Lecția 9. ACCESUL DISTANT LA MEMORIE. FUNCȚIILE RMA

9. 1 Comunicare unic-direcționată

Funcțiile RMA (Remote Memory Access) extind mecanismele de comunicare MPI, permițând unui proces să specifice parametrii de comunicare pentru ambele procese implicate în transmiterea 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. Operațiile de trimitere, recepționare și actualizare a datelor sunt reprezentate în MPI prin funcțiile **`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 dispoziția celorlalte pentru executarea funcțiilor RMA, și **`MPI_Win_free`** pentru eliberarea ferestrei, ca și operații de sincronizare a proceselor care fac acces la datele altui proces. Comunicările RMA se pot împărți în următoarele 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 participă 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).

Pentru comoditate, vom numi *inițiator* (`origin`) procesul care face un apel la o funcție RMA și *destinatar* (`target`) procesul la memoria căruia se face adresarea. Astfel, pentru operația `put` sursa de date este procesul *inițiator* (`source=origin`), locul de destinație al datelor este procesul *destinatar* (`destination=target`); și pentru operația `get` sursa de date o constituie procesul *destinatar* (`source=target`) și locul de destinație al datelor este procesul *inițiator* (`destination=origin`).

Funcția MPI_Win_create

Pentru a permite accesul memoriei de la distanță un proces trebuie să selecteze o regiune continuă de memorie și să o facă accesibilă pentru alt proces. Această regiune se numește *fereastră*. Celălalt proces trebuie să știe despre această fereastră. MPI realizează aceasta prin funcția colectivă **`MPI_Win_create`**. Pentru că este o funcție colectivă, toate procesele trebuie să deschidă o fereastră, dar dacă un proces nu trebuie să partajeze memoria sa, el poate defini fereastra de dimensiunea *0* (dimensiunea ferestrei poate fi diferită pentru fiecare proces implicat în comunicare. Prototipul funcției în limbajul C++ este

```
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit,
    MPI_Info info,MPI_Comm comm, MPI_Win *win)
```

unde

IN **base**	– adresa inițială (de start) a ferestrei care se indică prin numele unei variabile;
IN **size**	– dimensiunea ferestrei în octeți;
IN **disp_unit**	– valoarea în octeți a deplasării;
IN **info**	– argumentul pentru informație suplimentară;
IN **comm**	– numele comunicatorului;
OUT **win**	– numele ferestrei.

Funcția MPI_Win_free

Această funcție eliberează memoria ocupată de **`win,`** returnează o valoare egală cu **`MPI_WIN_NULL`**. Apelul este colectiv și se realizează de toate procesele din grupul asociat ferestrei **`win.`** Prototipul funcției în limbajul C++ este

```
int MPI_Win_free(MPI_Win *win)
```

unde

IN/OUT **win** – numele ferestrei.

9.2 Funcțiile RMA

Funcția MPI_Put

Executarea unei operații put este similară executării unei operații send de către procesul *inițiator* și, respectiv, receive de către procesul *destinatar*. Diferența constă în faptul că toți parametrii necesari executării operațiilor send-receive sunt puși la dispoziție de un singur apel realizat de procesul *inițiator*. Prototipul funcției în limbajul C++ este

```
int MPI_Put(void *origin_addr, int origin_count, MPI_Datatype
    origin_datatype, int target_rank, MPI_Aint target_disp, int
    target_count, MPI_Datatype target_datatype, MPI_Win win)
```

unde

IN **origin_addr** – adresa inițială (de start) a tamponului (buffer) pentru procesul *inițiator*;

IN **origin_count** – numărul de elemente al datelor din tamponul procesului *inițiator*;

IN **origin_datatype** – tipul de date din tamponul procesului *inițiator*;

IN **target_rank** – rankul procesului *destinatar*;

IN **target_disp** – deplasarea pentru fereastra procesului *destinatar*;

IN **target_count** – numărul de elemente al datelor din tamponul procesului *destinatar*;

IN **target_datatype** – tipul de date din tamponul procesului *destinatar*;

OUT **win** – numele ferestrei utilizate pentru comunicare de date.

