UNIVERSITATEA DE STAT DIN MOLDOVA
Facultatea de Matematică și Informatică
Departamentul Matematică
Boris HÎNCU, Elena CALMÎȘ

MODELE DE PROGRAMARE PARALELĂ PE CLUSTERE
PARTEA I. PROGRAMARE MPI

Note de curs

Aprobat de Consiliul Calității al USM

CEP USM
Chișinău, 2016

CZU .....
C .....
Recomandat de Consiliul Facultății de Matematică și Informatică

Autori: **Boris HÎNCU, Elena CALMÎȘ**
Recenzent: ***Grigore SECRIERU***, doctor în ştiinţe fizico-matematice, conferenţiar universitar, cercetător ştiinţific coordonator IMI al AŞM.

Descrierea CIP a Camerei Naționale a Cărții

Boris HÎNCU
Modele de programare paralela pe clustere / Elena Calmîș; Universitatea de Stat din Moldova. Facultatea de Matematică și Informatică, Departamentul Matematici Aplicate – Ch.: CEP USM, 2016.

..... ex.
ISBN .....

.....
.....

ISBN .....

Cuprins

Introducere....7
CAPITOLUL 1. SISTEME ŞI MODELE DE CALCUL PARALEL....11
1.1 Clasificarea sistemelor de calcul paralel....11
1.2 Definiția algoritmilor paraleli....15
CAPITOLUL 2. MODELE DE PROGRAMARE PARALELĂ....17
2.1 Sisteme de calcul cu memorie distribuită și partajată....17
2.2 Particularități de bază ale modelor de programare paralelă cu memorie partajată....19
CAPITOLUL 3. MODELUL MPI DE PROGRAMARE PARALELĂ PE SISTEME DE CALCUL CU MEMORIE DISTRIBUITĂ....22
3.1 Generalități....22
3.2 Funcțiile MPI de bază....24
3.3 Comunicarea de tip „proces-proces” (punct la punct)....27
3.3.1 Operații de bază....27
3.3.2 Funcţii MPI pentru transmiterea mesajelor....28
3.3.3 Exerciţii....32
3.4 Funcțiile MPI pentru comunicarea colectivă....32
3.4.1 Funcţiile MPI pentru asamblarea (adunarea) datelor....37
3.4.2 Funcţiile MPI pentru difuzarea (distribuirea) datelor....41
3.4.3 Funcţiile MPI pentru operații de reducere....47
3.4.4 Exerciţii....58
3.5 Rutine MPI de administrare a comunicatorului şi a grupelor....59
3.5.1 Grupe şi comunicatori....59
3.5.2 Funcţiile MPI de gestionare a grupelor de procese....61
3.5.3 Funcţiile MPI de gestionare a comunicatoarelor....64
3.5.4 Topologii virtuale. Rutine de creare a topologiei carteziene....68
3.5.5 Exerciţii....75
3.6 Accesul distant la memorie. Funcţiile RMA....75
3.6.1 Comunicare unic-direcționată....75
3.6.2 Funcţiile RMA....77
3.6.3 Exerciţii....82
3.7 Generarea dinamică a proceselor MPI....82

3.7.1 Modalităţi de generare dinamică a proceselor 82
3.7.2 Comunicarea între procesele generate dinamic 90
3.7.3 Exerciţii 94
3.8 Tipuri de date MPI 95
3.8.1 Harta tipului de date 95
3.8.2 Tipuri derivate de date 97
3.8.3 Exerciţii 101
3.9 Utilizarea fişierelor în MPI 102
3.9.1 Operaţiile de intrare/ieşire (I/O) în programe MPI 102
3.9.2 Funcţiile MPI pentru utilizarea fişierelor 106
3.9.3 Exerciţii 111
Bibliografie 112
Anexă 113

Terminologie generală pentru calcul și programare paralelă

Ca orice sector de activitate, calculul paralel și programarea paralelă are propriul „jargon”. În continuare sunt daţi câțiva termeni de uz obișnuit în calculul paralel.

Task – o secțiune logic discretă a efortului de calcul. Tipic, un task este un program sau un set de instrucțiuni asemănătoare unui program care este executat de un procesor.

Task paralel – un task care poate fi executat de procesoare multiple în condiții de siguranță (care generează rezultate corecte).

Execuție serială – executarea secvențială a unui program, declarație cu declarație. În sensul cel mai simplu, este ceea ce se întâmplă într-o mașină cu un singur procesor. Virtual, și când toate task-urile sunt paralele, există secțiuni ale unui program paralel care trebuie executate serial.

Execuție paralelă – execuție a unui program prin mai mult decât un task, cu fiecare task capabil să execute concomitent aceeași declarație sau declarații diferite.

Memorie partajată – din punct de vedere strict hardware, este o arhitectură în care toate procesoarele au acces direct (și de obicei bazat pe bus) la memoria fizică comună. În sensul programării, formula descrie un model în care task-urile paralele au toate aceeași „imagine” a memoriei și se pot adresa și pot accesa direct aceleași locații logice de memorie, indiferent unde există memoria fizică în realitate.

Memorie distribuită – în sensul hardware se referă la accesul memoriei, care fizic nu este comună, printr-o rețea. Ca model de programare, task-urile pot „vedea” logic numai memoria locală a mașinii și trebuie să apeleze la comunicare pentru a avea acces la memoria altor mașini unde alte task-uri sunt în curs de execuție.

Comunicare – obișnuit, task-urile paralele trebuie să schimbe date. Există mai multe moduri în care se poate comunica: printr-un bus al memoriilor utilizat în indiviziune (partajat) sau printr-o rețea. Indiferent de modalitatea utilizată, schimbul real de date este denumit uzual comunicare.

Sincronizare – coordonarea task-urilor paralele în timp real, foarte frecvent asociată cu comunicarea. Implementată adesea prin stabilirea unui punct de sincronizare în aplicație unde un task nu poate continua până când alt(-e) task(-uri) nu ating(-e) același punct sau același punct echivalent

logic. Sincronizarea implică uzual așteptări cel puțin pentru un task și poate cauza prin aceasta o creștere a timpului de execuție (timp universal – ceasul din perete) a aplicației paralele.

Granularitate – în calculul paralel, granularitatea este o măsură calitativă a raportului calcul/comunicare. Granularitatea poate fi:

- *grosieră*: relativ mult calcul între fazele (evenimentele) de comunicare.
- *fină*: relativ puțin calcul între fazele (evenimentele) de comunicare.

Accelerarea observată a unui cod care a fost paralelizat se definește ca raportul:

$$\frac{\text{timpul după ceasul din perete la execuția serială}}{\text{timpul după ceasul din perete la execuția paralelă}}$$

Este unul din cei mai simpli și mai larg utilizați indicatori ai performanţelor unui program paralel.

Overhead-ul paralel – durata necesară pentru a coordona task-urile paralele, în opoziție cu executarea de lucru util. Overhead-ul paralel poate include elemente de genul:

- timpul de pornire a task-ului;
- sincronizările;
- comunicarea de date;
- overhead-ul software impus de compilatoarele paralele, biblioteci, instrumente (tools), sistemul de operare etc.;
- timpul de încheiere a task-ului.

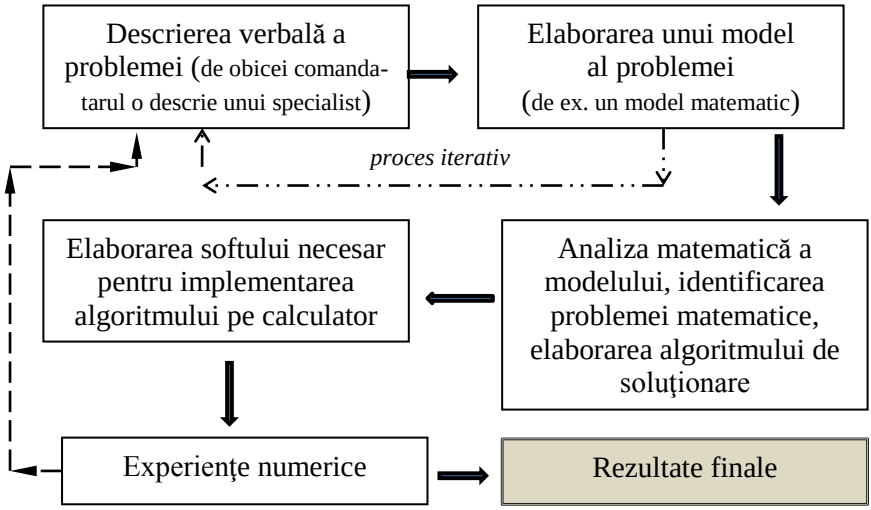
Masiv paralel – se referă la un sistem hardware care constituie un sistem paralel dat, care are multe procesoare. Semnificația lui „multe” este în creștere, dar în prezent „multe” înseamnă de la 1000 de procesoare în sus.

Scalabilitate – se referă la capacitatea unui sistem paralel (hardware și/sau software) de a demonstra în accelerarea (speedup) paralelă o creștere proporțională odată cu atașarea mai multor procesoare. Factori care contribuie la scalabilitate:

- hardware – în particular lărgimea de bandă și rețeaua de comunicare;
- algoritmul aplicației;
- overhead-ul paralel asociat;
- caracteristicile specifice ale aplicației și ale programului.

