Laborator nr. 3
Introducere în MPI (2)

1 Tipuri de date în MPI

Trebuie observat faptul că, pentru fiecare comunicaţie în parte, lungimea mesajului este specificată în număr de elemente, nu în număr de octeţi. Tipurile de date de bază corespund tipurilor de date de bază ale limbajului folosit. Astfel avem:

Tabelul 1: Tipuri de date primare în MPI

MPI Datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

2 Comunicaţii colective

Conceptul cheie al **comunicaţiilor colective** este acela de a avea un **grup de procese participante**. Pentru aceasta biblioteca MPI oferă următoarele funcţii:

bariera – sincronizarea prin barieră pentru toţii membrii grupului (MPI_Barrier);

broadcast – transmiterea de la un membru al grupului către ceilalţi membri (MPI_Bcast);

gather – adunarea datelor de la toţi membrii grupului la un singur membru din grup (MPI_Gather);

scatter – împrăştierea datelor de la un membru la toţi membrii grupului (MPI_Scatter);

all gather – o variaţie a MPI_Gather în care toţi membrii grupului primesc rezultatele (MPI_Allgather);

all-to-all scatter/gather – adunarea/împraştierea datelor de la toţi membrii grupului către toţi membrii (MPI_Alltoall – o extensie a MPI_Allgather în care fiecare proces trimite date distincte către fiecare preces din grup);

operaţii de reducere – adunare, înmulţire, min, max, sau funcţii definite de utilizator în care rezultatele sunt trimise la toţi membrii grupului sau numai la un membru.

Observaţie: *Toate funcţiile sunt executate colectiv (sunt apelate de toate procesele dintr-un comunicator); toate procesele folosesc aceiaşi parametri în apeluri.*

Funcţiile nu au un identificator de grup ca argument explicit. În schimb avem un comunicator ca argument. Începând cu standardul MPI 2.0, comunicaţiile colective acceptă ca parametru un inter-comunicator.

Caracteristicile comunicaţiilor colective:

- Comunicaţiile colective şi cele de tip individuale **nu se vor influenţa reciproc**.
- Toate procesele trebuie să execute comunicaţiile colective.
- Sincronizarea nu este garantată, excepţie facând doar bariera.
- Nu există comunicaţii colective neblocante.
- Nu există tag-uri.
- Bufferele de recepţie trebuie sa fie exact de dimensiunea potrivită.

2.1 Bariera

Bariera este un mecanism de sincronizare a unui grup de procese. Dacă un proces a realizat un apel către `MPI_Barrier(com)`, unde `com` este comunicatorul, atunci el va aştepta până când toate procesele au efectuat acest apel.

Apelul funcţiei **Barrier**:

```
MPI_BARRIER(comm)
```

Prototipul funcţiei este:

```
1 int MPI_Barrier( MPI_Comm comm )
```

Listing 1: Prototipul funcţiei MPI_Barrier (C)

2.2 Broadcast

Funcţia MPI_Bcast trimite un mesaj de la procesul cu rangul *root* către toate procesele din grup, inclusiv procesului *root*. Schematic, acest comportament este prezentat în Figura 1.

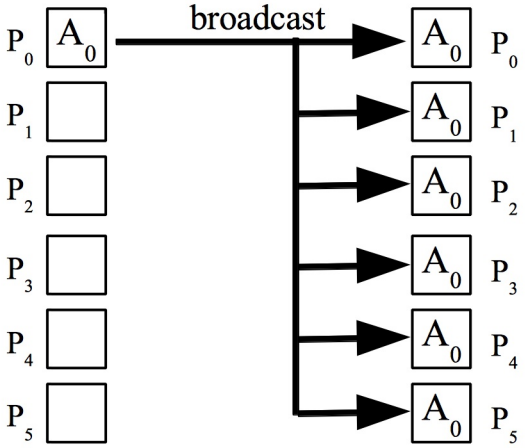

Figura 1: Operaţia colectivă *Broadcast*

Funcţia este apelată de toţi membrii grupului care folosesc acelaşi argument pentru comunicator şi, respectiv, acelaşi rang pentru procesul **root**. La ieşirea din funcţie, conţinutul buffer-ului de comunicaţie al procesului **root** a fost transferat către toate procesele.

