Lectia 4 FUNCȚIILE MPI PENTRU 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 includ:
  - 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.

Caracteristici comune ale operațiilor colective de transmitere a mesajelor.

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.

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 de tip barieră, o operație colectivă nu poate fi folosită pentru sincronizarea proceselor.

Comunicator

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

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.

Principala diferență dintre operațiile colective și operațiile de tip punct la punct este faptul că acestea implică întotdeauna toate procesele asociate cu un mediu virtual de comunicare (comunicator). Setul de operații colective include:

- ✓ sincronizarea tuturor proceselor prin bariere (funcția **`MPI_Barrier`**);
- ✓ acțiuni de comunicare colectivă, care includ:
  - livrare de informații de la un proces la toți ceilalți membri ai comunicatorului (funcția **`MPI_Bcast`**);
  - o asamblare (adunare) a masivelor de date distribuite pe o serie de procese într-un singur tablou și păstrarea lui în spațiul de adrese a unui singur proces (**`root`**) (funcțiile **`MPI_Gather`**, **`MPI_Gatherv`**);
  - o asamblare (adunare) a masivelor de date distribuite pe o serie de procese într-un singur tablou și păstrarea lui în spațiul de adrese al tuturor proceselor comunicatorului (funcțiile **`MPI_Allgather`**, **`MPI_Allgatherv`**);
  - divizarea unui masiv și trimiterea fragmentelor sale tuturor proceselor din comunicatorul indicat (funcțiile **`MPI_Scatter`**, **`MPI_Scatterv`**);
  - o combinare a operațiilor **`Scatter/Gather(All-to-All`**), fiecare proces împarte datele sale din memoria tampon de trimitere și distribuie fragmentele de date tuturor proceselor, și,

concomitent, colectează fragmentele trimise într-un tampon de recepție (funcțiile **MPI_Alltoall**, **MPI_Alltoallv**);

- ✓ operațiile globale de calcul asupra datelor din memoria diferitor procese:
  - cu păstrarea rezultatelor în spațiul de adrese al unui singur proces (funcția **MPI_Reduce**);
  - cu difuzarea (distribuirea) rezultatelor tuturor proceselor comunicatorului indicat (funcția **MPI_Allreduce**);
  - operații combinate **Reduce/Scatter** (funcția **MPI_Reduce_scatter**);
  - operație de reducere prefix (funcția **MPI_Scan**).

Toate rutinele de comunicare, cu excepția **MPI_Bcast**, sunt disponibile în două versiuni:

• varianta simplă, când toate mesajele transmise au aceeași lungime și ocupă zonele adiacente în spațiul de adrese al procesului;

• varianta de tip "vector" (funcțiile cu simbol suplimentar "v" la sfârșitul numelui), care oferă mai multe oportunități pentru organizarea de comunicații colective și elimină restricțiile inerente din varianta simplă, atât în ceea ce privește lungimea blocurilor de date, cât și în ceea ce privește plasarea datelor în spațiul de adrese al proceselor.

Funcția MPI_Barrier

Realizează o sincronizare de tip barieră într-un grup de procese MPI; apelul se blochează până când toate procesele dintr-un grup ajung la barieră. Este singura funcție colectivă ce asigură sincronizare. Prototipul acestei funcții în limbajul C este

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

unde

IN **comm** – comunicatorul implicat.

Această funcţie, împreună cu funcţia **sleep,** poate fi utilizată pentru a sincroniza (a ordona) tiparul astfel: primul tipăreşte procesul cu rankul *0*, după care tipărește procesul cu rankul *1*, ş.a.m.d. Codul de program care realizează acest lucru este:

```
#include <mpi.h>
int main (int argc , char *argv[])
{
int rank,time;
MPI_Init (&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
time=2;
sleep(time*rank);
std::cout << "Synchronization point for:" << rank << std::endl ;
MPI_Barrier(MPI_COMM_WORLD) ;
std::cout << "After Synchronization, id:" << rank << std::endl ;
MPI_Finalize();
return 0;
}
```

Rezultatele posibile ale executării programului:

```
[Hancu_B_S@hpc Finale]$ /opt/openmpi/bin/mpiCC -o Sinhronizare.exe Sinhronizare.cpp
[Hancu_B_S@hpc Finale]$ /opt/openmpi/bin/mpirun -n 4 -host compute-0-0,compute-0-1
    Sinhronizare.exe
Synchronization point for:0
Synchronization point for:1
Synchronization point for:2
Synchronization point for:3
After Synchronization, id:2
After Synchronization, id:0
After Synchronization, id:3
```

```
After Synchronization, id:1
```

