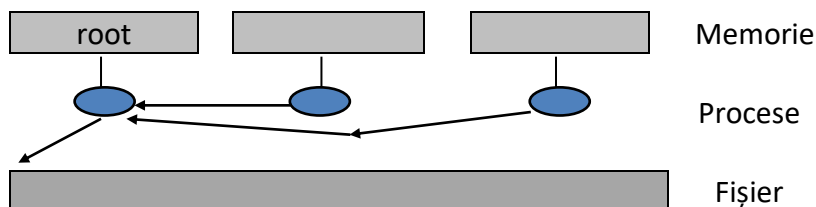


## Lectia 12. UTILIZAREA FIȘIERELOR ÎN MPI. LUCRAREA DE LABORATOR NR. 4

### 12.1 Operațiile de intrare/ieșire (I/O) în programe MPI

În acest paragraf vom analiza următoarele modalități de utilizare a fișierelor în programarea paralelă.

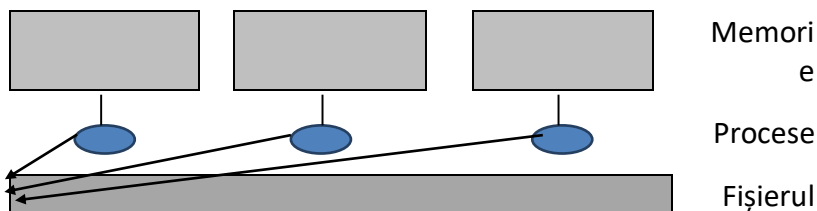
1. Utilizarea neparalelă a fișierelor – procesul root recepționează datele de la procese utilizând funcțiile MPI de transmitere/recepționare a mesajelor și apoi le scrie/citește în fișier. Acest mod schematic poate fi reprezentat astfel:



Această modalitate de utilizare a fișierelor poate fi exemplificată prin următorul cod de program:

```
#include "mpi.h"
#include <stdio.h>
#define BUFSIZE 100
int main(int argc, char *argv[]){
    int i, myrank, numprocs, buf[BUFSIZE];
    MPI_Status status;
    FILE *myfile;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    for (i=0; i<BUFSIZE; i++)
        buf[i] = myrank * BUFSIZE + i;
    if (myrank != 0)
        MPI_Send(buf, BUFSIZE, MPI_INT, 0, 99, MPI_COMM_WORLD);
    else {
        myfile = fopen("testfile", "w");
        fwrite(buf, sizeof(int), BUFSIZE, myfile);
        for (i=1; i<numprocs; i++) {
            MPI_Recv(buf, BUFSIZE, MPI_INT, i, 99, MPI_COMM_WORLD,
                &status);
            fwrite(buf, sizeof(int), BUFSIZE, myfile);
        }
        fclose(myfile);
    }
    MPI_Finalize();
    return 0;
}
```

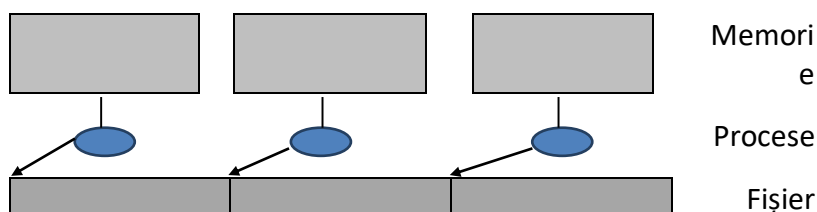
2. Fiecare proces utilizează în paralel fișierul său propriu. Acest mod schematic poate fi reprezentat astfel:



Această modalitate de utilizare a fișierelor poate fi exemplificată prin următorul cod de program:

```
#include "mpi.h"
#include "mpi.h"
#include <stdio.h>
#define BUFSIZE 100
int main(int argc, char *argv[])
{
    int i, myrank, buf[BUFSIZE];
    char filename[128];
    FILE *myfile;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    for (i=0; i<BUFSIZE; i++)
        buf[i] = myrank * BUFSIZE + i;
    sprintf(filename, "testfile.%d", myrank);
    myfile = fopen(filename, "w");
    fwrite(buf, sizeof(int), BUFSIZE, myfile);
    fclose(myfile);
    MPI_Finalize();
    return 0;
}
```