Apelul funcţiei **Bcast** este:

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

unde:

INOUT buffer – adresa de început a buffer-ului pentru date

IN count – numărul de intrări din buffer

IN datatype – tipul datelor din buffer

IN root – rangul procesului **root** care iniţiază comunicaţia

IN comm – comunicatorul pe care se va realiza comunicaţia efectivă.

Prototipul funcţiei este detaliat în Listing 2. În continuare, în Listing 3 este evidenţiat un exemplu de utilizare.

```
1 int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype,
2                                         int root, MPI_Comm comm)
```

Listing 2: Prototipul funcţiei MPI_Bcast (C)

```
1  //.......
2  double param;
3  MPI_Init(&argc, &argv);
4  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
5  if(rank==5) {
6          param=23.0;
7  }
8  MPI_Bcast(&param,1,MPI_DOUBLE,5,MPI_COMM_WORLD);
9  printf("P:%d dupa broadcast parametrul este %f\n", rank, param);
10 MPI_Finalize();
11 //........
```

Listing 3: Utilizare MPI_Bcast (C)

2.3 Scatter

Procesul **root** împarte şi trimite conţinutul *sendbuffer*-ului către toate procesele din grup (inclusiv către procesul **root**). Schematic, acest lucru este reprezentat în Figura 2 şi în Figura 3.

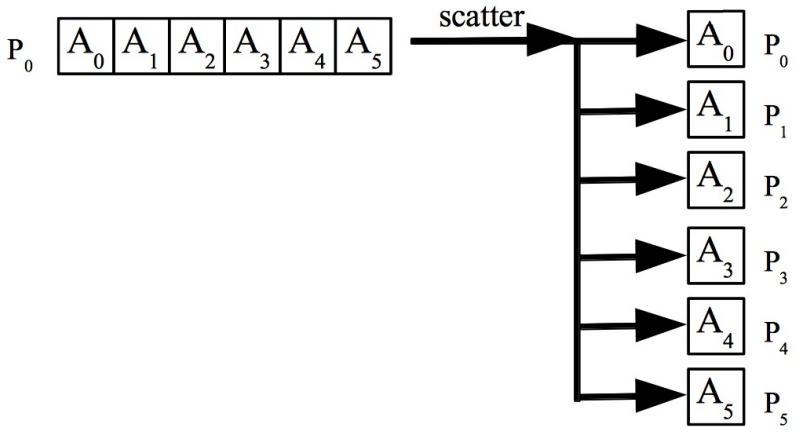

Figura 2: Operaţia colectivă *Scatter*

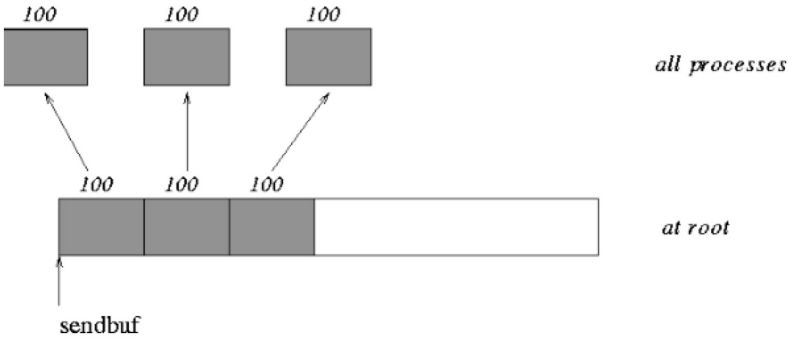

Figura 3: Comportamentul operaţiei colective *Scatter*

Acelaşi rezultat s-ar obţine dacă procesul **root** ar realiza `n` operaţii de tip *send* (unde `n` reprezintă numărul total de procese ce aparţin de grupul curent de lucru)

```
MPI_Send(sendbuf+i*sendcount*extent(sendtype), sendcount, sendtype, i,...)
```