Funcția MPI_Bcast

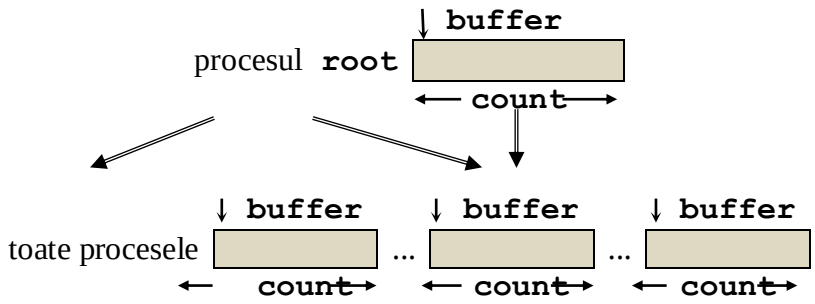
Trimite un mesaj de la procesul cu rankul „root” către toate celelalte procese ale comunicatorului. Prototipul acestei funcții în limbajul C este

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

unde parametrii sunt:

IN OUT **buf** – adresa inițială a tamponului destinatar;
IN **count** – numărul de elemente (întreg ne-negativ);
IN **datatype** – tipul fiecărui element;
IN **root** – numărul procesului care trimite datele;
IN **comm** – comunicatorul implicat.

Grafic această funcție poate fi interpretată astfel:



Vom exemplifica utilizarea funcției **MPI_Bcast** prin următorul exemplu.

Exemplu 4.1 *Să se realizeze transmiterea mesajelor pe cerc începând cu procesul „început”. Valoarea variabilei „început” este inițializată de procesul cu rankul 0.*

Mai jos este prezentat codul programului în limbajul C++ în care se realizează cele menționate mai sus.

```
#include<stdio.h>
#include <iostream>
#include<mpi.h>
using namespace std;
int main(int argc,char *argv[])
{
int size,rank,t,namelen,incep;
char processor_name[MPI_MAX_PROCESSOR_NAME];
 MPI_Status status;
 MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&size);
MPI_Comm_rank(MPI_COMM_WORLD,
      &rank);
MPI_Get_processor_name(processor_name,
      &namelen);
if (rank ==0)
printf("\n=====REZULTATUL PROGRAMULUI '%s' \n",argv[0]);
 while (true) {
   MPI_Barrier(MPI_COMM_WORLD);
   if (rank == 0) {
```

```
    cout << "Introduceti incep (de la 0 la " << size - 1 << ", sau numar negativ pentru a opri programul): ";
    cin >> incep;
   }
   MPI_Bcast(&incep, 1, MPI_INT, 0, MPI_COMM_WORLD);
   if (incep < 0) {
    MPI_Finalize();
    return 0;
   }
   if (size <= incep) {
    if (rank == 0) {
     cout << "Incep trebuie sa fie mai mic decit nr de procesoare (" << size - 1 << ").\n";
    }
    continue;
   }
   if(rank==incep){
   MPI_Send(&rank,1,MPI_INT, (rank + 1) % size, 10, MPI_COMM_WORLD);
   MPI_Recv(&t,1,MPI_INT, (rank+size-1) % size,10,MPI_COMM_WORLD,&status);
   }
   else{
  MPI_Recv(&t,1,MPI_INT, (rank+size-1)%size, 10,MPI_COMM_WORLD,&status);
MPI_Send(&rank,1,MPI_INT,(rank+1)%size,10,MPI_COMM_WORLD);
   }
printf("Procesul cu rancul %d al nodului '%s' a primit valoarea %d de la procesul cu rancul %d\n",rank,
      processor_name, t, t);
 }
 MPI_Finalize();
 return 0; }
```

Rezultatele posibile ale executării programului:

```
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpiCC -o Exemplu_3_4_1.exe Exemplu_3_4_1.cpp[1]
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpirun -n 9 -machinefile ~/nodes Exemplu_3_4_1.exe

=====REZULTATUL PROGRAMULUI 'Exemplu_3_4_1.exe'
Introduceti incep (de la 0 la 8, sau numar negativ pentru a opri programul): 44
Incep trebuie sa fie mai mic decit nr de procesoare (8).
Introduceti incep (de la 0 la 8, sau numar negativ pentru a opri programul): 8
=====REZULTATUL PROGRAMULUI 'HB_Mesage_Ring_Nproc_V0.exe'
Procesul cu rancul 0 al nodului 'compute-0-0.local' a primit valoarea 8 de la procesul cu rancul 8
Procesul cu rancul 1 al nodului 'compute-0-0.local' a primit valoarea 0 de la procesul cu rancul 0
Procesul cu rancul 2 al nodului 'compute-0-0.local' a primit valoarea 1 de la procesul cu rancul 1
Procesul cu rancul 3 al nodului 'compute-0-0.local' a primit valoarea 2 de la procesul cu rancul 2
Procesul cu rancul 4 al nodului 'compute-0-1.local' a primit valoarea 3 de la procesul cu rancul 3
Procesul cu rancul 8 al nodului 'compute-0-2.local' a primit valoarea 7 de la procesul cu rancul 7
Procesul cu rancul 5 al nodului 'compute-0-1.local' a primit valoarea 4 de la procesul cu rancul 4
Procesul cu rancul 6 al nodului 'compute-0-1.local' a primit valoarea 5 de la procesul cu rancul 5
Procesul cu rancul 7 al nodului 'compute-0-1.local' a primit valoarea 6 de la procesul cu rancul 6
```

[1] Numele programului corespunde numelui exemplului din notele de curs Boris HÎNCU, Elena CALMÎŞ "MODELE DE PROGRAMARE PARALELĂ PE CLUSTERE. PARTEA I. PROGRAMARE MPI". Chisinau 2016.