3. Utilizarea paralelă de către toate procesele unui mediu de comunicare a unui și același fișier. Acest mod schematic poate fi reprezentat astfel:



În acest paragraf vom studia detaliat modul 3) de utilizare a fișierelor. La început vom defini următoarele noțiuni:

**E-type** (tip elementar de date) – unitate de acces la date și de poziționare. Acest tip poate fi un tip predefinit în MPI sau un tip derivat de date.

**F-type** (tip fișier) – servește drept bază pentru partiționarea fișierului în mediul de procese, definește un șablon de acces la fișier. Reprezintă un șir de e-tip-uri sau tipuri derivate de date MPI.


**Vedere fișier** (file view) – un set de date vizibile și accesibile dintr-un fișier deschis ca un set ordonat de e-tipuri. Fiecare proces are propria sa vedere fișier specificată de următorii trei parametri: offset, e-tip și f-tip. Șablonul descris de f-tip se repetă începând cu poziția offset.

**Offset** – aceasta este poziția în fișier în raport cu vederea curentă, prezentă ca un număr de **e-tip**-uri. „Găurile” în **f-tip** sunt omise la calcularea numărului poziției. Zero este poziția primului **e-tip** vizibil în vederea fișierului.

**Referințe de fișier individuale** – referințe de fișier care sunt locale pentru fiecare proces într-un fișier deschis.

**Referințe de fișier comune** – referințe de fișier utilizate în același timp de un grup de procese pentru care este deschis fișierul.

Vom reprezenta grafic noțiunile definite mai sus.

 e-tip

 **f-tip**-ul pentru procesul 0

 **f-tip**-ul pentru procesul 1

 **f-tip**-ul pentru procesul 2

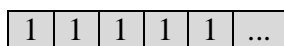
 **f-tip**-ul pentru procesul 3

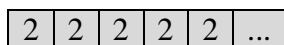
„Placarea” (tiling) fișierului

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |      |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|------|
| 0 | 1 | 1 | 2 | 2 | 3 | 0 | 1 | 1 | 2 | 2 | 3 | 0 | 1 | 1 | 2 | 2 | 3 | .... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|------|

Astfel fiecare proces „va vedea” și deci va avea acces la următoarele date:

 Procesul 0

 Procesul 1

 Procesul 2

 Procesul 3

Vom descrie algoritmul de bază pentru utilizarea fișierelor în programe MPI.

1. Definirea variabilelor și tipurilor necesare de date pentru construirea e-tip-urilor și a **f-tip**-urilor.
2. Deschiderea unui fișier (funcția **MPI\_File\_open**).
3. Determinarea pentru fiecare proces vederea fișier (funcția **MPI\_File\_set\_view**).
4. Citire/scriere de date (funcțiile **MPI\_File\_write**, **MPI\_File\_read**).
5. Închidere fișier (funcția **MPI\_File\_close**).

## 12.2 Funcțiile MPI pentru utilizarea fișierelor

În acest paragraf vom prezenta funcțiile de bază pentru utilizarea fișierelor în programe MPI.