şi fiecare proces execută o operaţie *receive*

```
MPI_Recv(recvbuf, recvcount, recvtype, i,...).
```

Apelul funcţiei **Scatter**:

```
MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
```

unde:

IN sendbuf – adresa de început a buffer-ului cu datele pentru trimitere

IN sendcount – numărul de intrări din buffer ce va fi trimis la fiecare proces

IN sendtype – tipul datelor din buffer

OUT recvbuf – adresa de început a bufferului de recepţie

IN recvcount – numărul de intrări ce vor fi primite de fiecare proces

IN recvtype – tipul datelor din bufferul de recepţie

IN root – rangul procesului ce trimite datele

IN comm – comunicatorul pe care se va realiza comunicaţia efectivă.

Prototipul funcţiei este expus în Listing 4.

```
1 MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,
2         void* recvbuf, int recvcount, MPI_Datatype recvtype,
3         int root, MPI_Comm comm)
```

Listing 4: Prototipul funcţiei MPI_Scatter (C)

Observaţii:

1. *Toate argumentele funcţiei sunt semnificative numai în procesul root, în timp ce pentru celelalte procese numai argumentele* `recvbuf`, `recvcount`, `recvtype`, `root` *şi* `comm` *sunt semnificative*.
2. `sendbuffer`, `sendcound` *şi* `sendtype` *sunt ignorate în procesele care nu au rangul root*.

În Listing 5 este prezentat un exemplu de utilizare a funcţiei MPI_Scatter.

```
1  //.......
2  MPI_Comm comm;
3  int gsize,*sendbuf;
4  int root, rbuf[100];
5  //...
6  MPI_Comm_size(comm, &gsize);
7  sendbuf = (int *)malloc(gsize*100*sizeof(int));
8  //...
9  MPI_Scatter(sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
10 //.......
```

Listing 5: Utilizare MPI_Scatter (C)

2.4 Gather

Fiecare proces (inclusiv procesul **root**) trimite conţinutul `sendbuffer`-ului către procesul root. Procesul **root** primeşte mesajele şi le **stochează în ordinea rangului**. Conceptual, acest lucru este exemplificat în Figurile 4 şi 5.