Introducere

Tehnologiile informaționale servesc drept suport (instrumentariu) pentru rezolvarea diferitor probleme teoretice și practice generate de activitatea umană. În general, procesul de rezolvare a problemelor cu caracter aplicativ poate fi reprezentat prin următoarea schemă:



O vastă clasă de probleme sunt de o complexitate foarte mare atât sub aspect de volum de date, cât și sub cel al numărului de operații. Astfel, apare o necesitate stringentă de a utiliza sisteme de calcul performant sau supercalculatoare. În prezent cele mai utilizate supercalculatoare sunt *calculatoarele paralele de tip cluster*.

În baza proiectului CRDF/MRDA „Project CERIM-1006-06" la Facultatea de Matematică și Informatică a Universității de Stat din Moldova a fost creat un laborator pentru utilizarea sistemelor paralele de calcul de tip cluster. Scopul acestui laborator este implementarea în procesul de instruire și cercetare a noilor tehnologii pentru elaborarea algoritmilor paraleli și executarea lor pe sisteme locale de tip cluster și chiar pe sisteme regionale de tip Grid. Actualmente laboratorul conține următoarele echipamente de calcul și de infrastructură:

- **1 server HP ProLiant DL380 G5**
  - ✓ CPU: 2x Dual Core Intel Xeon 5150 (2.7GHz)

- ✓ RAM: 8GB
- ✓ HDD: 2x72GB, 5x146GB
- ✓ Network: 3x1Gbps LAN

Acest echipament este utilizat la gestionarea mașinilor virtuale necesare pentru administrarea laboratorului, precum Web Server, FTP Server, PXE Server, Untangle server etc.

- **1 server HP ProLiant DL380 G5**
  - ✓ CPU: 2x QuadCore Intel Xeon 5420 (2.5 GHz)
  - ✓ RAM: 32GB
  - ✓ HDD: 2x72GB, 6x146GB
  - ✓ Network: 3x1Gbps LAN
- **2 servere HP ProLiant DL385G1**
  - ✓ CPU: 2xAMD 280 Dual-Core 2.4GHz
  - ✓ RAM: 6GB
  - ✓ HDD: SmartArray 6i, 4x146GB
  - ✓ Network: 3x1Gbps LAN

Aceste echipamente sunt utilizate pentru asigurarea logisticii necesare (la nivel de hard, soft și instruire de resurse umane) la realizarea și utilizarea MD-GRID NGI și a infrastructurii gridului European (EGI), care permit extinderea modalităților de utilizare a clusterului USM pentru realizarea unor calcule performante.

- **1 server HP ProLiant DL385G1 și 12 noduri HP ProLiant DL145R02**
  - ✓ CPU: 2xAMD 275 Dual-Core 2.2GHz
  - ✓ RAM: 4GB AECC PC3200 DDR
  - ✓ HDD: 80GB NHP SATA
  - ✓ Network: 3x1Gbps LAN
- **Storage HP SmartArray 6402**
  - ✓ hp StorageWorks MSA20
  - ✓ HDD: 4x500GB 7.2k SATA
- **Switch HP ProCurve 2650 (48 ports)**

Aceste echipamente sunt utilizate pentru elaborarea și implementarea soft a diferitor clase de algoritmi paraleli, efectuarea cursurilor de prelegeri și laborator la disciplinele corespunzătoare pentru ciclurile 1 și 2 de studii, precum și în cadrul diferitor proiecte de cercetare.

- **3 stații de management HP dx5150**
  - ✓ CPU: Athlon64 3200+

- ✓ RAM: 1GB PC3200 DDR
- ✓ Storage: 80GB SATA, DVD-RW
- ✓ Network: 1Gbps LAN
- ✓ Monitor: HP L1940 LCD

Aceste echipamente sunt utilizate pentru administrarea clusterului paralel şi a infrastructurii sale de la distanţă.

- **14 stații de lucru HP dx5150**
  - ✓ CPU: Athlon64 3200+
  - ✓ RAM: 512MB PC3200 DDR
  - ✓ HDD: 80GB SATA
  - ✓ Network: 1Gbps LAN
  - ✓ Monitor: HP L1706 LCD
- **Tablă interactivă (Smart board) și proiectoare.**

Aceste echipamente sunt utilizate drept stații de lucru pentru realizarea procesului didactic și de cercetare la Facultatea de Matematică și Informatică.

Clusterul paralel este înzestrat cu sisteme de calcul paralel licențiate sau „open source” necesare pentru realizarea procesului didactic și de cercetare necesar.

Notele de curs „Modele de programare paralelă. Partea 1. Programare MPI” îşi propun să acopere minimul de noţiuni necesare înţelegerii modalităților de implementare software a algoritmilor paraleli pe diverse sisteme paralele de calcul de tip cluster. Cursul va contribui esenţial la dezvoltarea aptitudinilor şi capacităţilor de construire, studiere a algoritmilor paraleli şi implementarea lor în diferite sisteme paralele de calcul. Drept rezultat al cunoştinţelor acumulate, studentul trebuie să poată aplica cele mai importante metode şi rezultate expuse în lucrare, pentru implementarea celor mai moderne realizări din domeniul informaticii şi tehnologiilor informaţionale.

Lucrarea dată acoperă obiectivele de referinţă şi unităţile de conţinut prezente în Curriculum la disciplina *Programare Paralelă*. Această disciplină, pe parcursul a mai mulţi ani, este predată la specialităţile *Informatica*, *Managmentul Informaţional* ale Facultăţii de Matematică şi Informatică şi la specialitatea *Tehnologii Informaţionale* a Facultăţii de Fizică şi Inginerie.

Această lucrare are drept scop dezvoltarea următoarelor competenţe generice şi specifice:

- cunoaşterea bazelor teoretice generale ale matematicii şi informaticii necesare la alcătuirea modelului problemei practice cu caracter socioeconomic;
- aplicarea de metode noi de cercetare şi implementare soft a algoritmilor paraleli;
- identificarea căilor şi a metodelor de utilizare a rezultatelor obţinute şi în alte domenii de activitate;
- elaborarea şi analiza algoritmilor paraleli de soluţionare a problemelor de o mare complexitate;
- implementarea metodelor noi şi concepţii moderne în realizarea lucrărilor proprii;
- posedarea diferitor abilităţi de analiză, sinteză şi evaluare în abordarea şi soluţionarea diferitor probleme;
- deţinerea capacităţilor şi a deprinderilor necesare pentru realizarea proiectelor de cercetare, demonstrând un grad înalt de autonomie.

Programele prezentate în „Exemple” au fost elaborate şi testate cu suportul studenţilor de la specialitatea „Informatică” a Facultății de Matematică și Informatică.

Cunoştinţele acumulate la acest curs vor fi utilizate şi la studierea cursului *Calcul paralel pe clustere* pentru studii de masterat.

La elaborarea acestui suport de curs au fost consultate sursele bibliografice prezente în „Bibliografie”.

CAPITOLUL 1. SISTEME ŞI MODELE DE CALCUL PARALEL

Obiective

- Să delimiteze corect clasele de sisteme paralele de calcul şi locul lor în problematica algoritmilor şi a programării paralele;
- Să definească noţiunea de algoritmi paraleli;
- Să poată construi un algoritm paralel în baza unui algoritm secvenţial.

1.1 Clasificarea sistemelor de calcul paralel

Calculul paralel, în înțelesul cel mai simplu, constă în **utilizarea concomitentă a unor resurse multiple** pentru a rezolva o problemă de calcul sau de gestionare a informației în forma electronică. Aceste resurse pot include următoarele componente:

- ✓ un singur calculator cu mai multe procesoare;
- ✓ un număr arbitrar de calculatoare conectate la o rețea;
- ✓ o combinație a acestora.

Problema de calcul trebuie să etaleze uzual caracteristici de genul:

- ✓ posibilitatea de a fi divizată în părți care pot fi rezolvate concomitent;
- ✓ posibilitatea de a fi executate instrucțiuni multiple în oricare moment;
- ✓ perspectiva de a fi rezolvată în timp mai scurt pe resurse de calcul multiple decât pe o resursă standard unică.

În cazul calculului secvențial analiza și studiul algoritmilor secvențiali este strâns legată de structura logică a calculatorului secvențial sau a mașinii de calcul abstracte, mașina Turing, de exemplu. Acest lucru este adevărat și în cazul calculului paralel. Iată de ce înainte de a trece la studierea metodelor de elaborare și implementare soft a algoritmilor paraleli vom prezenta acele arhitecturi ale calculatoarelor care corespund clasificării în baza taxonomiei lui Flynn.

Clasificarea clasică după Flynn

Există modalități diferite de a clasifica calculatoarele paralele. Una din cele mai folosite, în uz din 1966, este clasificarea lui Flynn. Clasificarea Flynn face distincție între arhitecturile calculatoarelor cu mai multe procesoare în raport cu două caracteristici independente, *instrucțiunile* și *datele*. Fiecare din aceste caracteristici pot avea una din două stări: unice

sau multiple. Tabelul de mai jos definește cele patru clase posibile potrivit criteriilor lui Flynn:

SISD Single instruction, single data (instrucțiune unică, date unice)	**SIMD** Single instruction, multiple data (instrucțiune unică, date multiple)
MISD Mutiple instruction, single data (instrucțiune multiplă, date unice)	**MIMD** Multiple instruction, multiple data (instrucțiune multiplă, date multiple)

Facem o succintă caracteristică a fiecărei dintre aceste clase.