### Funcția **MPI\_File\_open**

```
int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info
info, MPI_File *fh)
```

unde

|                    |   |
|--------------------|---|
| IN <b>comm</b>     | – nume comunicator;   |
| IN <b>filename</b> | – numele fișierului;  |
| IN <b>amode</b>    | – tipul de operații care se pot<br>executa asupra fișierului; |
| IN <b>info</b>     | – obiect informațional;                                       |
| OUT <b>fh</b>      | – descriptorul de fișier.                                     |

Valorile posibile ale parametrului **amode**:

**MPI\_MODE\_RDONLY** – accesibil numai la citire;

**MPI\_MODE\_RDWR** – accesibil la citire și înscris;

**MPI\_MODE\_WRONLY** – accesibil numai la înscriere;  
**MPI\_MODE\_CREATE** – creare fișier, dacă nu există;  
**MPI\_MODE\_EXCL** – eroare, dacă fișierul creat deja există;  
**MPI\_MODE\_DELETE\_ON\_CLOSE** – distrugere fișier la închidere;  
**MPI\_MODE\_UNIQUE\_OPEN** – fișierul nu se va deschide și de alte procese;  
**MPI\_MODE\_SEQUENTIAL** – acces secvențial la datele din fișier;  
**MPI\_MODE\_APPEND** – indicarea poziției inițiale a parametrului offset la sfârșitul fișierului.

La fel se pot utiliza și combinații logice așe diferitor valori.

#### Funcția *MPI\_File\_set\_view*

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype
    etype, MPI_Datatype filetype, char *datarep, MPI_Info info)
```

unde

|                    |   |
|--------------------|---|
| IN/OUT <b>fh</b>   | – descriptorul de fișier;                     |
| IN <b>disp</b>     | – valoarea deplasării;                        |
| IN <b>etype</b>    | – tipul elementar de date;                    |
| IN <b>filetype</b> | – tipul fișier de date;                       |
| IN <b>datarep</b>  | – modul de reprezentare a datelor din fișier; |
| IN <b>info</b>     | – obiect informațional.                       |

Valorile parametrului **datarep** pot fi „native”, „internal”, „external32” sau definite de utilizator.

#### Funcția *MPI\_File\_read*

```
int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype
    datatype, MPI_Status *status)
```

unde

|                    |   |
|--------------------|---|
| IN/OUT <b>fh</b>   | – descriptorul de fișier;                   |
| OUT <b>buf</b>     | – adresa de start a tamponului de date;     |
| IN <b>count</b>    | – numărul de elemente din tamponul de date; |
| IN <b>datatype</b> | – tipul de date din tamponul <b>buf</b> ;   |
| OUT <b>status</b>  | – obiectul de stare.                        |

Procesul MPI care face apel la această funcție va citi **count** elemente de tip **datatype** din fișierul **fh**, în conformitate cu vederea la fișier fixată de funcția **MPI\_File\_set\_view**, și le va memora în variabila **buf**.

#### Funcția *MPI\_File\_write*