[Hancu_B_S@hpc]$

4.1 Funcțiile MPI pentru asamblarea (adunarea) datelor

Funcția MPI_ Gather

Asamblează (adună) mesaje distincte de la fiecare proces din grup într-un singur proces destinație. Asamblarea se face în ordinea creșterii rankului procesorului care trimite datele. Adică datele trimise de procesul *i* din tamponul său sendbuf se plasează în porțiunea *i* din tamponul recvbuf al procesului root.

Prototipul acestei funcții în limbajul C este

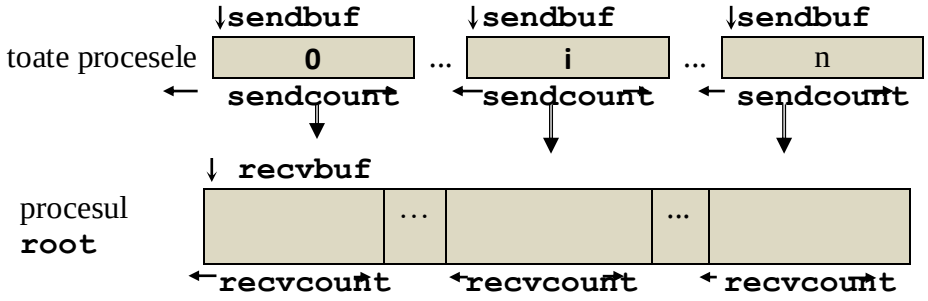
```
int MPI_Gather(void* sendbuf,int sendcount, MPI_Datatype
    sendtype,void* recvbuf,int recvcount,MPI_Datatype recvtype,int
    root, MPI_Comm comm),
```

unde

IN **sendbuf**	– adresa inițială a tamponului pentru datele trimise;
IN **sendcount**	– numărul de elemente trimise (întreg ne-negativ);
IN **sendtype**	– tipul fiecărui element trimis;
out **recvbuf**	– adresa inițială a tamponului pentru datele recepționate (este utilizat numai de procesul **root**);
IN **recvcount**	– numărul de elemente recepționate de la fiecare proces (este utilizat numai de procesul **root**);
IN **recvtype**	– tipul fiecărui element primit (este utilizat numai de procesul **root**);
IN **root**	– numărul procesului care recepționează datele;
IN **comm**	– comunicatorul implicat.

Tipul fiecărui element trimis **sendtype** trebuie să coincidă cu tipul fiecărui element primit **recvtype** și numărul de elemente trimise **sendcount** trebuie să fie egal cu numărul de elemente recepționate **recvcount.**

Grafic această funcție poate fi interpretată astfel:



Funcția MPI_Allgather

Această funcție se execută la fel ca funcția **MPI_Gather**, dar destinatarii sunt toate procesele grupului. Datele trimise de procesul *i* din tamponul său sendbuf se plasează în porțiunea *i* din tamponul fiecărui proces. Prototipul acestei funcții în limbajul C este

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

unde

IN **sendbuf**	– adresa inițială a tamponului pentru datele trimise;
IN **sendcount**	– numărul de elemente trimise (întreg ne-negativ);
IN **sendtype**	– tipul fiecărui element trimis;
OUT **recvbuf**	– adresa inițială a tamponului pentru datele recepționate;
IN **recvcount**	– numărul de elemente recepționate de la fiecare proces;
IN **recvtype**	– tipul fiecărui element primit;
IN **comm**	– comunicatorul implicat.

Funcția MPI_Gatherv

Se colectează blocuri cu un număr diferit de elemente pentru fiecare proces, deoarece numărul de elemente preluate de la fiecare proces este setat individual prin intermediul vectorului **recvcounts**. Această funcție oferă o mai mare flexibilitate în plasarea datelor în memoria procesului destinatar, prin introducerea unui tablou **displs.** Prototipul acestei funcții în limbajul C este