Clasa SISD

Un calculator din această clasă constă dintr-o singură unitate de procesare care primește un singur șir de instrucțiuni ce operează asupra unui singur șir de date. La fiecare pas al execuției unitatea de control trimite o instrucție ce operează asupra unei date obținute din unitatea de memorie. De exemplu, instrucțiunea poate cere procesorului să execute o operație logică sau aritmetică asupra datei și depunerea rezultatului în memorie. Majoritatea calculatoarelor actuale folosesc acest model inventat de John von Neumann la sfârșitul anilor 40. Un algoritm ce lucrează pe un astfel de model se numește algoritm *secvențial* sau *serial*.

Clasa SIMD

Pentru acest model, calculatorul paralel constă din *N* procesoare identice. Fiecare procesor are propria sa memorie locală în care poate stoca programe sau date. Toate procesoarele lucrează sub controlul unui singur șir de instrucțiuni emise de o singură unitate centrală de control. Altfel spus, putem presupune că toate procesoarele păstrează o copie a programului de executat (unic) în memoria locală. Există *N* șiruri de date, câte unul pentru fiecare procesor. Toate procesoarele operează sincronizat: la fiecare pas, toate execută aceeași instrucțiune, fiecare folosind alte date de intrare (recepționate din memoria externă). O instrucțiune poate fi una simplă (de exemplu, o operație logică) sau una compusă (de exemplu, interclasarea a două liste). Analog, datele pot fi simple sau complexe. Uneori este necesar ca numai o parte din procesoare să execute instrucțiunea primită. Acest lucru poate fi codificat în instrucțiune, spunând procesorului dacă la pasul respectiv va fi activ (și execută instrucțiunea) sau

inactiv (și așteptă următoarea instrucțiune). Există un mecanism de genul unui „ceas global”, care asigură sincronizarea execuției instrucțiunilor. Intervalul dintre două instrucțiuni succesive poate fi fix sau să depindă de instrucțiunea ce urmează să se execute. În majoritatea problemelor ce se rezolvă pe acest model este necesar ca procesoarele să poată comunica între ele (pe parcursul derulării algoritmului) pentru a transmite rezultate intermediare sau date. Acest lucru se poate face în două moduri: o memorie comună (Shared Memory) sau folosind o rețea de conexiuni (Interconnection Network).

Shared Memory SIMD

Acest model este cunoscut în literatura de specialitate sub numele de **Modelul PRAM (Parallel Random-Access Machine model).** Cele *N* procesoare partajează o memorie comună, iar comunicarea între procesoare se face prin intermediul acestei memorii. Să presupunem că procesorul *i* vrea să comunice un număr procesorului *j*. Acest lucru se face în doi pași. Întâi procesorul *i* scrie numărul în memoria comună la o adresă cunoscută de procesorul *j*, apoi procesorul *j* citește numărul de la acea adresă. Pe parcursul execuției algoritmului paralel, cele *N* procesoare lucrează simultan cu memoria comună pentru a citi datele de intrare, a citi/scrie rezultate intermediare sau pentru a scrie rezultatele finale. Modelul de bază permite accesul simultan la memoria comună dacă datele ce urmează să fie citite/scrise sunt situate în locații distincte. Depinzând de modul de acces simultan la aceeași adresă în citire/scriere, modelele se pot divide în:

- Modelul **EREW SM SIMD (E**xclusive-**R**ead**, E**xclusive-**W**rite**).** Nu este permis accesul simultan a două procesoare la aceeași locație de memorie, nici pentru citire, nici pentru scriere.
- Modelul **CREW SM SIMD** (**C**oncurrent-**R**ead, **E**xclusive-**W**rite). Este permis accesul simultan a două procesoare la aceeași locație de memorie pentru citire, dar nu și pentru scriere. Este cazul cel mai natural și mai frecvent folosit.
- Modelul **ERCW SM SIMD** (**E**xclusive-**R**ead, **C**oncurrent-**W**rite). Este permis accesul simultan a două procesoare la aceeași locație de memorie pentru scriere, dar nu și pentru citire.
- Modelul **CRCW SM SIMD** (**C**oncurrent-**R**ead, **C**oncurrent-**W**rite). Este permis accesul simultan a două procesoare la aceeași locație de memorie pentru scriere sau pentru citire.

Citirea simultană de la aceeași adresă de memorie nu pune probleme, fiecare procesor poate primi o copie a datei de la adresa respectivă. Pentru scriere simultană, probabil a unor date diferite, rezultatul este imprevizibil. De aceea se adoptă o regulă de eliminare a conflictelor la scrierea simultană (atunci când modelul o permite). Exemple de asemenea reguli pot fi: se scrie procesorul cu numărul de identificare mai mic sau se scrie suma tuturor cantităților pe care fiecare procesor le are de scris, sau se atribuie priorități procesoarelor și se scrie procesorul cu cea mai mare prioritate etc.

Clasa MISD

În acest model *N* procesoare, fiecare cu unitatea sa de control, folosesc în comun aceeași unitate de memorie unde se găsesc datele de prelucrat. Sunt N șiruri de instrucțiuni și un singur șir de date. La fiecare pas, o dată primită din memoria comună este folosită simultan de procesoare, fiecare executând instrucția primită de la unitatea de control proprie. Astfel, paralelismul apare lăsând procesoarele să execute diferite acțiuni în același timp asupra aceleiași date. Acest model se potrivește rezolvării problemelor în care o intrare trebuie supusă mai multor operații, iar o operație folosește intrarea în forma inițială. De exemplu, în probleme de clasificare, este necesar să stabilim cărei clase (de obiecte) îi aparține un obiect dat. La analiza lexicală trebuie stabilită unitatea lexicală (clasa) căreia îi aparține un cuvânt din textul sursă. Aceasta revine la recunoașterea cuvântului folosind automate finite specifice fiecărei unități lexicale. O soluție eficientă ar fi în acest caz funcționarea simultană a automatelor folosite la recunoaștere, fiecare pe un procesor distinct, dar folosind același cuvânt de analizat.

Clasa MIMD

Acest model este cel mai general și cel mai puternic model de calcul paralel. Avem *N* procesoare, *N* șiruri de date și *N* șiruri de instrucțiuni. Fiecare procesor execută propriul program furnizat de propria sa unitate de control. Putem presupune că fiecare procesor execută programe diferite asupra datelor distincte în timpul rezolvării diverselor subprobleme în care a fost împărțită problema inițială. Aceasta înseamnă că procesoarele lucrează asincron. Ca și în cazul SIMD, comunicarea între procese are loc printr-o memorie comună sau cu ajutorul unei rețele de conexiuni. Calculatoarele MIMD cu memorie comună sunt denumite în mod uzual multiprocesoare (sau tightly coupled machines), iar cele ce folosesc rețele

de conexiuni se mai numesc multicomputere (sau loosely coupled machines).

Desigur că pentru multiprocesoare avem submodelele EREW, CREW, CRCW, ERCW. Uneori multicomputerele sunt denumite sisteme distribuite. De obicei se face distincție între denumiri după lungimea conexiunilor (dacă procesoarele sunt la distanță se folosește termenul de sistem distribuit).

1.2 Definiția algoritmilor paraleli

Vom introduce următoarea definiție a noțiunii de **algoritm paralel.**

Definiție 1.2.1 *Prin algoritm paralel vom înțelege un graf orientat și aciclic* $G = (V, E)$*, astfel încât pentru orice nod* $v \in V$ *există o aplicație* $\varphi_v : D^{\gamma(v)} \to D$*, unde* φ_v *denotă operația executată asupra spațiului de date D și* $\gamma(v)$ *denotă adâncimea nodului* $v \in V$*.*

Această definiție necesită următoarele explicații:

- Mulțimea de noduri *V* indică de fapt mulțimea de operații, și mulțimea de arce *E* indică fluxul informațional.
- Toate operațiile $\{\varphi_v\}_{v \in V}$ pentru nodurile *v* pentru care $\gamma(v)$ coincid (adică nodurile de la același nivel) se execută în paralel.
- Dacă $\gamma(v) = 0$, atunci operațiile $\varphi_v : D^0 \to D$ descriu nu altceva decât *atribuire de valori inițiale.* Mulțimea de noduri pentru care $\gamma(v) = 0$ se vor nota prin V_0.