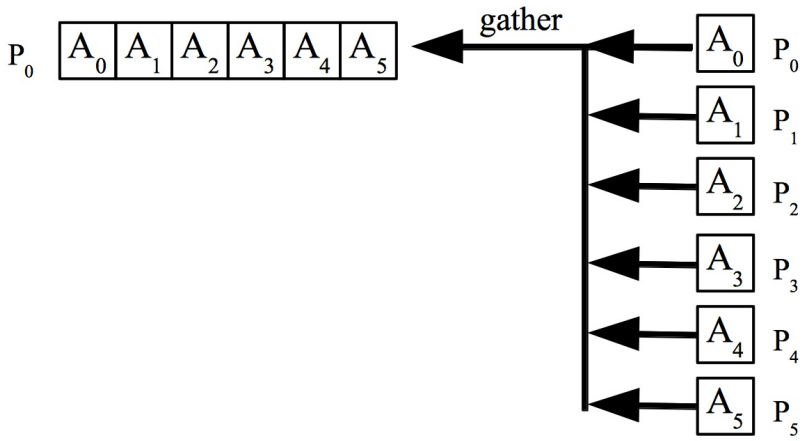

Figura 4: Operaţia colectivă *Gather*

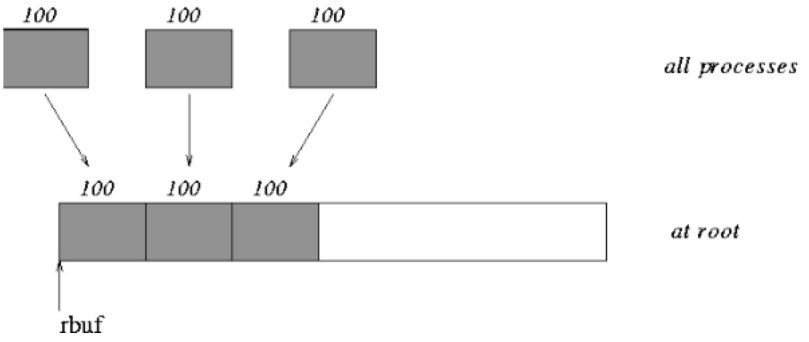

Figura 5: Comportamentul operaţiei colective *Gather*

Acelaşi rezultat s-ar obţine dacă toate procesele (inclusiv procesul **root**) ar realiza câte o operaţie de tip *send*

```
MPI_Send(sendbuf, sendcount, sendtype, root,...)
```

şi procesul **root** execută `n` operaţii de tip *receive* (unde `n` reprezintă numărul total de procese ce aparţin de grupul curent de lucru)

`MPI_Recv(recvbuf+i*recvcount*extent(recvtype), recvcount, recvtype, i ,...).`

Apelul funcţiei **Gather**:

```
MPI_GATHER( sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
```

unde:

IN sendbuf – adresa de început a zonei buffer-ului de date

IN sendcount – numărul de elemente din buffer

IN sendtype – tipul datelor din buffer

OUT recvbuf – adresa buffer-ului de recepţie (are semnificaţie numai pentru procesul root)

IN recvcount – numărul de elemente pentru fiecare recepţie în parte (are semnificaţie numai pentru procesul root)

IN recvtype – tipul datelor din buffer-ul de recepţie (are semnificaţie numai pentru procesul root)

IN root – rangul proceselor care recepţionează datele

IN comm – comunicatorul pe care se va realiza comunicaţia efectivă.

Prototipul funcţiei este următorul:

```
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
               void* recvbuf, int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm)
```

Listing 6: Prototipul funcţiei MPI_Gather (C)

Observaţii:

1. *Buffer-ul de recepţie este ignorat pentru procesele care nu au rangul root.*
2. *Argumentul* `recvcount` *la procesul root indică dimensiunea datelor primite de la fiecare proces şi nu numărul total de date primite.*

Listing-urile 7 şi 8 exemplifică utilizarea corectă a funcţiei Gather.

```
//.......
MPI_Comm comm;
int gsize,sendarray[100];
int root, *rbuf;
//...
MPI_Comm_size( comm, &gsize);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
//.......
```

Listing 7: Utilizare MPI_Gather (C) (1)

```
//.......
MPI_Comm comm;
int gsize,sendarray[100];
int root, myrank, *rbuf;
//...
MPI_Comm_rank( comm, &myrank);
if ( myrank == root) {
        MPI_Comm_size( comm, &gsize);
```

```
 9         rbuf = (int *)malloc(gsize*100*sizeof(int));
10 }
11 MPI_Gather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
12 //.......
```

Listing 8: Utilizare MPI_Gather (C) (2)

3 Topologii de comunicare în MPI

O topologie oferă un mecanism convenabil pentru denumirea proceselor dintr-un grup, şi în plus, poate oferi soluţii sistemului, în timpul rulării, pentru maparea proceselor pe hardware-ul existent. Un grup de procese în MPI reprezintă o colecţie de `n` procese. Fiecare proces din grup are atribuit un rang (valoare întreagă) între `0` şi `n-1`. În multe aplicaţii paralele o atribuire liniară a acestor ranguri nu reflectă logica modelului de comunicaţii necesar. Adesea rangurile proceselor sunt aranjate în topologii, precum o plasă cu 2 sau 3 dimensiuni. Mai general, logica aranjării proceselor este descrisă de un graf. În continuare, această aranjare logică a proceselor o vom numi: „topologie virtuală”.