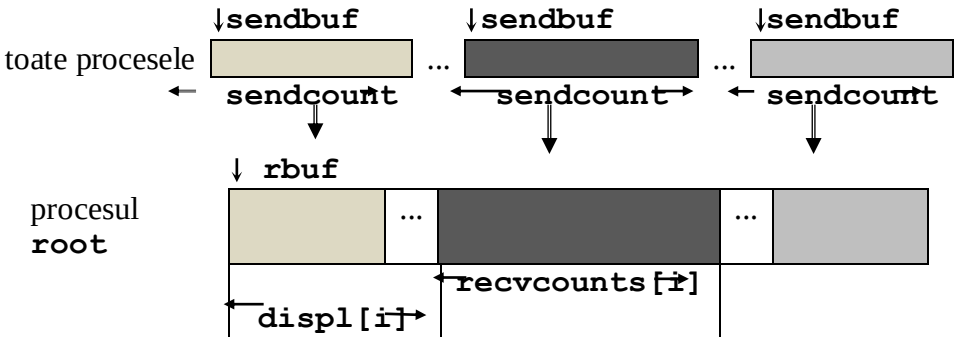
```
int MPI_Gatherv(void* sendbuf,int sendcount, MPI_Datatype
    sendtype,void* rbuf,int *recvcounts,int *displs,MPI_Datatype
    recvtype,int root,MPI_Comm comm)
```

unde

IN **sendbuf**	– adresa inițială a tamponului pentru datele trimise;
IN **sendcount**	– numărul de elemente trimise (întreg ne-negativ);
IN **sendtype**	– tipul fiecărui element trimis;
OUT **rbuf**	– adresa inițială a tamponului pentru datele recepționate (este utilizat numai de procesul **root**);
IN **recvcounts**	– un vector cu numere întregi (dimensiunea este egală cu numărul de procese din grup), al cărui element *i* determină numărul de elemente care sunt recepționate de la procesul cu *i* (este utilizat numai de procesul **root**);
IN **displs**	– un vector cu numere întregi (dimensiunea este egală cu numărul de procese din grup), al cărui element *i* determină deplasarea blocului *i* de date față de **rbuf** (este utilizat numai de procesul **root**);
IN **recvtype**	– tipul fiecărui element primit (este utilizat numai de procesul **root**);
IN **root**	– numărul procesului care recepționează datele;
IN **comm**	– comunicatorul implicat.

Mesajele sunt plasate în memoria tampon al procesului **root,** în conformitate cu rankurile proceselor care trimit datele, și anume datele transmise de procesul *i*, sunt plasate în spațiul de adrese al procesului **root**, începând cu adresa **rbuf +displs [i]**.

Grafic această funcție poate fi interpretată astfel:



Funcția MPI_ MPI_Allgatherv

Această funcție este similară funcției **MPI_Gatherv**, culegerea datelor se face de toate procesele din grup. Prototipul acestei funcții în limbajul C este

```
int MPI_Allgatherv(void* sendbuf,int sendcount, MPI_Datatype
    sendtype,void* rbuf, int *recvcounts, int *displs, MPI_Datatype
    recvtype, MPI_Comm comm)
```

unde

IN **sendbuf**	– adresa inițială a tamponului pentru datele trimise;
IN **sendcount**	– numărul de elemente trimise (întreg ne-negativ);
IN **sendtype**	– tipul fiecărui element trimis;
OUT **rbuf**	– adresa inițială a tamponului pentru datele recepționate;
IN **recvcounts**	– un vector cu numere întregi (dimensiunea este egală cu numărul de procese din grup), al cărui element *i* determină numărul de elemente care sunt recepționate de la procesul cu *i*;
IN **displs**	– un vector cu numere întregi (dimensiunea este egală cu numărul de procese din grup), al cărui element *i* determină deplasarea blocului *i* de date față de **rbuf**;
IN **recvtype**	– tipul fiecărui element primit;
IN **comm**	– comunicatorul implicat.

4.2 Funcțiile MPI pentru difuzarea (distribuirea) datelor

Funcțiile MPI prin intermediul cărora se poate realiza distribuirea colectivă a datelor tuturor proceselor din grup sunt: **MPI_Scatter** și **MPI_Scaterv**.

Funcția MPI_Scatter

Funcția MPI_Scatter împarte mesajul care se află pe adresa variabilei **sendbuf** a procesului cu rankul **root** în părți egale de dimensiunea **sendcount** și trimite partea *i* pe adresa variabilei **recvbuf** a procesului cu rankul *i* (inclusiv sie). Procesul **root** utilizează atât tamponul **sendbuf** cât și **recvbuf,** celelalte procese ale comunicatorului **comm** sunt doar beneficiari, astfel încât parametrii care încep cu „**send**" nu sunt esențiali. Prototipul acestei funcții în limbajul C este

