rol de confirmare.

Protocolul este implementat printr-o combinație de rutine:

- Unele extind nucleul sistemului de operare

- Altele formează o bibliotecă URTP în spațiul utilizatorului

8.1.3. MPI-CCL

Scopul acestui nivel este să pună în corespondență rutinele MPI-CCL cu serviciile URTP, într-o manieră eficientă și să împartă mesajele în pachete (sau să refacă mesajele din pachete). Pentru interfața cu URTP se folosesc primitivele **multicast**, **send**, **receive** și **multireceive**. Ne referim în continuare la implementarea câtorva operații colective MPI.

- MPI_Bcast

Corespondența este directă. Dacă M este lungimea mesajului, m lungimea pachetelor URTP, $p = \left\lceil \frac{M}{m} \right\rceil$

numărul de pachete pe mesaj, pentru implementarea MPI_Bcast

- Rădăcina apelează de p ori **multicast**
- Toate celelalte procesoare apelează **receive** de p ori.
- MPI_Allgather

Există două variante. În prima, fiecare procesor devine pe rând sursă de difuzare, în timp ce restul proceselor primesc. Recepții succesive sunt combinate într-un **multireceive**.

```
i=mypid;
srcs1 = {0, 1, ..., i-1};
srcs2 = {i+1, i+2, ..., p-1}; /* p=numar procesoare */
distr = srcs1 ∪ srcs2;
for (j=0; j<q; j++) /* q=nr pachete/procesor, de difuzat */
{ multireceive (srcs1, recvbufs, j);
  multicast (dests, sendbuf, j);
  multireceive (srcs2, recvbufs, j);
}
```

În a doua variantă, fiecare procesor generează k apeluri multicast, urmate de k apeluri multireceive, unde k este un întreg pozitiv ales conform proprietății “kp-buffer correct”.

```
dests = {0, 1, ..., p-1} - {mypid};
j1=0; j2=0;
for (j=0; j<⌈ $\frac{q}{k}$ ⌉; j++)
{ for (l=0; l<k; l++)
  if (j1<q)
  { multicast (dests, sendbuf, i); j1++; }
  for (l=0; l<k; l++)
  if (j2<q)
  { multireceive (dests, recvbufs, j2); }
}
```

- MPI_Scatter și MPI_Gather

Pentru mesaje mici, MPI_scatter este implementată prin împachetarea lor într-un singur pachet, care este transmis tuturor proceselor. Pentru mesaje lungi se folosesc transmisii punct la punct. Ultimul pachet al tuturor destinațiilor poate fi transmis prin multicast, după modelul mesajelor scurte.

- MPI_Barrier

Este implementată ca un **MPI_Gather** cu un mesaj de lungime 0, de la fiecare procesor la procesorul 0, urmat de MPI_Bcast a unui mesaj de lungime 0 de la procesorul 0.