Trebuie făcută distincţie între topologia virtuală a proceselor şi topologia fizică (hardware). Descrierea topologiilor virtuale depinde numai de aplicaţie şi este independentă de maşină. Structura comunicaţiilor pentru un set de procese poate reprezenta un graf, în care nodurile reprezintă procesele iar muchiile conectează procesele care comunică între ele.

MPI asigură transmiterea mesajelor între oricare dintre perechile de procese din grup. Nu este nevoie ca un canal de comunicaţie să fie deschis explicit. **De aceea, o muchie lipsă în graful de procese definit de utilizator nu asigură faptul că procesele care nu sunt conectate nu fac schimb mesaje**.

Informaţiile legate de topologie sunt asociate comunicatorilor. Astfel, următoarele informaţii legate de structura topologiei pot fi obţinute prin interogarea comunicatorului:

1. Tipul topologiei (carteziană, graf sau nedefinită)
2. Pentru o topologie carteziană:
   (a) ndims (numărul de dimensiuni),
   (b) dims (numărul de procese pe direcţia de coordonate),
   (c) periods (periodicitatea informaţiei),
   (d) own_position (poziţia din topologia creată)
3. Pentru o topologie de tip graf:
   (a) index,
   (b) muchii (reprezintă vectori ce definesc structura grafului).

Ca şi pentru alte apeluri de funcţii colective, **programul trebuie scris astfel încât să funcţioneze corect indiferent dacă apelurile se sincronizează sau nu**.

3.1 Topologii carteziene

Funcţia `MPI_CART_CREATE` poate fi folosită pentru a descrie structuri carteziene de dimensiune arbitrară. Pentru fiecare coordonată se poate specifica dacă structura este periodică sau nu. De exemplu, o topologie 1D, este liniară dacă nu este periodică şi este un inel dacă este periodică. Pentru o topologie 2D, avem un dreptunghi, un cilindru sau un tor după cum o dimensiune este sau nu periodică.

Observaţie: *Un* hipercub cu n dimensiuni *este* un tor cu n dimensiuni cu câte 2 procese pe fiecare coordonată. *De aceea nu este necesar un suport special pentru o structură de hipercub.*

Apelul funcţiei **MPI_Cart_create**:

```
MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)
```

unde:

IN comm_old – comunicatorul peste care va fi creată topologia nouă

IN ndims – numărul de dimensiuni ale topologiei carteziene

IN dims – tablou de întregi de dimensiune ndims care specifică numărul de procese de pe fiecare dimensiune

IN periods – tablou de dimensiune ndims care specifică dacă „grila” este periodică (true) sau nu (false) pe fiecare dimensiune

IN reorder – rangurile pot fi reordonate (true) sau nu (false)

OUT comm_cart – comunicator cu noua topologie carteziană.

Prototipul funcţiei este următorul:

```
1 int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,
2         int *periods, int reorder, MPI_Comm *comm_cart)
```

Listing 9: Prototipul funcţiei MPI_Cart_create (C)

Pentru topologiile carteziene, funcţia `MPI_DIMS_CREATE` ajută utilizatorul să realizeze o distribuire echilibrată a proceselor pe fiecare direcţie de coordonate. O posibilitate ar fi împărţirea tuturor proceselor într-o topologie n-dimensională.

Apelul funcţiei **MPI_Dims_create**:

```
MPI_DIMS_CREATE(nnodes, ndims, dims)
```

unde:

IN nnodes – numărul de noduri din grilă

IN ndims – numărul de dimensiuni ale topologiei

INOUT dims – tablou de întregi caare specifică numărul de noduri pe fiecare dimensiune

Prototipul funcţiei este următorul:

```
1 int MPI_Dims_create(int nnodes, int ndims, int *dims)
```

Listing 10: Prototipul funcţiei MPI_Dims_create (C)

Pentru fiecare `dims[i]` setat de apelul funcţiei `MPI_DIMS_CREATE`, `dims[i]` va fi ordonat descrescător. Tabloul `dims` este potrivit pentru a fi utilizat ca intrare în apelul funcţiei `MPI_CART_CREATE`. Un exemplu de rezultat oferit de funcţia `MPI_Dims_create` poate fi observat în Tabelul 2.