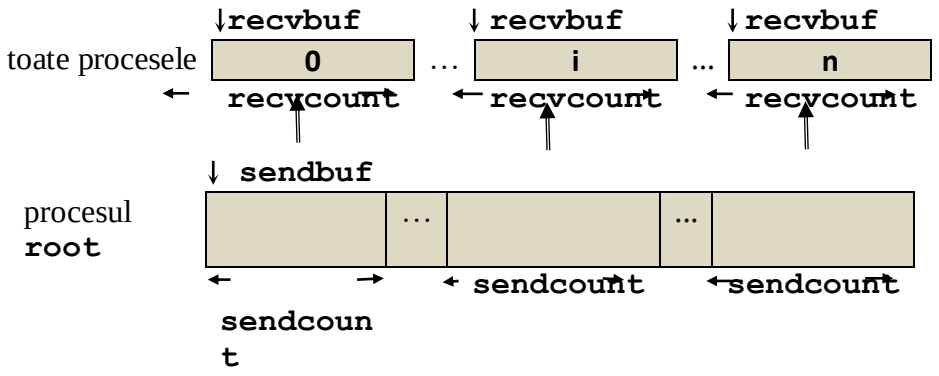
```
int MPI_Scatter(void* sendbuf,int sendcount, MPI_Datatype
    sendtype,void* recvbuf,int recvcount,MPI_Datatype recvtype,int
    root, MPI_Comm comm),
```

unde

IN **sendbuf**	– adresa inițială a tamponului pentru datele trimise (distribuite) (este utilizat numai de procesul **root**);
IN **sendcount**	– numărul de elemente trimise (întreg ne-negativ) fiecărui proces;
IN **sendtype**	– tipul fiecărui element trimis;
OUT **recvbuf**	– adresa inițială a tamponului pentru datele recepționate;
IN **recvcount**	– numărul de elemente recepționate de la fiecare proces;
IN **recvtype**	– tipul fiecărui element recepționat de fiecare proces;
IN **root**	– numărul procesului care distribuie datele;
IN **comm**	– comunicatorul implicat.

Tipul fiecărui element trimis **sendtype** trebuie să coincidă cu tipul fiecărui element primit **recvtype** și numărul de elemente trimise **sendcount** trebuie să fie egal cu numărul de elemente recepționate **recvcount.**

Grafic această funcție poate fi interpretată astfel:



Funcția MPI_Scaterv

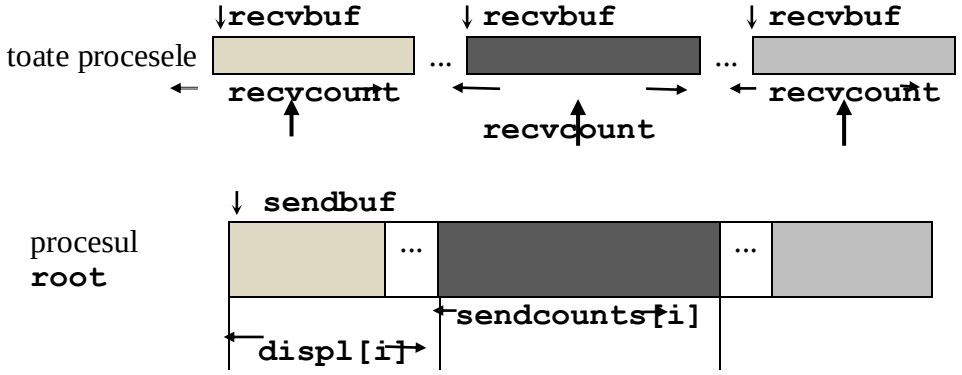
Această funcție este o versiune vectorială a funcției **MPI_Scatter** și permite distribuirea pentru fiecare proces a unui număr diferit de elemente. Adresa de start a elementelor transmise procesului cu rankul *i* este indicat în vectorul **displs,** și numărul de elemente trimise se indică în vectorul **sendcounts.** Această funcție este inversa funcției **MPI_Gatherv.** Prototipul acestei funcții în limbajul C este

```
int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,
    MPI_Datatype sendtype, void* recvbuf, int recvcount,
    MPI_Datatype recvtype, int root, MPI_Comm comm)
```

unde

- IN **sendbuf** – adresa inițială a tamponului pentru datele trimise (este utilizat numai de procesul **root**);
- IN **sendcounts** – un vector cu numere întregi (dimensiunea este egală cu numărul de procese din grup), al cărui element *i* indică numărul de elemente trimise procesului cu rankul *i*;
- IN **displs** – un vector cu numere întregi (dimensiunea este egală cu numărul de procese din grup), al cărui element *i* determină deplasarea blocului *i* de date față de **sendbuf**;
- IN **sendtype** – tipul fiecărui element trimis;
- OUT **recvbuf** – adresa inițială a tamponului pentru datele recepționate;
- IN **recvcount** – numărul de elemente care sunt recepționate;
- IN **recvtype** – tipul fiecărui element primit.
- IN **root** – numărul procesului care distribuie datele
- IN **comm** – comunicatorul implicat

Grafic această funcție poate fi interpretată astfel:



Vom ilustra utilizarea funcțiilor **MPI_Scatter**, **MPI_Gather** prin următorul exemplu.