Este clar că această definiție este formală și indică, de fapt, ce operații și asupra căror fluxuri de date se execută în paralel. Pentru implementarea algoritmilor paraleli este nevoie de o descriere mai detaliată, și aceasta se realizează prin intermediul unui program de implementare. Acest program la fel poate fi definit formal folosind un limbaj matematic prin intermediul unei *scheme a algoritmului.*

Definiție 1.2.2 *Fie* $G = (V, E)$ *cu mulțimea de noduri inițiale* V_0*. Schema (schedule) a algoritmului paralel va conține următoarele elemente:*

- *funcția de alocare a procesoarelor* $\pi: V \backslash V_0 \to N_p$*, unde* $Np = \{0,1, \dots, p-1\}$ *indică mulțimea de procesoare;*
- *funcția de alocare a timpului* $\tau: V \to N$*, unde N este timpul discret.*

Aceste elemente trebuie să verifice următoarele condiții:

i. *Dacă* $v \in V_0$*, rezultă că* $\tau(v) = 0$*.*

ii. *Dacă arcul* $(v, v') \in E$*, rezultă că* $\tau(v') \geq \tau(v) + 1$*.*

iii. *Pentru orie* $v, v' \in V \backslash V_0$ *și* $v \neq v', \tau(v) = \tau(v')$, *rezultă că* $\pi(v) \neq \pi(v')$.

În continuare vom nota schema algoritmului paralel prin (π, τ).

Notăm prin ***P*** problema ce urmează a fi rezolvată și prin ***D*** volumul de date necesar pentru rezolvarea problemei. Din definițiile prezentate rezultă că pentru elaborarea și implementarea unui algoritm paralel trebuie:

A) Să se realizeze *paralelizarea la nivel de date și operații*: problema ***P*** se divizează într-un număr finit de subprobleme $\{P_i\}_{i\prime 1}^{n}$ și volumul de date ***D*** se divizează în $\{D_i\}_{i\prime 1}^{n}$. Cu alte cuvinte, se construieşte graful G al algoritmului paralel.

B) Să se realizeze *distribuirea calculelor*: în baza unor algoritmi secvențiali bine cunoscuți, pe un sistem de calcul paralel cu *n* procesoare, în paralel (sau concurent) se rezolvă fiecare subproblemă P_i cu volumul său de date D_i.

C) În baza rezultatelor subproblemelor obținute la etapa B) se construiește soluția problemei inițiale ***P***.

Parcurgerea acestor trei etape de bază este inevitabilă pentru construirea oricărui algoritm paralel și implementarea soft pe calculatoare de tip cluster.

CAPITOLUL 2. MODELE DE PROGRAMARE PARALELĂ

Obiective

- Să definească noţiunea de model de programare paralelă;
- Să cunoască criteriile de clasificare a sistemelor paralele de calcul;
- Să cunoască particularităţile de bază la elaborarea programelor paralele ale sistemelor de calcul cu memorie partajată, cu memorie distribuită şi mixte;
- Să definească noţiunea de secvenţă critică şi să poată utiliza în programe paralele astfel de secvenţe.

2.1 Sisteme de calcul cu memorie distribuită și partajată

Model de programare este un set de abilități de programare (de elaborare a programelor) utilizate pentru un calculator abstract cu o arhitectură data. Astfel, modelul de programare este determinat de structura logică a calculatorului, de arhitectura lui. În prezenta lucrare vor fi studiate următoarele modele de programare paralelă:

- modele de programare paralelă cu memorie distribuită (modele bazate pe programarea MPI) [în Partea 1 a lucrării];
- modele de programare paralelă cu memorie partajată (modele bazate pe programarea OpenMP) [în Partea 2 a lucrării];
- modele de programare paralelă mixte (modele bazate atât pe programarea MPI cât și pe programarea OpenMP) [în Partea 2 a lucrării].

Vom descrie mai jos particularitățile de bază ale acestor modele.

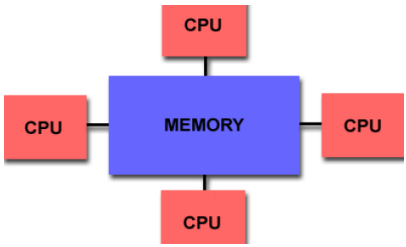
Calcul paralel este numită execuția în paralel pe mai multe procesoare a aceloraşi instrucțiuni sau și a unor instrucțiuni diferite, cu scopul rezolvării mai rapide a unei probleme, de obicei special adaptată sau subdivizată. Ideea de bază este aceea că problemele de rezolvat pot fi uneori împărțite în mai multe probleme mai simple, de natură similară sau identică între ele, care pot fi rezolvate simultan. Rezultatul problemei inițiale se află apoi cu ajutorul unei anumite coordonări a rezultatelor parțiale.

Calculul distribuit (din engleză Distributed computing) este un domeniu al informaticii care se ocupă de sisteme distribuite. Un sistem distribuit este format din mai multe calculatoare autonome care comunică printr-o rețea. Calculatoarele interacționează între ele pentru atingerea unui

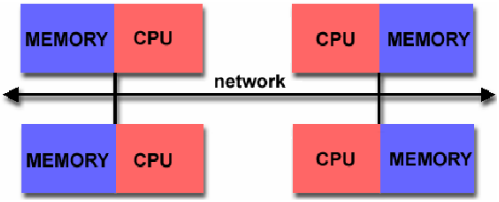
scop comun. Un program care rulează într-un sistem distribuit se numește program distribuit, iar procesul de scriere a astfel de programe se numește programare distribuită.

Sistemele distribuite sunt calculatoarele dintr-o rețea ce operează cu aceleași procesoare. Termenii de *calcul concurent, calcul paralel* și *calcul distribuit* au foarte multe în comun. Același sistem poate fi caracterizat ca fiind atât „paralel" cât și „distribuit" și procesele dintr-un sistem distribuit tipic rulează în paralel. Sistemele concurente pot fi clasificate ca fiind paralele sau distribuite în funcție de următoarele criterii:

- În calcul paralel (memorie partajată – Shared Memory Model (SMM)) toate procesoarele au acces la o memorie partajată. Memoria partajată poate fi utilizată pentru schimbul de informații între procesoare. Schematic sistemele de tip SMM pot fi reprezentate astfel:



- În calcul distribuit (memorie distribuită – Distributed Memory Model (DMM)) fiecare procesor are memorie proprie (memorie distribuită). Schimbul de informații are loc prin trimiterea de mesaje între procesoare. Schematic sistemele de tip DMM pot fi reprezentate astfel:



2.2 Particularități de bază ale modelelor de programare paralelă cu memorie partajată

Regiuni critice

O regiune critică este o zonă de cod care în urma unor secvențe de acces diferite poate genera rezultate diferite. Entitățile de acces sunt procesele sau thread-urile. O regiune critică se poate referi, de asemenea, la o dată sau la un set de date, care pot fi accesate de mai multe procese/thread-uri.

O regiune critică la care accesul nu este sincronizat (adică nu se poate garanta o anumită secvență de acces) generează o așa-numită condiție de cursă (race condition). Apariția unei astfel de condiții poate genera rezultate diferite în contexte similare.

Un exemplu de regiune critică este secvența de pseudocod:

move $r1, a valoarea **a** este adusă într-un registru,
$r1 <- $r1 + 1 valoarea registrului este incrementată,
write a, $r1 valoarea din registru este rescrisă în memorie.

Presupunem că avem două procese care accesează această regiune (procesul A și procesul B). Secvența probabilă de urmat este:

A: **move $r1, a**
A: **$r1 <- $r1 + 1**
A: **write a, $r1**
B: **read $r1**
B: **$r1 <- $r1 + 1**
B: **write a, $r1**

Dacă **a** are inițial valoarea 5, se va obține în final valoarea dorită 7.

Totuși, această secvență nu este garantată. Cuanta de timp a unui proces poate expira chiar în mijlocul unei operații, iar planificatorul de procese va alege alt proces. În acest caz, o secvență posibilă ar fi:

A: **move $r1,a**	**a** este 5
A: **$r1<-$r1+1**	în memorie **a** este încă 5, **r1** ia valoarea 6
B: **move $r1,a**	**a** este 5, în **r1** se pune valoarea 5
B: **$r1<-$r1+1**	**r1** devine 6
B: **move a,$r1**	se scrie 6 în memorie
A: **move a,$r1**	se scrie 6 în memorie

Se observă că pentru valoarea inițială 5 se obține valoarea 6, diferită de valoarea dorită.

Pentru a realiza sincronizarea accesului la regiuni critice va trebui să permitem accesul unei singure instanțe de execuție (thread, proces). Este necesar un mecanism care să forțeze o instanță de execuție sau mai multe să aștepte la intrarea în regiunea critică, în situația în care un thread/proces are deja acces. Situația este similară unui ghișeu sau unui cabinet. O persoană are permis accesul și, cât timp realizează acel lucru, celelalte așteaptă. În momentul în care instanța de execuție părăsește regiunea critică, o altă instanță de execuție capătă accesul.

Excluderea reciprocă

Din considerente practice s-a ajuns la necesitatea ca, în contexte concrete, o secvență de instrucțiuni din programul de cod al unui proces să poată fi considerată ca fiind indivizibilă. Prin proprietatea de a fi indivizibilă înțelegem că odată începută execuția instrucțiunilor din această secvență, nu se va mai executa vreo acțiune dintr-un alt proces până la terminarea execuției instrucțiunilor din acea secvență. Aici trebuie să subliniem un aspect fundamental, și anume că această secvență de instrucțiuni este o entitate logică ce este accesată exclusiv de unul dintre procesele curente. O astfel de secvență de instrucțiuni este o **secțiune critică logică**. Deoarece procesul care a intrat în secțiunea sa critică exclude de la execuție orice alt proces curent, rezultă că are loc o **excludere reciprocă (mutuală)** a proceselor între ele. Dacă execuția secțiunii critice a unui proces impune existenţa unei zone de memorie pe care acel proces să o monopolizeze pe durata execuției secțiunii sale critice, atunci acea zonă de memorie devine o **secțiune critică fizică** sau **resursă critică**. De cele mai multe ori, o astfel de secțiune critică fizică este o zonă comună de date partajate de mai multe procese.