Tabelul 2: MPI_Dims_create – scenarii de utilizare

dims (înainte de apel)	parametrii de apel	dims (după apel)
(0, 0)	MPI_DIMS_CREATE(6, 2, dims)	(3, 2)
(0, 0)	MPI_DIMS_CREATE(7, 2, dims)	(7, 1)
(0, 3, 0)	MPI_DIMS_CREATE(6, 3, dims)	(2, 3, 1)
(0, 3, 0)	MPI_DIMS_CREATE(7, 3, dims)	apel eronat

Odată topologia stablită, aceasta poate fi interogată pentru a obţine diferite informaţii.

Aflarea dimensiunilor un topologii carteziene:

```
MPI_CARTDIM_GET(comm, ndims)
```

unde:

IN comm – comunicator cu structura cartezianã

OUT ndims – numărul de dimensiuni al structurii carteziene

Prototipul funcţiei este următorul:

```
1 int  MPI_Cartdim_get(MPI_Comm comm, int *ndims)
```

Listing 11: Prototipul funcţiei MPI_Cartdim_get (C)

Aflarea rank-ului unui proces mapat la un anumit set de coordonate:

```
MPI_CART_RANK (comm, coords, rank)
```

unde:

IN comm – comunicator cu structura cartezianã

IN coords – tablou de întregi care specifică coordonatele carteziene ale unui proces

OUT rank – ragul procesului specificat

Prototipul funcţiei este următorul:

```
1 int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
```

Listing 12: Prototipul funcţiei MPI_Cart_rank (C)

Pentru un grup de procese cu o structură cartezianã, funcţia `MPI_CART_RANK` transformă coordonatele logice ale procesului în rangul procesului.

Aflarea informaţiilor despre topologia cartezianã curentă:

```
MPI_CART_GET(comm, maxdims, dims, periods, coords)
```

unde:

IN comm – comunicator cu structura cartezianã

IN maxdims – lungimea vectorilor dims, periods, şi coords în program

OUT dims – numărul de procese pe fiecare dimesiune cartezianã

OUT periods – periodicitatea (true/ false) pentru fiecare dimesiune

OUT coords – coordonatele procesului care aplează funcţia in structura cartezianã

Prototipul funcţiei este următorul:

```
1 int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods,
2                  int *coords)
```

Listing 13: Prototipul funcţiei MPI_Cart_get (C)

Aflarea coordonatelor unui anumit proces:

```
MPI_CART_COORDS(comm, rank, maxdims, coords)
```

unde:

IN comm – comunicator cu structură carteziană

IN rank – rank-ul procesului pentru care se doreşte aflarea coordonatelor

IN maxdims – lungimea vectorului dims

OUT coords – vector de ieşire ce conţine coordonatele rank-ului indicat

Prototipul funţiei este următorul:

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)
```

Listing 14: Prototipul funcţiei MPI_Cart_coords (C)

Listingul 15 prezintă un model de utilizare a metodelor descrise anterior (crearea unei topologii carteziene, interogare coordonate rank, interogare rank la coordonate).

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

void main(int argc, char *argv[]) {
        int rank;
        MPI_Comm vu;
        int dim[2],period[2],reorder;
        int coord[2],id;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);
        dim[0]=4; dim[1]=3;
        period[0]=TRUE; period[1]=FALSE;
        reorder=TRUE;
        MPI_Cart_create(MPI_COMM_WORLD, 2, dim, period, reorder, &vu);

        if(rank==5){
                MPI_Cart_coords(vu, rank, 2, coord);
                printf("P:%d My coordinates are %d %d\n" rank, coord[0],
        }
        if(rank==0) {
                coord[0]=3; coord[1]=1;
                MPI_Cart_rank(vu, coord, &id);
                printf("The processor at position (%d, %d) has rank%d\n",
                                coord[0], coord[1], id);
        }

        MPI_Finalize();
        return 0;
}
```