```
int MPI_File_write(MPI_File fh, void *buf, int count, MPI_Datatype
    datatype, MPI_Status *status)
```

unde

|                    |   |
|--------------------|---|
| IN/OUT <b>fh</b>   | – descriptorul de fișier;                   |
| IN <b>buf</b>      | – adresa de start a tamponului de date;     |
| IN <b>count</b>    | – numărul de elemente din tamponul de date; |
| IN <b>datatype</b> | – tipul de date din tamponul <b>buf</b> ;   |
| OUT <b>status</b>  | – obiectul de stare.                        |

Procesul MPI care face apel la această funcție va înscrie **count** elemente de tip **datatype** din variabila **buf** în fișierul **fh**, în conformitate cu vederea la fișier fixată de funcția **MPI\_File\_set\_view**.

În tabelul de mai jos vom prezenta rutinele MPI pentru operațiile de citire/sciere a datelor din/în fișier.

| Mod de poziționare      | Mod de sincronizare | Mod de coordonare   |  |
|-------------------------|---------------------|---|--|
|                         |                     | noncolectiv   | colectiv   |
| Explicit offset         | cu blocare          | <code>MPI_File_read_at</code><br><code>MPI_File_write_at</code>           | <code>MPI_File_read_at_all</code><br><code>MPI_File_write_at_all</code>  |
|                         | cu nonblo-care      | <code>MPI_File_iread_at</code><br><code>MPI_File_iwrite_at</code>         | <code>MPI_File_read_at_all_begin</code><br><code>MPI_File_read_at_all_end</code><br><code>MPI_File_write_at_all_begin</code><br><code>MPI_File_write_at_all_end</code>     |
| Poziționare individuală | cu blocare          | <code>MPI_File_read</code><br><code>MPI_File_write</code>                 | <code>MPI_File_read_all</code><br><code>MPI_File_write_all</code>  |
|                         | cu nonblo-care      | <code>MPI_File_iread</code><br><code>MPI_File_iwrite</code>               | <code>MPI_File_read_all_begin</code><br><code>MPI_File_read_all_end</code><br><code>MPI_File_write_all_begin</code><br><code>MPI_File_write_all_end</code>                 |
| Poziționare colectivă   | cu blocare          | <code>MPI_File_read_shared</code><br><code>MPI_File_write_shared</code>   | <code>MPI_File_read_ordered</code><br><code>MPI_File_write_ordered</code>  |
|                         | cu nonblo-care      | <code>MPI_File_iread_shared</code><br><code>MPI_File_iwrite_shared</code> | <code>MPI_File_read_ordered_begin</code><br><code>MPI_File_read_ordered_end</code><br><code>MPI_File_write_ordered_begin</code><br><code>MPI_File_write_ordered_end</code> |

Vom ilustra utilizarea funcțiilor MPI pentru realizarea operațiilor de citire/scriere a datelor din/în fișier prin următorul exemplu

**Exemplul 12.1** Fie dată o matrice de dimensiunea 4x8. Să se elaboreze un program MPI în limbajul C++ care va realiza următoarele:

- se creează un fișier și procesul 0 scrie în el primele două rânduri ale matricei, procesul 1 scrie în el linia a treia a matricei, și la rândul său, procesul 2 scrie în el rândul al patrulea al matricei;
- procesul 3 citește primele două rânduri ale matricei din fișierul creat și le tipărește, corespunzător, procesul 4 (procesul 5) citește rândul trei (patru) al matricei din fișierul creat și le tipărește.

**Indicație.** Rândurile matricei sunt inițializate de procesele corespunzătoare.

Mai jos este prezentat codul programului în limbajul C++ în care se realizează cele indicate în exemplul 12.1.

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv)
{
    int rank, count, amode, i, j;
    MPI_File OUT;
    MPI_Aint rowsize;
    MPI_Datatype etype, ftype0, ftype1, ftype2;
    MPI_Status state;
    MPI_Datatype MPI_ROW;
    int blengths[2]={0,1};
    MPI_Datatype types[2];
    MPI_Aint disps[2];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```

amode=MPI_MODE_DELETE_ON_CLOSE;
MPI_File_open(MPI_COMM_WORLD, "array.dat", amode, MPI_INFO_NULL, &OUT);
MPI_File_close(&OUT);
amode=MPI_MODE_CREATE | MPI_MODE_RDWR;
MPI_File_open(MPI_COMM_WORLD, "array.dat", amode, MPI_INFO_NULL, &OUT);
MPI_Type_contiguous(8, MPI_INT, &MPI_ROW);
MPI_Type_commit(&MPI_ROW);
etype=MPI_ROW;
ftype0=MPI_ROW;
types[0]=types[1]=MPI_ROW;
disps[0]=(MPI_Aint) 0;
MPI_Type_extent(MPI_ROW, &rowsize); disps[1]=2*rowsize;
MPI_Type_struct(2, blengths, disps, types, &ftype1);
MPI_Type_commit(&ftype1);
disps[1]=3*rowsize;
MPI_Type_struct(2, blengths, disps, types, &ftype2);
MPI_Type_commit(&ftype2);
count=0;
// Inscrierea in paralel a datelor in fisierul "array.dat"
if (rank==0)
{
    int value0[2][8];
    for (i=1; i<=2; ++i)
        for (j=1; j<=8; ++j)
            value0[i-1][j-1]= 10*i+j;
    MPI_File_set_view(OUT, 0, etype, ftype0, "native", MPI_INFO_NULL);
    MPI_File_write(OUT, &value0[rank][0], 2, MPI_ROW, &state);
    MPI_Get_count(&state, MPI_ROW, &count);
    printf("===Procesul %d a in scris in fisierul ""array.dat"" urmatoarele %d randuri:\n", rank, count);
    for(int i = 0; i<2; i++)
    {
        for(int j = 0; j<8; j++)
            printf("%d\t", value0[i][j]);
        printf("\n");
    }
}
if (rank==1)
{
    int value1[8];
    for (j=1; j<=8; ++j)
        value1[j-1]= 10*3+j;
    MPI_File_set_view(OUT, 0, etype, ftype1, "native", MPI_INFO_NULL);
    MPI_File_write(OUT, &value1, 1, MPI_ROW, &state);
    MPI_Get_count(&state, MPI_ROW, &count);
    printf("===Procesul %d a in scris in fisierul ""array.dat"" urmatoarele %d randuri\n", rank, count);
    for(int j = 0; j<8; j++)
    {
        printf("%d\t", value1[j]);
    }
}