Exemplul 4.2 *Să se distribuie liniile unei matrice pătrate (dimensiunea este egală cu numărul de procese) proceselor din comunicatorul* ***MPI_COMM_WORLD****. Elementele matricei sunt inițializate de procesul cu rankul 0.*

Mai jos este prezentat codul programului în limbajul C++ în care se realizează cele menționate mai sus.

```
#include<mpi.h>
#include<stdio.h>
int main(int argc, char *argv[])
{
int numtask,sendcount,reccount,source;
double *A_Init,*A_Fin;
```

```
int i, myrank, root=0;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &numtask);
double Rows[numtask];
sendcount=numtask;
reccount=numtask;
if (myrank ==0)
printf("\n=====REZULTATUL PROGRAMULUI '%s' \n",argv[0]);
MPI_Barrier(MPI_COMM_WORLD);
//Procesul cu rankul root aloca spatiul și initializeaza matrice
if(myrank==root)
{
A_Init=(double*)malloc(numtask*
      numtask*sizeof(double));
A_Fin=(double*)malloc(numtask*
      numtask*sizeof(double));
for(int i=0;i<numtask*numtask;i++)
A_Init[i]=rand()/1000000000.0;
printf("Tipar datele initiale\n");
for(int i=0;i<numtask;i++)
{
printf("\n");
for(int j=0;j<numtask;j++)
printf("A_Init[%d,%d]=%5.2f ",i,j, A_Init[i*numtask+j]);
}
printf("\n");
MPI_Barrier(MPI_COMM_WORLD);
}
else MPI_Barrier(MPI_COMM_WORLD);
MPI_Scatter(A_Init, sendcount, MPI_DOUBLE,Rows, reccount, MPI_DOUBLE, root,
      MPI_COMM_WORLD);
printf("\n");
printf("Resultatele f-tiei MPI_Scatter pentru procesul cu rankul %d \n", myrank);
for (i=0; i<numtask; ++i)
printf("Rows[%d]=%5.2f ",i, Rows[i]);
printf("\n");
MPI_Barrier(MPI_COMM_WORLD);
MPI_Gather(Rows, sendcount, MPI_DOUBLE, A_Fin, reccount, MPI_DOUBLE, root,
      MPI_COMM_WORLD);
if(myrank==root){
printf("\n");
printf("Resultatele f-tiei MPI_Gather ");
for(int i=0;i<numtask;i++)
{
printf("\n");
for(int j=0;j<numtask;j++)
printf("A_Fin[%d,%d]=%5.2f ",i,j, A_Fin[i*numtask+j]);
}
```

```
printf("\n");
MPI_Barrier(MPI_COMM_WORLD);
}
else MPI_Barrier(MPI_COMM_WORLD);
 MPI_Finalize();
return 0;
}
```

Rezultatele posibile ale executării programului:

```
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpiCC -o Exemplu_3_4_2.exe Exemplu_3_4_2.cpp
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpirun -n 5  -machinefile ~/nodes Exemplu_3_4_2.exe
=====REZULTATUL PROGRAMULUI ' Exemplu_3_4_2.exe'
Tipar datele initiale
A_Init[0,0]= 1.80 A_Init[0,1]= 0.85 A_Init[0,2]= 1.68 A_Init[0,3]= 1.71 A_Init[0,4]= 1.96
A_Init[1,0]= 0.42 A_Init[1,1]= 0.72 A_Init[1,2]= 1.65 A_Init[1,3]= 0.60 A_Init[1,4]= 1.19
A_Init[2,0]= 1.03 A_Init[2,1]= 1.35 A_Init[2,2]= 0.78 A_Init[2,3]= 1.10 A_Init[2,4]= 2.04
A_Init[3,0]= 1.97 A_Init[3,1]= 1.37 A_Init[3,2]= 1.54 A_Init[3,3]= 0.30 A_Init[3,4]= 1.30
A_Init[4,0]= 0.04 A_Init[4,1]= 0.52 A_Init[4,2]= 0.29 A_Init[4,3]= 1.73 A_Init[4,4]= 0.34
Resultatele f-tiei MPI_Scatter pentru procesul cu rankul 0
Rows[0]= 1.80 Rows[1]= 0.85 Rows[2]= 1.68 Rows[3]= 1.71 Rows[4]= 1.96
Resultatele f-tiei MPI_Scatter pentru procesul cu rankul 1
Resultatele f-tiei MPI_Scatter pentru procesul cu rankul 4
Resultatele f-tiei MPI_Scatter pentru procesul cu rankul 3
Rows[0]= 0.04 Rows[1]= 0.52 Rows[2]= 0.29 Rows[3]= 1.73 Rows[4]= 0.34
Rows[0]= 0.42 Rows[1]= 0.72 Rows[2]= 1.65 Rows[3]= 0.60 Rows[4]= 1.19
Rows[0]= 1.97 Rows[1]= 1.37 Rows[2]= 1.54 Rows[3]= 0.30 Rows[4]= 1.30
Resultatele f-tiei MPI_Scatter pentru procesul cu rankul 2
Rows[0]= 1.03 Rows[1]= 1.35 Rows[2]= 0.78 Rows[3]= 1.10 Rows[4]= 2.04
Resultatele f-tiei MPI_Gather
A_Fin[0,0]= 1.80 A_Fin[0,1]= 0.85 A_Fin[0,2]= 1.68 A_Fin[0,3]= 1.71 A_Fin[0,4]= 1.96
A_Fin[1,0]= 0.42 A_Fin[1,1]= 0.72 A_Fin[1,2]= 1.65 A_Fin[1,3]= 0.60 A_Fin[1,4]= 1.19
A_Fin[2,0]= 1.03 A_Fin[2,1]= 1.35 A_Fin[2,2]= 0.78 A_Fin[2,3]= 1.10 A_Fin[2,4]= 2.04
A_Fin[3,0]= 1.97 A_Fin[3,1]= 1.37 A_Fin[3,2]= 1.54 A_Fin[3,3]= 0.30 A_Fin[3,4]= 1.30
A_Fin[4,0]= 0.04 A_Fin[4,1]= 0.52 A_Fin[4,2]= 0.29 A_Fin[4,3]= 1.73 A_Fin[4,4]= 0.34
[Hancu_B_S@hpc Finale]$
```