Listing 15: Creare unui topologiie carteziene şi exemple de interogări (C)

3.2 Topologiile de tip graf

Crearea unei topologii de tip graf:

```
MPI_GRAPH_CREATE(comm_old, nnodes, index, edges, reorder, comm_graph)
```

unde:

IN comm_old – comunicator de intrare

IN nnodes – numărul de noduri din graf

IN index – tablou de întregi care descriu nodurile

IN edges – tablou de întregi care descriu muchiile grafului

IN reorder – rangurile pot fi ordonate sau nu

OUT comm_graph – comunicator cu topologia grafului

Prototipul funcţiei este următorul:

```
1 int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index,
2                      int *edges, int reorder, MPI_Comm *comm_graph)
```

Listing 16: Prototipul funcţiei MPI_Graph_create (C)

Pentru exemplificare, considerăm următorul graf:

Tabelul 3: Exemplu de graf

proces	vecini
0	1, 3
1	0
2	3
3	0, 2

În acest caz, argumentele `nnodes`, `index` şi `edges` corespunzătoare vor fi:

```
nnodes=4
index={2,3,4,6}
edges={1,3,0,3,0,2}
```

Pentru o structură de tip graf, numărul de noduri este egal cu numărul de procese din grup. De aceea numărul nodurilor nu trebuie memorat separat.

Funcţiile `MPI_GRAPHDIMS_GET` şi `MPI_GRAPH_GET` primesc informaţii privind topologia grafului care a fost asociat cu un comunicator de către funcţia `MPI_GRAPH_CREATE`. Informaţiile furnizate de `MPI_GRAPHDIMS_GET` pot fi utilizate pentru dimensionarea corectă a vectorilor `index` şi `edges` pentru a realiza un apel `MPI_GRAPH_GET`.

```
MPI_GRAPHDIMS_GET(comm, nnodes, nedges)
```

unde:

IN comm – comunicator cu grupul care are structura de graf

OUT nnodes – numărul de noduri din graf (acelaşi cu numărul proceselor din grup)

OUT nedges – numărul de muchii din graf

Prototipul funcţiei este următorul:

```
1 int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)
```

Listing 17: Prototipul funcţiei MPI_Graphdims_get (C)

```
MPI_GRAPH_GET(comm, nnodes, nedges)
```

unde:

IN comm – comunicator cu structura grafului

IN maxindex – lungimea vectorului index în programul apelant

IN maxedges – lungimera vectorului edges în programul apelant

OUT index – tablou de întregi ce conţine structura grafului

OUT edges – tablou de întregi ce conţine structura grafului

Prototipul funcţiei este următorul:

```
1 int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges,
2                   int *index, int *edges)
```

Listing 18: Prototipul funcţiei MPI_Graph_get (C)

Funcţiile `MPI_GRAPH_NEIGHBORS_COUNT` şi `MPI_GRAPH_NEIGHBORS` furnizează informaţii suplimentare pentru o topologie de tip graf.

```
MPI_GRAPH_NEIGHBORS_COUNT(comm, rank, nneighbors)
```

unde:

IN comm – comunicator cu topologia grafului

IN rank – rangul procesului din grupul comm

OUT nneighbors – numărul de vecini ai procesului specificat

Prototipul funcţiei este următorul:

```
1 int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)
```

Listing 19: Prototipul funcţiei MPI_Graph_neighbors_count (C)

```
MPI_GRAPH_NEIGHBORS(comm, rank, maxneighbors, neighbors)
```

unde:

IN comm – comunicator cu topologia grafului

IN rank – rangul procesului din grupul comm

IN maxneighbors – dimensiunea tabloului ce va stoca rangurile vecinilor

OUT neighbors – rangurile proceselor care au ca vecini procesul specificat

Prototipul funcţiei este următorul:

```
1 int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,
2                         int *neighbors)
```

Listing 20: Prototipul funcţiei MPI_Graph_neighbors (C)