Lectia 11. TIPURI DE DATE MPI

11.1 Harta tipului de date

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. 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.

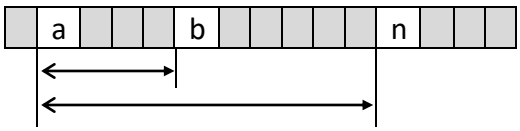
Typemap = {(type0 , disp0), . . . , (typen-1, dispn-1)}

Typesig = {type0, . . . , typen-1}

Fie că într-un program sunt declarate variabilele

```
float a,b;
int n;
```

Schematic presupunem că variabilele sunt stocate în memorie astfel:



Atunci harta noului tip de date creat în baza tipurilor indicate va fi

{(MPI_FLOAT, 0), (MPI_FLOAT, 4), (MPI_INT, 10)}

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

- acesta are *n* intrări;
- fiecare intrare *i* are tipul typei și începe la adresa buf+ dispi.

Vom face următoarea 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 astfel folosim 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 asamblează datele, se utilizează 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 multiple de *8*. Deci, chiar dacă dimensiunea tipului este *9*, din motive de aliniere, o nouă valoare începe la o distanţă de *16* octeți de începutul valorii precedente. Extinderea și dimensiunea unui tip de date pot fi aflate prin apelurile funcțiilor **MPI_Type_extent** și **MPI_Type_size**. Prototipul în limbajul C++ a acestor funcții este

```
int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
```

unde

IN **datatype**	– numele tipului de date;
OUT **extent**	– extinderea tipului de date;

```
int MPI_Type_size(MPI_Datatype datatype, int *size);
```

unde

IN **datatype** – numele tipului de date;
OUT **size** – dimensiunea tipului de date.

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 funcția **MPI_Address**. Prototipul în limbajul C++ a acestei funcții este

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

unde

IN **location** – locația de memorie (numele variabilei);
OUT **address** – adresa.

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

11.2 Tipuri derivate de date

MPI prevede diferite funcții care servesc drept constructori de tipuri derivate de date. Utilizarea tipurilor noi de date va permite extinderea modalităților de transmitere/recepționare a mesajelor în MPI. Pentru generarea unui nou tip de date (cum s-a menționat mai sus) este nevoie de următoarea informație:

- ✓ numărul de elemente care vor forma noul tip de date;
- ✓ o listă de tipuri de date deja existente sau prestabilite;
- ✓ adresa relativă din memorie a elementelor.

Funcția *MPI_Type_contiguous*

Este cel mai simplu constructor care produce un nou tip de date făcând mai multe copii ale unui tip existent, cu deplasări care sunt multipli ai extensiei tipului vechi. Prototipul în limbajul C++ a acestei funcții este

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
    MPI_Datatype *newtype)
```

unde

IN **count** – numărul de copii (replicări);
IN **oldtype** – tipul vechi de date;
OUT **newtype** – tipul nou de date.

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 în mod obligatoriu trebuie încredințat sistemului înainte de a fi utilizat în operațiile de transmitere-recepționare. Acest lucru se face în baza funcției

```
int MPI_Type_commit(&newtype).
```

Când un tip de date nu mai este folosit, el trebuie eliberat. Aceasta se face utilizând funcția

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

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);
```

Funcția *MPI_Type_vector*

Acestă funcție 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). Prototipul în limbajul C++ a acestei funcții este

```
int MPI_Type_vector(int count, int blocklength, int
    stride,MPI_Datatype oldtype, MPI_Datatype *newtype)
```

unde

IN **count**	– numărul de blocuri de date;
IN **blocklength**	– numărul de elemente în fiecare bloc de date;
IN **stride**	– pasul, deci numărul de elemente între începuturile a două blocuri vecine;
IN **oldtype**	– vechiul tip de date;
OUT **newtype**	– tipul nou de date.

Pentru exemplificare, să considerăm o matrice a de *5* linii și *7* coloane (cu elementele de tip **floa**t memorate pe linii). Pentru a construi tipul de date corespunzător unei coloane folosim funcția:

MPI_Type_vector (5, 1, 7, MPI_FLOAT, &columntype);

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.

Atunci pentru a transmite coloana a treia a matricii se poate folosi funcția

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

Funcția *MPI_Type_indexed*

Această funcție se utilizează pentru generarea unui tip de date pentru care 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. Prototipul în limbajul C++ al acestei funcții este

```
int MPI_Type_indexed(int count, int *array_of_blocklengths, int
    *array_of_displacements, MPI_Datatype oldtype, MPI_Datatype
    *newtype)
```

unde

IN **count**	– numărul de blocuri de date;
IN **array_of_blocklengths**	– numărul de elemente pentru fiecare bloc de date;
IN **array_of_displacements**	– pasul pentru fiecare bloc (în octeți),
IN **oldtype**	– vechiul tip de date;
OUT **newtype**	– tipul nou de date.