Vom ilustra utilizarea funcțiilor **`MPI_Scatterv, MPI_Gatherv`** prin următorul exemplu.

Exemplul 4.3 *Să se distribuie liniile unei matrice de dimensiuni arbitrare proceselor din comunicatorul MPI_COMM_WORLD. Elementele matricei sunt inițializate de procesul cu rankul 0. Fiecare proces MPI recepționează cel puțin* $l_{min} = \left\lfloor \frac{n}{size} \right\rfloor$ *linii, unde* n *– numărul de linii în matricea inițială, size – numărul de procese generate,* $\lfloor \cdot \rfloor$ *– partea întreagă.*

Mai jos este prezentat codul programului în limbajul C++ în care se realizează cele menționate în enunțul exemplului 4.3.

```
#include<mpi.h>
#include<stdio.h>
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
```

```
{
int size,reccount, source;
double *A_Init,*A_Fin;
int myrank, root=0;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
int sendcounts[size], displs[size];
int rows, cols;
double *Rows;
if (myrank ==0)
printf("\n=====REZULTATUL PROGRAMULUI '%s' \n",argv[0]);
MPI_Barrier(MPI_COMM_WORLD);
if(myrank==root)
{
cout << "Introduceti numarul de rinduri:  ";
cin >> rows;
cout << "Introduceti numarul de coloane:  ";
cin >> cols;
A_Init=(double*)malloc(rows*cols*sizeof
      (double));
A_Fin=(double*)malloc(rows*cols*sizeof
       (double));
for(int i=0;i<rows*cols;i++)
A_Init[i]=rand()/1000000000.0;
printf("Matricea initiala\n");
for(int i=0;i<rows;i++)
{
printf("\n");
for(int j=0;j<cols;j++)
printf("A_Init[%d,%d]=%5.2f ",i,j, A_Init[i * cols + j]);
}
printf("\n");
MPI_Barrier(MPI_COMM_WORLD);
}
else
MPI_Barrier(MPI_COMM_WORLD);
MPI_Bcast(&rows, 1, MPI_INT, root, MPI_COMM_WORLD);
MPI_Bcast(&cols, 1, MPI_INT, root, MPI_COMM_WORLD);
int rinduriPeProces = rows / size;
int rinduriRamase = rows % size;
int deplasarea = 0;
for (int i = 0; i < size; ++i)
{
displs[i] = deplasarea;
if (i < rinduriRamase)
sendcounts[i] = (rinduriPeProces + 1) * cols;
else
sendcounts[i] = rinduriPeProces * cols;
```

```
deplasarea += sendcounts[i];
}
reccount = sendcounts[myrank];
Rows = new double[reccount];
MPI_Scatterv(A_Init, sendcounts, displs, MPI_DOUBLE, Rows, reccount, MPI_DOUBLE, root,
      MPI_COMM_WORLD);
printf("\n");
printf("Rezultatele f-tiei MPI_Scatterv pentru procesul cu rankul %d \n", myrank);
for (int i=0; i<reccount; ++i)
printf(" Rows[%d]=%5.2f ", i, Rows[i]);
printf("\n");
cout << "\nProcesul " << myrank << " a primit " << reccount << " elemente (" << reccount / cols << "
      linii)" << endl;
MPI_Barrier(MPI_COMM_WORLD);
int sendcount = reccount;
int *recvcounts = sendcounts;
MPI_Gatherv(Rows, sendcount, MPI_DOUBLE,  A_Fin, recvcounts, displs, MPI_DOUBLE, root,
      MPI_COMM_WORLD);
if(myrank==root)
{
printf("\n");
printf("Resultatele f-tiei MPI_Gatherv ");
for(int i=0;i<rows;i++)
   {
      printf("\n");
      for(int j=0;j<cols;j++)
      printf("A_Fin[%d,%d]=%5.2f ",i,j, A_Fin[i*cols+j]);
   }
      printf("\n");
      MPI_Barrier(MPI_COMM_WORLD);
free(A_Init);
free(A_Fin);
   }
   else
MPI_Barrier(MPI_COMM_WORLD);

MPI_Finalize();
delete[] Rows;
return 0;
}
```