```

```

printf("\n");
}
if (rank==2)
{
int value2[8];
for (j=1;j<=8;++j)
value2[j-1]= 10*4+j;
MPI_File_set_view(OUT,0,etype, ftype2,"native",MPI_INFO_NULL);
MPI_File_write(OUT,&value2,1, MPI_ROW,&state);
MPI_Get_count(&state,MPI_ROW, &count);
printf("===Procesul %d a in scris in fisierul ""array.dat"" urmatoarele %d randuri\n",rank,count);
for(int j = 0; j<8; j++)
{
printf("%d\t", value2[j]);
}
printf("\n");
}
// Citirea in paralel a datelor din fisierul "array.dat"
if (rank==3)
{
int myval3[2][8];
MPI_File_set_view(OUT,0,etype, ftype0,"native",MPI_INFO_NULL);
MPI_File_read(OUT,&myval3[0][0], 2,MPI_ROW,&state);
MPI_Get_count(&state,MPI_ROW, &count);
printf("---Procesul %d a citit din fisierul ""array.dat"" urmatoarele %d rinduri\n",rank,count);
for(int i = 0; i<2; i++)
{
for(int j = 0; j<8; j++)
printf("%d\t", myval3[i][j]);
printf("\n");
}
}
if (rank==4)
{
int myval4[8];
MPI_File_set_view(OUT,0,etype, ftype1,"native",MPI_INFO_NULL);
MPI_File_read(OUT,&myval4,1, MPI_ROW,&state);
MPI_Get_count(&state,MPI_ROW, &count);
printf("---Procesul %d a citit din fisierul ""array.dat"" urmatoarele %d rinduri\n",rank,count);
for(int j = 0; j<8; j++)
{
printf("%d\t", myval4[j]);
}
printf("\n");
}
if (rank==5)
{
int myval5[8];
MPI_File_set_view(OUT,0,etype, ftype2,"native",MPI_INFO_NULL);

```

```

MPI_File_read(OUT,&myval5,1, MPI_ROW,&state);
MPI_Get_count(&state,MPI_ROW, &count);
printf("---Procesul %d a citit din fisierul ""array.dat"" urmatoarele %d rinduri\n",rank,count);
    for(int j = 0; j<8; j++)
    {
        printf("%d\t", myval5[j]);
    }
    printf("\n");
}
MPI_File_close(&OUT);
MPI_Finalize();
}