Astfel, în baza funcției **MPI_Put** procesul *inițiator* (adică care execută funcția) transmite origin_count date de tipul origin_datatype, pornind de la adresa origin_addr procesului determinat de target_rank. Datele sunt scrise în tamponul procesului *destinație* la adresa target_addr=window_base + target_disp * disp_unit, unde window_base și disp_unit sunt parametrii ferestrei determinate de procesul *destinație* prin executarea funcției **MPI_Win_create.** Tamponul procesului destinație este determinat de parametrii target_count și target_datatype.

Transmisia de date are loc în același mod ca și în cazul în care procesul *inițiator* a executat funcția MPI_Send cu parametrii origin_addr, origin_count, origin_datatype, target rank, tag, comm, iar procesul *destinație* a executat funcția MPI_Receive cu parametrii taget_addr, target_datatype, source, tag, comm, unde target_addr este adresa tamponului pentru procesul *destinație*, calculat cum s-a explicat mai sus, și *com* – comunicator pentru grupul de procese care au creat fereastra win.

Funcția MPI_Get

Executarea unei operații get este similară executării unei operații receive de către procesul *destinatar*, și respectiv, send de către procesul *inițiator*. Diferența constă în faptul că toți parametrii necesari executării operațiilor send-receive sunt puși la dispoziție de un singur apel realizat de procesul *destinatar*. Prototipul funcției în limbajul C++ este

```
int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype
    origin_datatype, int target_rank, MPI_Aint target_disp, int
    target_count, MPI_Datatype target_datatype, MPI_Win win).
```

Funția **MPI_Get** este similară funcției **MPI_Put**, cu excepția faptului că transmiterea de date are loc în direcția opusă. Datele sunt copiate din memoria procesului *destinatar* în memoria procesului *inițiator*.

Funcția MPI_Acumulate

Este adesea util în operația **put** de a combina mai degrabă datele transferate la *procesul-destinatar* cu datele pe care le deține, decât de a face înlocuirea (modificarea) datelor în *procesul-inițiator*. Acest lucru permite, de exemplu, de a face acumularea unei sume de date, obligând procesele implicate să contribuie prin adăugarea la variabila de sumare, care se află în memoria unui anumit proces. Prototipul funcției în limbajul C++ este

```
int MPI_Accumulate(void *origin_addr, int origin_count,MPI_Datatype
    origin_datatype, int target_rank,MPI_Aint target_disp, int
    target_count,MPI_Datatype target_datatype, MPI_Op op, MPI_Win
    win)
```

unde

IN **origin_addr**	– adresa inițială (de start) a tamponului (buffer) pentru procesul *inițiator*;
IN **origin_count**	– numărul de elemente al datelor din tamponul procesului *inițiator*;
IN origin_datatype	– tipul de date din tamponul procesului *inițiator*;
IN **target_rank**	– rankul procesului *destinatar*;
IN **target_disp**	– deplasarea pentru fereastra procesului *destinatar*;
IN **target_count**	– numărul de elemente al datelor din tamponul procesului *destinatar*;
IN **target_datatype**	– tipul de date din tamponul procesului *destinatar*;
IN **op**	– operația de reducere;
OUT **win**	– numele ferestrei utilizate pentru comunicare de date.

Această funcție înmagazinează (acumulează) conținutul memoriei tampon a procesului *inițiator* (care este determinat de parametrii origin_addr, origin_datatype și origin_count) în memoria tampon determinată de parametrii target_count și target_datatype, target_disp ai ferestrei win a procesului target_rank, folosind operația de reducere op. Funcția este similară funcției **MPI_Put** cu excepția faptului că datele sunt acumulate în memoria procesului *destinatar*. Aici pot fi folosite oricare dintre operațiunile definite pentru funcția **MPI_Reduce**. Operațiile definite de utilizator nu pot fi utilizate.

Funcția MPI_Win_fencee

Această funcție se utilizează pentru sincronizarea colectivă a proceselor care apelează funcțiile RMA (**MPI_Put, MPI_Get, MPI_accumulate**). Astfel orice apel al funcțiilor RMA trebuie „bordat” cu funcția **MPI_Win_fence.** Prototipul funcției în limbajul C++ este

```
int MPI_Win_fence(int assert, MPI_Win win)
```

unde

IN	– un număr întreg;

assert

IN **win** – numele ferestrei.