Rezultatele posibile ale executării programului:

```
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpiCC -o Exemplu_3_4_3.exe Exemplu_3_4_3.cpp
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpirun -n 2  -machinefile ~/nodes Exemplu_3_4_3.exe
=====REZULTATUL PROGRAMULUI ‘Exemplu_3_4_3.exe'
Introduceti numarul de rinduri:  5
Introduceti numarul de coloane:  6
Matricea initiala
```

```
A_Init[0,0]= 1.80 A_Init[0,1]= 0.85 A_Init[0,2]= 1.68 A_Init[0,3]= 1.71 A_Init[0,4]= 1.96 A_Init[0,5]= 0.42
A_Init[1,0]= 0.72 A_Init[1,1]= 1.65 A_Init[1,2]= 0.60 A_Init[1,3]= 1.19 A_Init[1,4]= 1.03 A_Init[1,5]= 1.35
A_Init[2,0]= 0.78 A_Init[2,1]= 1.10 A_Init[2,2]= 2.04 A_Init[2,3]= 1.97 A_Init[2,4]= 1.37 A_Init[2,5]= 1.54
A_Init[3,0]= 0.30 A_Init[3,1]= 1.30 A_Init[3,2]= 0.04 A_Init[3,3]= 0.52 A_Init[3,4]= 0.29 A_Init[3,5]= 1.73
A_Init[4,0]= 0.34 A_Init[4,1]= 0.86 A_Init[4,2]= 0.28 A_Init[4,3]= 0.23 A_Init[4,4]= 2.15 A_Init[4,5]= 0.47
Rezultatele f-tiei MPI_Scatterv pentru procesul cu rankul 0
Rows[0]= 1.80  Rows[1]= 0.85  Rows[2]= 1.68  Rows[3]= 1.71  Rows[4]= 1.96  Rows[5]= 0.42  Rows[6]=
0.72  Rows[7]= 1.65  Rows[8]= 0.60  Rows[9]= 1.19  Rows[10]= 1.03  Rows[11]= 1.35  Rows[12]= 0.78
Rows[13]= 1.10  Rows[14]= 2.04  Rows[15]= 1.97  Rows[16]= 1.37  Rows[17]= 1.54
Rezultatele f-tiei MPI_Scatterv pentru procesul cu rankul 1
Rows[0]= 0.30  Rows[1]= 1.30  Rows[2]= 0.04  Rows[3]= 0.52  Rows[4]= 0.29  Rows[5]= 1.73  Rows[6]=
0.34  Rows[7]= 0.86  Rows[8]= 0.28  Rows[9]= 0.23  Rows[10]= 2.15  Rows[11]= 0.47
Procesul 0 a primit 18 elemente (3 linii)
Procesul 1 a primit 12 elemente (2 linii)
Resultatele f-tiei MPI_Gatherv
A_Fin[0,0]= 1.80 A_Fin[0,1]= 0.85 A_Fin[0,2]= 1.68 A_Fin[0,3]= 1.71 A_Fin[0,4]= 1.96 A_Fin[0,5]= 0.42
A_Fin[1,0]= 0.72 A_Fin[1,1]= 1.65 A_Fin[1,2]= 0.60 A_Fin[1,3]= 1.19 A_Fin[1,4]= 1.03 A_Fin[1,5]= 1.35
A_Fin[2,0]= 0.78 A_Fin[2,1]= 1.10 A_Fin[2,2]= 2.04 A_Fin[2,3]= 1.97 A_Fin[2,4]= 1.37 A_Fin[2,5]= 1.54
A_Fin[3,0]= 0.30 A_Fin[3,1]= 1.30 A_Fin[3,2]= 0.04 A_Fin[3,3]= 0.52 A_Fin[3,4]= 0.29 A_Fin[3,5]= 1.73
A_Fin[4,0]= 0.34 A_Fin[4,1]= 0.86 A_Fin[4,2]= 0.28 A_Fin[4,3]= 0.23 A_Fin[4,4]= 2.15 A_Fin[4,5]= 0.47
[Hancu_B_S@hpc]$
```