Pentru exemplificare, să considerăm o matrice a de 4 linii și 4 coloane (cu elementele de tip **int** memorate pe linii). Pentru a construi tipul de date corespunzător diagonalei principale folosim funcția:

MPI_Type_ indexed(4, block_lengths, relloc, MPI_INT, &diagtype);

unde **block_lengths={1, 1, 1}** și **relloc={4, 4, 4, 4}.**

Funcția *MPI_Type_struct*

Această funcție este o generalizare a funcțiilor precedente prin aceea că permite să genereze tipuri de date pentru care fiecare bloc să constea din replici ale unor tipuri de date diferite. Prototipul în limbajul C++ al acestei funcții este

```
int MPI_Type_struct(int count, int array_of_blocklengths[],MPI_Aint
    array_of_displacements[],MPI_Datatype array_of_types[],
    MPI_Datatype *newtype)
```

unde

IN **count**	– numărul de blocuri de date;
IN **array_of_blocklengths**	– numărul de elemente pentru fiecare bloc de date;
IN **array_of_displacements**	– pasul pentru fiecare bloc (în octeți);
IN **array_of_types**	– vechiul tip de date pentru fiecare bloc;
OUT **newtype**	– tipul nou de date.

Vom ilustra utilizarea funcțiilor de generare a tipurilor noi de date pentru realizarea operațiilor de trimitere/recepționare prin următorul exemplu.

Exemplul 11.1 *Să se elaboreze un program MPI în limbajul C++ pentru generarea noilor tipuri de date și utilizarea lor la operațiile de trimitere/recepționare.*

Indicație. Tipul vechi de date este o structură din două elemente care indică poziția în spațiu și masa unei particule.

Mai jos este prezentat codul programului în limbajul C++ în care se realizează cele indicate în exemplul 11.1.

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
typedef struct
{
    float position[3];
    float mass;
} Particle;
MPI_Datatype MPI_Particle;
    void construct_datatypes(void)
    {
    Particle p;
    int blens[2];
    MPI_Aint displ[2];
    MPI_Datatype types[2];
    blens[0]=3; types[0]=MPI_FLOAT;
    displ[0]=(MPI_Aint)&p.position-(MPI_Aint)&p;
    blens[1]=1; types[1]=MPI_FLOAT;
```

```
      displ[1]=(MPI_Aint)&p.mass-(MPI_Aint)&p;
      MPI_Type_struct(2,blens,displ,types,
      &MPI_Particle);
      MPI_Type_commit(&MPI_Particle);
      return;
      }
int main(int argc, char *argv[])
{
int nProc,myRank,i;
Particle *myP;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,
      &nProc);
MPI_Comm_rank(MPI_COMM_WORLD,
      &myRank);
construct_datatypes();
if (myRank ==0)
printf("\n=====REZULTATUL PROGRAMULUI '%s' \n",argv[0]);
MPI_Barrier(MPI_COMM_WORLD);
myP=(Particle*)calloc(nProc,sizeof(Particle));
if(myRank == 0){
for(i=0;i<nProc;i++){
myP[i].position[0]=i;myP[i].position[1]=i+1;myP[i].position[2]=i+2;
myP[i].mass=10+100.0*rand()/RAND_MAX;
            }
           }
MPI_Bcast(myP,nProc,MPI_Particle,0, MPI_COMM_WORLD);
printf("Proces rank %d: pozitia particuleei (%f, %f, %f) masa ei %f\n", myRank, myP[myRank].position[0],
      myP[myRank].position[1], myP[myRank].position[2], myP[myRank].mass);
MPI_Finalize();
return 0;
}
```

Rezultatele posibile ale executării programului:

```
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpiCC -o Exemplu_3_8_1.exe Exemplu_3_8_1.cpp[1]
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpirun -n 16 -machinefile ~/nodes4  Exemplu_3_8_1.exe

=====REZULTATUL PROGRAMULUI 'Exemplu_3_8_1.exe'
Proces rank 0: pozitia particuleei (0.000000, 1.000000, 2.000000) masa ei 94.018768
Proces rank 2: pozitia particuleei (2.000000, 3.000000, 4.000000) masa ei 88.309921
Proces rank 4: pozitia particuleei (4.000000, 5.000000, 6.000000) masa ei 101.164734
Proces rank 12: pozitia particuleei (12.000000, 13.000000, 14.000000) masa ei 46.478447
Proces rank 8: pozitia particuleei (8.000000, 9.000000, 10.000000) masa ei 37.777470
Proces rank 6: pozitia particuleei (6.000000, 7.000000, 8.000000) masa ei 43.522274
Proces rank 14: pozitia particuleei (14.000000, 15.000000, 16.000000) masa ei 105.222969
Proces rank 10: pozitia particuleei (10.000000, 11.000000, 12.000000) masa ei 57.739704
Proces rank 1: pozitia particuleei (1.000000, 2.000000, 3.000000) masa ei 49.438293
Proces rank 3: pozitia particuleei (3.000000, 4.000000, 5.000000) masa ei 89.844002
```

[1] 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.

```
Proces rank 5: pozitia particuleei (5.000000, 6.000000, 7.000000) masa ei 29.755136
Proces rank 15: pozitia particuleei (15.000000, 16.000000, 17.000000) masa ei 101.619507
Proces rank 11: pozitia particuleei (11.000000, 12.000000, 13.000000) masa ei 72.887093
Proces rank 7: pozitia particuleei (7.000000, 8.000000, 9.000000) masa ei 86.822960
Proces rank 13: pozitia particuleei (13.000000, 14.000000, 15.000000) masa ei 61.340092
Proces rank 9: pozitia particuleei (9.000000, 10.000000, 11.000000) masa ei 65.396996
[Hancu_B_S@hpc]$
```