Problema excluderii mutuale este legată de proprietatea secțiunii critice de a fi atomică, indivizibilă la nivelul de bază al execuției unui proces. Această atomicitate se poate atinge fie valorificând facilitățile primitivelor limbajului de programare gazdă, fie prin mecanisme controlate direct de programator (din această a doua categorie fac parte algoritmii specifici scriși pentru rezolvarea anumitor probleme concurente).

Problema excluderii mutuale apare deoarece în fiecare moment **cel mult** un proces poate să se afle în secțiunea lui critică. Excluderea mutuală este o primă formă de sincronizare, ca o alternativă pentru sincronizarea pe condiție.

Problema excluderii mutuale este una centrală în contextul programării paralele. Rezolvarea problemei excluderii mutuale depinde de îndeplinirea a trei cerințe fundamentale, și anume:

1) **excluderea reciprocă propriu-zisă**, care constă în faptul că la fiecare moment cel mult un proces se află în secțiunea sa critică;
2) **competiția constructivă, neantagonistă**, care se exprimă astfel: dacă niciun proces nu este în secțiunea critică și dacă există procese care doresc să intre în secțiunile lor critice, atunci unul dintre acestea va intra efectiv;
3) **conexiunea liberă între procese**, care se exprimă astfel: dacă un proces „întârzie" în secțiunea sa necritică, atunci această situație nu trebuie să împiedice alt proces să intre în secțiunea sa critică (dacă dorește acest lucru).

Îndeplinirea condiției a doua asigură că procesele nu se împiedică unul pe altul să intre în secțiunea critică și nici nu se invită la nesfârșit unul pe altul să intre în secțiunile critice. În condiția a treia, dacă „întârzierea" este definitivă, aceasta nu trebuie să blocheze întregul sistem (cu alte cuvinte, blocarea locală nu trebuie să antreneze automat blocarea globală a sistemului).

Excluderea reciprocă (mutuală) fiind un concept central, esențial al concurenței, în paragrafele următoare vom reveni asupra rezolvării acestei probleme la diferite nivele de abstractizare și folosind diferite primitive. Principalele abordări sunt:

1) **folosind suportul hardware** al sistemului gazdă: rezolvarea excluderii mutuale se face cu task-uri; aceste primitive testează și setează variabile booleene (o astfel de variabilă este asociată unei resurse critice); un task nu este o operație atomică;
2) **folosind suportul software**: rezolvarea excluderii mutuale se bazează pe facilităţile oferite de limbajele de programare pentru comunicarea și sincronizarea proceselor executate nesecvențial (paralel sau concurent).

CAPITOLUL 3. MODELUL MPI DE PROGRAMARE PARALELĂ PE SISTEME DE CALCUL CU MEMORIE DISTRIBUITĂ

Obiective

- Să cunoască noţiunile de „standard MPI”, „programe MPI”, să poată paraleliza un cod secvenţial al unui program în limbajul C++;
- Să cunoască sintaxa și modul de utilizare a funcţiilor MPI;
- Să elaboreze programe MPI în limbajul C++ pentru implementarea diferitor algoritmi paraleli pe sisteme de calcul paralel cu memorie distribuită;
- Să poată utiliza comenzile sistemului ROCKS pentru lansarea în execuție şi monitorizarea programelor MPI pe sisteme paralele de tip cluster.

3.1 Generalități

MPI este un standard pentru comunicarea prin mesaje, elaborat de MPI Forum. În definirea lui au fost utilizate caracteristicile cele mai importante ale unor sisteme anterioare, bazate pe comunicația de mesaje. A fost valorificată experiența de la IBM, Intel (NX/2) Express, nCUBE (Vertex), PARMACS, Zipcode, Chimp, PVM, Chameleon, PICL. MPI are la bază modelul proceselor comunicante prin mesaje: un calcul este o colecție de procese secvențiale care cooperează prin comunicare de mesaje. MPI este o bibliotecă, nu un limbaj. El specifică convenții de apel pentru mai multe limbaje de programare: C, C++, FORTRAN.77, FORTRAN90, Java.

MPI a ajuns la versiunea 3, trecând succesiv prin versiunile:

- ✓ MPI 1 (1993), un prim document, orientat pe comunicarea punct la punct;
- ✓ MPI 1.0 (iunie 1994) este versiunea finală, adoptată ca standard; include comunicarea punct la punct şi comunicarea colectivă;
- ✓ MPI 1.1 (1995) conține corecții și extensii ale documentului inițial din 1994, modificările fiind univoce;
- ✓ MPI 1.2 (1997) conține extensii şi clarificări pentru MPI 1.1;
- ✓ MPI 2 (1997) include funcționalități noi:
  - procese dinamice;
  - comunicarea „one-sided”;
  - operații de I/E (utilizarea fişierelor) paralele.

Obiectivele MPI:

- proiectarea unei interfețe de programare a aplicațiilor;
- comunicare eficientă;
- să fie utilizabil în medii eterogene;
- portabilitate;
- interfața de comunicare să fie sigură (erorile tratate dedesubt);
- apropiere de practici curente (PVM, NX, Express, p4;
- semantica interfeței să fie independentă de limbaj.

Astfel, MPI este o bibliotecă de funcții, care permite realizarea interacțiunii proceselor paralele prin mecanismul de transmitere a mesajelor. Este o bibliotecă complexă, formată din aproximativ 130 de funcții care includ:

- funcții de inițializare și închidere (lichidare) a proceselor MPI;
- funcții care implementează operațiunile de comunicare de tip „proces-proces”;
- funcții care implementează operațiunile colective;
- funcții pentru lucrul cu grupuri de procese și comunicatori;
- funcții pentru lucrul cu structuri de date;
- funcția care implementează generarea dinamică a proceselor;
- funcții care implementează comunicări unidirecționale între procese (accesul la distanță a memoriei);
- funcții de lucru cu fișierele.

Fiecare dintre funcțiile MPI este caracterizată prin metoda de realizare (executare):

- funcție locală – se realizează în cadrul procesului care face apel la ea, la finalizarea ei nu este nevoie de transmitere de mesaje;
- funcție nelocală – pentru finalizarea ei este necesară utilizarea unor proceduri MPI, executate de alte procese;
- funcție globală – procedura trebuie să fie executată de toate procesele grupului;
- funcție de blocare;
- funcție de non-blocare.

În tabelul de mai jos se arată corespondența dintre tipurile predefinite de date MPI și tipurile de date în limbajul de programare.

Tip de date MPI	**Tip de date C**
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_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

3.2 Funcțiile MPI de bază

Fiecare program MPI trebuie să conțină un apel al operației **MPI_Init.** Rolul acesteia este de a inițializa „mediul" în care programul va fi executat. Ea trebuie executată o singură dată, înaintea altor operații MPI. Pentru a evita execuția sa de mai multe ori (și deci provocarea unei erori), MPI oferă posibilitatea verificării dacă **MPI_Init** a fost sau nu executată. Acest lucru este făcut prin funcţia **MPI_Iniatialized.** În limbajul C funcția are următorul prototip:

```
int MPI_Initialized(int *flag);
```

unde

OUT **flag** – este true dacă **MPI_Init** a fost deja executată.

Aceasta este, de altfel, singura operație ce poate fi executată înainte de **MPI_Init**.

Funcția *MPI_Init*

Forma generală a funcției este

```
int MPI_Init(int *argc, char ***argv);
```

Funcția de inițializare acceptă ca argumente **argc** și **argv,** argumente ale funcției **main,** a căror utilizare nu este fixată de standard și depinde de implementare. Ca rezultat al executării acestei funcții se creează

un grup de procese, în care se includ toate procesele generate de utilitarul **mpirun**, și se creează pentru acest grup un mediu virtual de comunicare descris de comunicatorul cu numele **MPI_COMM_WORLD**. Procesele din grup sunt numerotate de la *0* la *groupsize-1*, unde *groupsize* este egal cu numărul de procese din grup. La fel se creează comunicatorul cu numele **MPI_COMM_SELF**, care descrie mediul de comunicare pentru fiecare proces în parte.

Perechea funcției de inițializare este **MPI_Finalize**, care trebuie executată de fiecare proces, pentru a „închide" mediul MPI. Forma acestei operații este următoarea:

```
int MPI_Finalize(void).
```

Utilizatorul trebuie să se asigure că toate comunicațiile în curs de desfășurare s-au terminat înainte de apelul operației de finalizare. După **MPI_Finalize**, nici o altă operație MPI nu mai poate fi executată (nici măcar una de inițializare).

Funcția *MPI_Comm_size*

În limbajul C funcția are următorul prototip:

```
int MPI_Comm_size(MPI_Comm comm, int *size),
```

unde

IN **comm** – nume comunicator,
OUT **size** – numărul de procese ale comunicatorului **comm**.