Argumentul **assert** este folosit pentru a furniza diverse optimizări a procesului de sincronizare.

Vom ilustra rutinele MPI pentru accesul distant la memorie prin următorul exemplu.

Exemplul 9.1 *Să se calculeze valoarea aproximativă a lui π prin integrare numerică cu formula* $\pi = \int_0^1 \frac{4}{1+x^2} dx$*, folosind formula dreptunghiurilor. Intervalul închis [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. Pentru realizarea operațiilor colective:*

- ✓ *difuzarea valorilor lui n, tuturor proceselor;*
- ✓ *însumarea valorilor calculate de procese.*

Să se utilizeze funcțiile RMA.

Mai jos este prezentat codul programului în limbajul C++ care determină valoarea aproximativă a lui π folosind funcţiile RMA.

```
#include <stdio.h>
#include <mpi.h>
#include <math.h>
double f(double a)
{
   return (4.0 / (1.0 + a*a));
}
int main(int argc, char *argv[])
{
double PI25DT = 3.14159265358979323846264 3;
int n, numprocs, myid, i,done = 0;
double mypi, pi, h, sum, x;
int  namelen;
char processor_name[MPI_MAX_PROCESSOR_NAME];
MPI_Win nwin, piwin;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,
      &myid);
MPI_Get_processor_name(processor_name,&namelen);
//===Crearea locatiilor de memorie pentru realizarea RMA
if (myid==0)
   {
MPI_Win_create(&n,sizeof(int),1,MPI_INFO_NULL, MPI_COMM_WORLD, &nwin);
MPI_Win_create(&pi,sizeof(double),1, MPI_INFO_NULL, MPI_COMM_WORLD, &piwin);
   }
else
    {
// procesele nu vor utiliza datele din memoria ferestrelor sale nwin și piwin
    MPI_Win_create(MPI_BOTTOM, 0, 1, MPI_INFO_NULL, MPI_COMM_WORLD, &nwin);
    MPI_Win_create(MPI_BOTTOM, 0, 1, MPI_INFO_NULL, MPI_COMM_WORLD, &piwin);
    }
while (!done)
{
     if (myid == 0)
```

```
      {
      printf("Enter the number of intervals: (0 quits):\n");
         fflush(stdout);
         scanf("%d",&n);
         pi=0.0;
      }
//Procesele cu rank diferit de 0 "citesc" variabila n a procesului cu rank 0
      MPI_Win_fence(0,nwin);
    if (myid != 0)
      MPI_Get(&n, 1, MPI_INT, 0, 0, 1, MPI_INT, nwin);
      MPI_Win_fence(0,nwin);
if ( n == 0 ) done = 1;
else
{
h = 1.0 / (double) n;
sum = 0.0;
for (i = myid + 1; i <= n; i += numprocs)
  {
  x = h * ((double)i - 0.5); sum += f(x);
  }
mypi = h * sum;
MPI_Win_fence(0, piwin);
MPI_Accumulate(&mypi, 1,MPI_DOUBLE, 0, 0, 1,MPI_DOUBLE,MPI_SUM, piwin);
MPI_Win_fence(0, piwin);
if (myid == 0) {
printf("For number of intervals %d pi is approximately %.16f, Error is %.16f\n ", n, pi, fabs(pi-PI25DT));
fflush(stdout); }
                    }
}
MPI_Win_free(&nwin);
MPI_Win_free(&piwin);
MPI_Finalize();
return 0;
}
```

Rezultatele posibile ale executării programului:

```
 [Hancu_B_S@hpc]$ /opt/openmpi/bin/mpiCC -o Exemplu_3_6_1.exe Exemplu_3_6_1.cpp[1]
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpirun -n 16 -machinefile ~/nodes4  Exemplu_3_6_1.exe
Enter the number of intervals: (0 quits):
100000
For number of intervals 100000 pi is approximately 3.1415926535981260, Error is 0.0000000000083329
Enter the number of intervals: (0 quits):
100000000
For number of intervals 100000000 pi is approximately 3.1415926535897749, Error is
0.0000000000000182
 Enter the number of intervals: (0 quits):
```

[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

```
0
[Hancu_B_S@hpc]$
```