```

Rezultatele posibile ale executării programului:

```

[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpiCC -o Exemplu_3_9_1.exe Exemplu_3_9_1.cpp1
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpirun -n 6 -machinefile ~/nodes6 Exemplu_3_9_1.exe
===Procesul 1 a in scris in fisierul array.dat urmatoarele 1 randuri
31    32    33    34    35    36    37    38
---Procesul 3 a citit din fisierul array.dat urmatoarele 2 rinduri
11    12    13    14    15    16    17    18
21    22    23    24    25    26    27    28
---Procesul 5 a citit din fisierul array.dat urmatoarele 1 rinduri
41    42    43    44    45    46    47    48
---Procesul 4 a citit din fisierul array.dat urmatoarele 1 rinduri
31    32    33    34    35    36    37    38
===Procesul 2 a in scris in fisierul array.dat urmatoarele 1 randuri
41    42    43    44    45    46    47    48
===Procesul 0 a in scris in fisierul array.dat urmatoarele 2 randuri:
11    12    13    14    15    16    17    18
21    22    23    24    25    26    27    28
[Hancu_B_S@]$ /opt/openmpi/bin/mpirun -n 2 -machinefile ~/nodes6 Exemplu_3_9_1.exe
===Procesul 0 a in scris in fisierul array.dat urmatoarele 2 randuri:
11    12    13    14    15    16    17    18
21    22    23    24    25    26    27    28
===Procesul 1 a in scris in fisierul array.dat urmatoarele 1 randuri
31    32    33    34    35    36    37    38
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpirun -n 4 -machinefile ~/nodes6 Exemplu_3_9_1.exe
===Procesul 1 a in scris in fisierul array.dat urmatoarele 1 randuri
31    32    33    34    35    36    37    38
===Procesul 0 a in scris in fisierul array.dat urmatoarele 2 randuri:
11    12    13    14    15    16    17    18
21    22    23    24    25    26    27    28
---Procesul 3 a citit din fisierul array.dat urmatoarele 2 rinduri
11    12    13    14    15    16    17    18
21    22    23    24    25    26    27    28
===Procesul 2 a in scris in fisierul array.dat urmatoarele 1 randuri
41    42    43    44    45    46    47    48

```

<sup>1</sup> Numele programului corespunde numelui exemplului din notele de curs Boris HÎNCU, Elena CALMÎȘ "MODELE DE PROGRAMARE PARALELĂ PE CLUSTERE. PARTEA I. PROGRAMARE MPI". Chisinau 2016.



Fișierul array.dat

```
[Hancu_B_S@hpc]$ od -d array.dat
```

```
000000  11  0 12  0 13  0 14  0
000020  15  0 16  0 17  0 18  0
000040  21  0 22  0 23  0 24  0
000060  25  0 26  0 27  0 28  0
000100  31  0 32  0 33  0 34  0
000120  35  0 36  0 37  0 38  0
000140  41  0 42  0 43  0 44  0
000160  45  0 46  0 47  0 48  0
```

## 12.2 Lucrare de laborator nr. 4

**Titlul lucrării** *Utilizarea fișierelor MPI pentru prelucrarea paralelă a datelor structurate în forma unor tabele de dimensiuni foarte mari.*

Fie dată o matrice  $A = \left\| a_{ij} \right\|_{\substack{i=1,m \\ j=1,n}}$  care este divizată în blocuri  $A_{kp}$  de dimensiunea  $m_k \times n_p$ . Să se

elaboreze și să se execute pe clusterul USM un program MPI în limbajul C++ în care fiecare proces cu coordonatele  $(k, p)$  dintr-un comunicator cu topologie carteziana inițializează cu valori aleatoare matricea  $A_{kp}$  și înscrie în același fișier submatricea  $A_{kp}$ . După aceasta un alt grup de procese cu

coordoanatele  $(\tilde{k}, \tilde{p})$  dintr-un comunicator cu topologie carteziana, citește din fișierul creat submatricea  $A_{\tilde{k} \tilde{p}}$  și determina elementul maximal al matricei pe care îl trimite procesului root, care la randul sau va

determina elementul maximal al întregii matrici. Să se elaboreze un prgram MPI în care:

- Matricea A se divizeaza în submatrici utilizand algoritmul 2D-ciclic (a se vedea Lucrarea de laborator nr. 3)
- Matricea A se divizeaza în submatrici în mod arbitrar.