Funcția returnează numărul de procese MPI ale mediului de comunicare cu numele **comm**. Funcția este locală.

Funcția *MPI_Comm_rank*

În limbajul C funcția are următorul prototip:

```
int MPI_Comm_rank(MPI_Comm comm, int *rank),
```

unde

IN **comm** – nume comunicator,
OUT **rank** – identificatorul procesului din comunicatorul **comm.**

Un proces poate afla poziția sa în grupul asociat unui comunicator prin apelul funcției **MPI_Comm_rank.** Funcția returnează un număr din diapazonul `0,..,size-1`, care semnifică identificatorul procesului MPI care a executat această funcție. Astfel proceselor li se atribuie un număr în funcție de ordinea în care a fost executată funcția. Această funcție este

locală.

Vom exemplifica utilizarea funcțiilor MPI descrise mai sus.

Exemplul 3.2.1. *Să se elaboreze un program MPI în care fiecare proces tipărește rankul său și numele nodului unde se execută.*

Mai jos este prezentat codul programului în limbajul C++ în care se realizează condițiile enunțate în exemplu 3.2.1.

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int main(int argc,char *argv[])
{
int size,rank,namelen;
int local_rank =
        atoi(getenv("OMPI_COMM_WORLD_L
        OCAL_RANK"));
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);
if (rank ==0)
printf("\n=====REZULTATUL PROGRAMULUI
        '%s'\n", argv[0]);
MPI_Barrier(MPI_COMM_WORLD);
MPI_Get_processor_name(processor_name
        , &namelen);
if (local_rank == 0) {
printf("==Au fost generate %s MPI processes
        pe nodul %s ===\n",
        getenv("OMPI_COMM_WORLD_LOCA
        L_SIZE"), processor_name); }
MPI_Barrier(MPI_COMM_WORLD);
if (rank ==0)
printf("==Procesele comunicatorului
        MPI_COMM_WORLD au fost
        'distribuite' pe noduri astfel: \n");
MPI_Barrier(MPI_COMM_WORLD);
printf("Hello, I am the process number %d
        (local runk %d) on the compute
        hostname %s,from total number of
        process %d\n",rank,
        local_rank,processor_name, size);
MPI_Finalize();
return 0;
}
```

Rezultatele posibile ale executării programului:

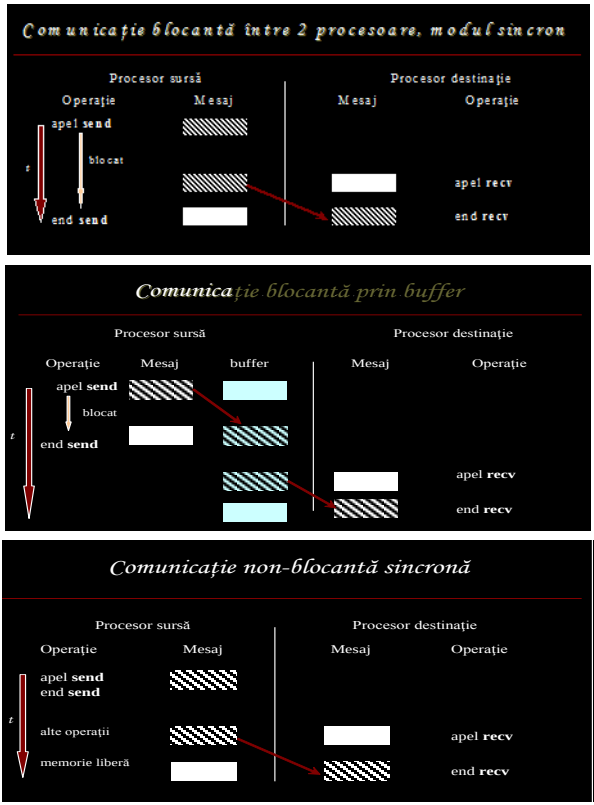
```
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpiCC -o Exemplu_3_2_1.exe Exemplu_3_2_1.cpp
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpirun -n 7 -machinefile ~/nodes
Exemplu_3_2_1.exe
=====REZULTATUL PROGRAMULUI 'Exemplu_3_2_1.exe'
==Au fost generate 4 MPI processes pe nodul compute-0-0.local ===
==Au fost generate 3 MPI processes pe nodul compute-0-1.local ===
==Procesele comunicatorului MPI_COMM_WORLD au fost 'distribuite' pe noduri astfel:
Hello, I am the process number 0 (local runk 0) on the compute hostname compute-0-0.local , from
total number of process 7
Hello, I am the process number 3 (local runk 3) on the compute hostname compute-0-0.local , from
total number of process 7
Hello, I am the process number 2 (local runk 2) on the compute hostname compute-0-0.local , from
total number of process 7
Hello, I am the process number 4 (local runk 0) on the compute hostname compute-0-1.local , from
total number of process 7
Hello, I am the process number 5 (local runk 1) on the compute hostname compute-0-1.local , from
total number of process 7
```

```
Hello, I am the process number 1 (local runk 1) on the compute hostname compute-0-0.local , from total number of process 7
Hello, I am the process number 6 (local runk 2) on the compute hostname compute-0-1.local , from total number of process 7
[Hancu_B_S@hpc]$
```

3.3 Comunicarea de tip „proces-proces” (punct la punct)

3.3.1 Operații de bază

Operațiile de bază pentru comunicarea punct la punct sunt **send** și **receive.** Modalitățile de transmitere a mesajelor pot fi explicate prin următoarele desene:



Considerăm mai întâi operația de transmitere de mesaje, în forma sa „uzuală”:

send(adresa,lungime,destinatie,tip)

unde

• **adresa** identifică începutul unei zone de memorie unde se află mesajul de transmis,

• **lungime** este lungimea în octeți a mesajului,

• **destinatie** este identificatorul procesului căruia i se trimite mesajul (uzual un număr întreg),

• **tip(flag)** este un întreg ne-negativ care restricționează recepția mesajului. Acest argument permite programatorului să rearanjeze mesajele în ordine, chiar dacă ele nu sosesc în secvența dorită. Acest set de parametri este un bun compromis între ceea ce utilizatorul dorește și ceea ce sistemul poate să ofere: transferul eficient al unei zone contigue de memorie de la un proces la altul. În particular, sistemul oferă mecanismele de păstrare în coadă a mesajelor, astfel că o operație de recepție **recv(adresa,maxlung, sursa,tip,actlung)** se execută cu succes doar dacă mesajul are tipul corect. Mesajele care nu corespund așteaptă în coadă. În cele mai multe sisteme, sursa este un argument de ieșire, care indică originea mesajului.

3.3.2 Funcţii MPI pentru transmiterea mesajelor

Funcția *MPI_Send*

În limbajul C această funcție are următorul prototip, valorile returnate reprezentând coduri de eroare:

```
int MPI_Send(void *buf,int count, MPI_Datatype
    datatype,int dest,int tag, MPI_Comm comm),
```

unde parametrii sunt:

IN	**buf**	– adresa inițială a tamponului sursă;
IN	**count**	– numărul de elemente (întreg ne-negativ);
IN	**datatype**	– tipul fiecărui element;
IN	**dest**	– numărul de ordine al destinatarului (întreg);
IN	**tag**	– tipul mesajului (întreg);
IN	**comm**	– comunicatorul implicat.

Astfel, procesul din mediul de comunicare **comm**, care execută funcția **MPI_Send**, expediază procesului **dest, count** date de tip **datatype** din memoria sa începând cu adresa **buf.** Acestor date li se atribuie eticheta **tag.**

Funcția *MPI_Recv*

Prototipul acestei funcții în limbajul C este

```
int MPI_Recv(void *buf,int count, MPI_Datatype
    datatype,int source,int tag,MPI_Comm comm,
    MPI_Status *status),
```

unde parametrii sunt:

OUT **buf**	– adresa inițială a tamponului destinatar;
IN **count**	– numărul de elemente (întreg ne-negativ);
IN **datatype**	– tipul fiecărui element;
IN **source**	– numărul de ordine al sursei (întreg);
IN **tag**	– tipul mesajului (întreg);
IN **comm**	– comunicatorul implicat;
OUT **status**	– starea, element (structură) ce indică sursa, tipul și contorul mesajului efectiv primit.

Astfel, procesul din mediul de comunicare **comm**, care execută funcția **MPI_Recv**, recepționează de la procesul **source, count** date de tip **datatype** care au eticheta **tag** și le salvează în memoria sa începând cu adresa **buf.**

Operația **send** este blocantă. Ea nu redă controlul apelantului până când mesajul nu a fost preluat din tamponul sursă, acesta din urmă putând fi reutilizat de transmițător. Mesajul poate fi copiat direct în tamponul destinatar (dacă se execută o operație **recv**) sau poate fi salvat într-un tampon temporar al sistemului. Memorarea temporară a mesajului decuplează operațiile **send** și **recv**, dar este consumatoare de resurse. Alegerea unei variante aparține implementatorului MPI. Și operația **recv** este blocantă: controlul este redat programului apelant doar când mesajul a fost recepționat.

Funcția *MPI_Sendrecv*

În situațiile în care se dorește realizarea unui schimb reciproc de date între procese, mai sigur este de a utiliza o funcție combinată **MPI_Sendrecv**.

Prototipul acestei funcții în limbajul C este

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
    MPI_Datatype sendtype, int dest, int
```

```
    sendtag, void *recvbuf, int recvcount,
    MPI_Datatype recvtype, int source,
    MPI_Datatypea recvtag, MPI_Comm comm,
    MPI_Status *status),
```

unde parametrii sunt:

IN **sendbuf**	– adresa inițială a tamponului sursă;
IN **sendcount**	– numărul de elemente transmise (întreg ne-negativ);
IN **sendtype**	– tipul fiecărui element trimis;
IN **dest**	– numărul de ordine al destinatarului (întreg);
IN **sendtag**	– identificatorul mesajului trimis;
OUT **recvbuf**	– adresa inițială a tamponului destinatar;
IN **recvcount**	– numărul maximal de elemente primite;
IN **recvtype**	– tipul fiecărui element primit;
IN **source**	– numărul de ordine al sursei (întreg);
IN **recvtag**	– identificatorul mesajului primit;
IN **comm**	– comunicatorul implicat;
OUT **status**	– starea, element (structura) ce indică sursa, tipul și contorul mesajului efectiv primit.

Există și alte funcții MPI care permit programatorului să controleze modul de comunicare de tip proces-proces. În tabelul de mai jos prezentăm aceste funcții.

Funcțiile de transmitere a mesajelor de tip „proces-proces”		
Modul de transmitere	**Cu blocare**	**Cu non-blocare**
Transmitere standart	MPI_Send	MPI_Isend
Transmitere sincron	MPI_Ssend	MPI_Issend
Transmitere prin tampon	MPI_Bsend	MPI_Ibsend
Transmitere coordonată	MPI_Rsend	MPI_Irsend
Recepționare mesaje	MPI_Recv	MPI_Irecv

Vom ilustra utilizarea funcțiilor **MPI_Send** și **MPI_Recv** prin următorul exemplu.

Exemplul 3.3.1 *Să se elaboreze un program MPI în limbajul C++ în care se realizează transmiterea mesajelor pe cerc începând cu procesul*

„început”. Valoarea variabilei „început” este inițializată de toate procesele și se află în diapazonul 0,…,size-1.

Mai jos este prezentat codul programului în care se realizează condițiile enunțate în exemplul 3.3.1.

```
#include<stdio.h>
#include <iostream>
#include<mpi.h>
using namespace std;
int main(int argc,char *argv[])
{
int size,rank,t,namelen;
int incep=8;
char
        processor_name[MPI_MAX_PROCESS
        OR_NAME];
MPI_Status status;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&siz
        e);
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]);
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,1
        0,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 Finale]$ /opt/openmpi/bin/mpiCC -o Exemplu_3_3_1.exe
Exemplu_3_3_1.cpp
[Hancu_B_S@hpc Finale]$ /opt/openmpi/bin/mpirun -n 9 -machinefile ~/nodes
Exemplu_3_3_1.exe

=====REZULTATUL PROGRAMULUI '  Exemplu_3_3_1.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
[Hancu_B_S@hpc Finale]$
```

3.3.3 Exerciţii

1. Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++ în care se realizează transmiterea mesajelor pe cerc în direcţia acelor de ceas, începând cu procesul „încep”, între procesele cu rankul par. Valoarea variabilei „încep” este iniţializată de toate procesele şi se află în diapazonul 0,…,size-1.
2. Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++ în care se realizează transmiterea mesajelor pe cerc în direcţia inversă acelor de ceas, începând cu procesul „încep”, între procesele cu rankul impar. Valoarea variabilei „încep” este iniţializată de toate procesele şi se află în diapazonul 0,…,size-1.
3. Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++ folosind funcţia **`MPI_Sendrecv`** în care se realizează transmiterea mesajelor pe cerc începând cu procesul „încep”. Valoarea variabilei „încep” este iniţializată de toate procesele şi se află în diapazonul *0,...,size-1*. Comparaţi timpul de execuţie al programului elaborat cu timpul de execuţie al programului din Exemplu 3.3.1. Pentru determinarea timpului de execuţie se poate utiliza funcţia **`MPI_Wtime()`**.
4. Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++ în care procesul cu rankul *0*, utilizând funcţiile **`MPI_Send`** şi **`MPI_Recv`**, transmite un mesaj tuturor proceselor din comunicatorul **`MPI_COMM_WORLD`**.
5. Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++ în care procesul cu rankul *0*, utilizând funcţia **`MPI_Sendrecv`**, transmite un mesaj tuturor proceselor din comunicatorul **`MPI_COMM_WORLD`**.

3.4 Funcţiile MPI pentru comunicarea colectivă

Operaţiile colective implică un grup de procese. Pentru execuţie, operaţia colectivă trebuie apelată de toate procesele, cu argumente corespondente. Procesele implicate sunt definite de argumentul comunicator, care precizează totodată contextul operaţiei. Multe operaţii colective, cum este difuzarea, presupun existenţa unui proces deosebit de celelalte, aflat la originea transmiterii sau recepţiei. 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 operaţie 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 faţă 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 tipărește procesul cu rankul *0*, după care tipărește procesul cu rankul *1*, ș.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

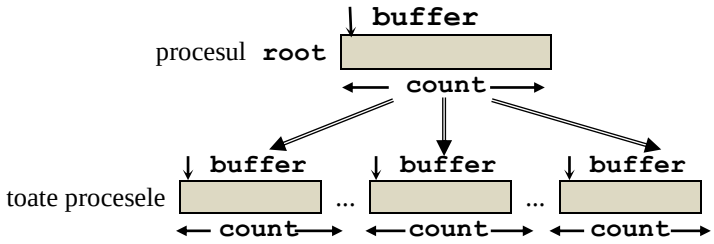
Trimite un mesaj de la procesul cu rankul „root” către toate celelalte procese ale comunicatorului. Prototipul acestei funcții în limbajul C este

```
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 3.4.1 *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_PROCESS
      OR_NAME];
 MPI_Status status;
 MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&siz
      e);
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,1
       0,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
[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
[Hancu_B_S@hpc]$
```

3.4.1 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 creşterii 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**);

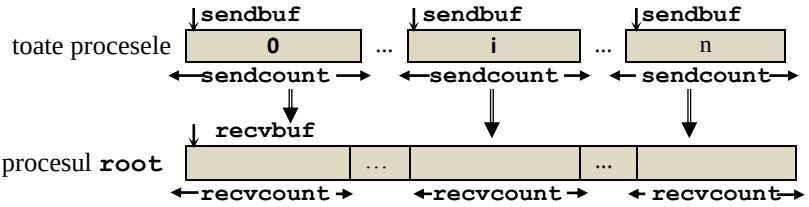
IN **recvtype** – tipul fiecărui element primit (este utilizat numai de procesul **root**);

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**);

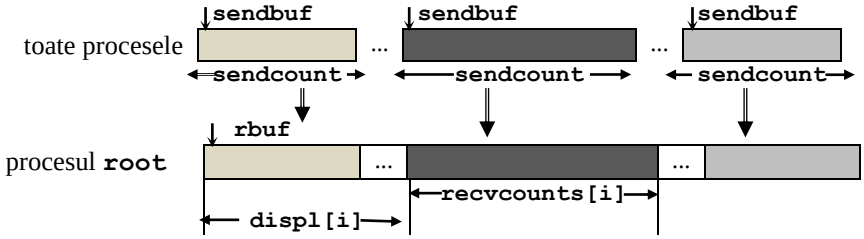
IN **recvtype** – tipul fiecărui element primit (este utilizat numai de procesul **root**);

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.

3.4.2 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;

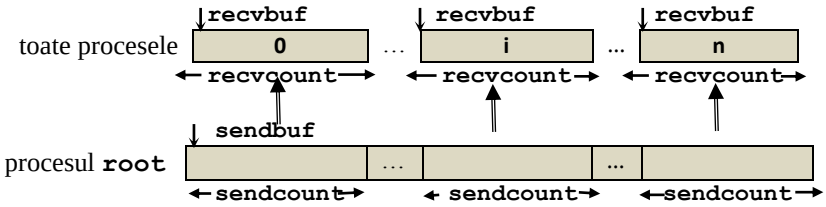
IN **recvtype** – tipul fiecărui element recepționat de fiecare proces;

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;

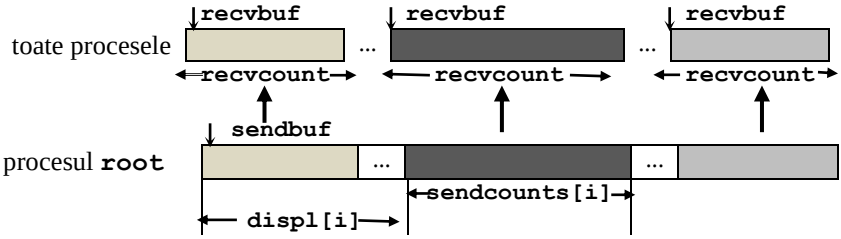
IN **recvcount** – numărul de elemente care sunt recepționate;

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 3.4.2 *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 3.4.3 *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 3.4.3.

```
#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]$
```

3.4.3 Funcțiile MPI pentru operații de reducere

În programarea paralelă operațiile matematice pe blocuri de date care sunt distribuite între procesoare se numesc *operații globale de reducere*. O operație de acumulare se mai numește și operație globală de reducere. Fiecare proces pune la dispoziție un bloc de date, care sunt combinate cu o operație binară de reducere; rezultatul acumulat este colectat la procesul

root. În MPI, o operațiune globală de reducere este reprezentată în următoarele moduri:

- menținerea rezultatelor în spațiul de adrese al unui singur proces (funcția **MPI_Reduce**);
- menținerea rezultatelor în spațiul de adrese al tuturor proceselor (funcția **MPI_Allreduce**);
- operația de reducere prefix, care în calitate de rezultat returnează un vector al cărui componentă *i* este rezultatul operației de reducere ale primelor *i* componente din vectorul distribuit (funcția **MPI_Scan**).

Funcția **MPI_Reduce** se execută astfel. Operația globală de reducere indicată prin specificarea parametrului **op**, se realizează pe primele elemente ale tamponului de intrare, iar rezultatul este trimis în primul element al tamponului de recepționare al procesului **root**. Același lucru se repetă pentru următoarele elemente din memoria tampon etc. Prototipul acestei funcții în limbajul C este:

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int
    count, MPI_Datatype datatype, MPI_Op op, int
    root, MPI_Comm comm)
```

unde

IN **sendbuf** – adresa inițială a tamponului pentru datele de intrare;

OUT **recvbuf** – adresa inițială a tamponului pentru rezultate (se utilizează numai de procesul **root**);

IN **count** – numărul de elemente în tamponul de intrare;

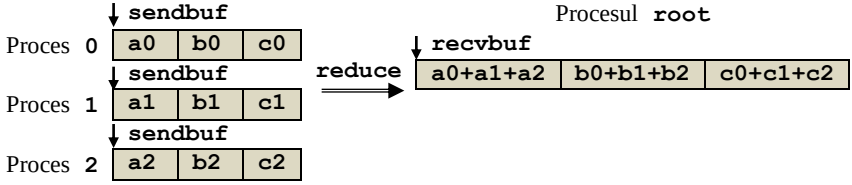
IN **datatype** – tipul fiecărui element din tamponul de intrare;

IN **op** – operația de reducere;

IN **root** – numărul procesului care recepționează rezultatele operației de reducere;

IN **comm** – comunicatorul implicat.

Grafic această funcție poate fi interpretată astfel:



În acest desen operația "+" semnifică orice operație admisibilă de reducere. În calitate de operație de reducere se poate utiliza orice operație predefinită sau operații determinate (construite) de utilizator folosind funcția **MPI_Op_create.**

În tabelul de mai jos sunt prezentate operațiile predefinite care pot fi utilizate în funcția **MPI_Reduce**.

Nume	**Operația**	**Tipuri de date admisibile**
MPI_MAX **MPI_MIN**	Maximum Minimum	integer, floating point
MPI_SUM **MPI_PROD**	Suma Produsul	integer,floating point, complex
MPI_LAND **MPI_LOR** **MPI_LXOR**	AND OR excludere OR	integer, logical
MPI_BAND **MPI_BOR** **MPI_BXOR**	AND OR excludere OR	integer, byte
MPI_MAXLOC **MPI_MINLOC**	Valoarea maximală și indicele Valoarea minimală și indicele	Tip special de date

În tabelul de mai sus pentru tipuri de date se utilizează următoarele notații:

integer	**MPI_INT, MPI_LONG, MPI_SHORT, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG**
floating point	**MPI_FLOAT, MPI_DOUBLE, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_LONG_DOUBLE**
logical	**MPI_LOGICAL**
complex	**MPI_COMPLEX**
byte	**MPI_BYTE**

Operațiile **MAXLOC** și **MINLOC** se execută pe un tip special de date fiecare element conținând două valori: valoarea maximului sau minimului și indicele elementului. În MPI există 9 astfel de tipuri predefinite.

MPI_FLOAT_INT	float and int
MPI_DOUBLE_INT	double and int
MPI_LONG_INT	long and int
MPI_2INT	int and int
MPI_SHORT_INT	short and int
MPI_LONG_DOUBLE_INT	long double and int

Vom ilustra utilizarea operațiilor globale de reducere **MPI_SUM** în baza următorului exemplu.

Exemplul 3.4.4 *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. Cele două operații colective care apar în rezolvare sunt:*

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

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

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
double f(double a)
{
   return (4.0 / (1.0 + a*a));
}
int main(int argc, char *argv[])
{
int done = 0, n, myid, numprocs, i;
double PI25DT =
       3.14159265358979323846264643;
double mypi, pi, h, sum, x;
double startwtime, endwtime;
int  namelen;
char
       processor_name[MPI_MAX_PROCESS
       OR_
       NAME];
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);
n = 0;
   while (!done)
   {
      if (myid == 0)
      {
        printf("===== Rezultatele programului
        '%s' =====\n",argv[0]);
        printf("Enter the number of intervals:
        (0 quits) ");fflush(stdout);
          scanf("%d",&n);
          MPI_Barrier(MPI_COMM_WORLD);
          startwtime = MPI_Wtime();
      }
      else MPI_Barrier(MPI_COMM_WORLD);
      MPI_Bcast(&n, 1, MPI_INT, 0,
        MPI_COMM_WORLD);
      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;
      fprintf(stderr,"Process %d on %s
      mypi= %.16f\n", myid,
      processor_name, mypi);
      fflush(stderr);
      MPI_Reduce(&mypi, &pi, 1,
      MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
           if (myid == 0)
           {
          printf("Pi is approximately %.16f,
      Error is %.16f\n",
                pi, fabs(pi - PI25DT));
           endwtime = MPI_Wtime();
           printf("wall clock time = %f\n",
      endwtime-startwtime);
           }
    }
  }
  MPI_Finalize();
}
```

Rezultatele posibile ale executării programului:

```
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpiCC -o Exemplu 3.4.4.exe Exemplu_3_4_4.cpp
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpirun -n 16 -machinefile ~/nodes4 Exemplu 3.4.4.exe
===== Rezultatele programului '  Exemplu_3_4_4.exe' =====
Enter the number of intervals: (0 quits) 100000
Process 1 on compute-0-2.local mypi= 0.1963576657186469
Process 2 on compute-0-2.local mypi= 0.1963564157936524
Process 3 on compute-0-2.local mypi= 0.1963551658561532
Process 4 on compute-0-4.local mypi= 0.1963539159061520
Process 10 on compute-0-6.local mypi= 0.1963464159436181
Process 12 on compute-0-8.local mypi= 0.1963439158561127
Process 0 on compute-0-2.local mypi= 0.1963589156311392
Process 6 on compute-0-4.local mypi= 0.1963514159686426
Process 8 on compute-0-6.local mypi= 0.1963489159811299
Process 15 on compute-0-8.local mypi= 0.1963401656311267
Process 7 on compute-0-4.local mypi= 0.1963501659811359
Process 9 on compute-0-6.local mypi= 0.1963476659686233
Process 14 on compute-0-8.local mypi= 0.1963414157186182
Process 5 on compute-0-4.local mypi= 0.1963526659436477
Process 11 on compute-0-6.local mypi= 0.1963451659061137
Process 13 on compute-0-8.local mypi= 0.1963426657936135
Pi is approximately 3.1415926535981260, Error is 0.0000000000083329
wall clock time = 0.000813
===== Rezultatele programului 'HB_Pi.exe' =====
Enter the number of intervals: (0 quits) 100000000
Process 8 on compute-0-6.local mypi= 0.1963495402243708
Process 15 on compute-0-8.local mypi= 0.1963495314743671
Process 4 on compute-0-4.local mypi= 0.1963495452243360
Process 5 on compute-0-4.local mypi= 0.1963495439743813
Process 3 on compute-0-2.local mypi= 0.1963495464743635
Process 11 on compute-0-6.local mypi= 0.1963495364743647
Process 1 on compute-0-2.local mypi= 0.1963495489743604
Process 13 on compute-0-8.local mypi= 0.1963495339743620
Process 14 on compute-0-8.local mypi= 0.1963495327243415
```

```
Process 0 on compute-0-2.local mypi= 0.1963495502243742
Process 7 on compute-0-4.local mypi= 0.1963495414743800
Process 2 on compute-0-2.local mypi= 0.1963495477243492
Process 12 on compute-0-8.local mypi= 0.1963495352243697
Process 9 on compute-0-6.local mypi= 0.1963495389743577
Process 6 on compute-0-4.local mypi= 0.1963495427243758
Process 10 on compute-0-6.local mypi= 0.1963495377243211
Pi is approximately 3.1415926535897749, Error is 0.0000000000000182
wall clock time = 0.172357
===== Rezultatele programului 'HB_Pi.exe' =====
Enter the number of intervals: (0 quits) 0
[Hancu_B_S@hpc]$
```

Vom exemplifica utilizarea operațiilor globale de reducere **MPI_MAX** și **MPI_MIN** în cele ce urmează.

Exemplul 3.4.5 *Să se determine elementele maximale de pe coloanele unei matrice pătrate (dimensiunea este egală cu numărul de procese). Elementele matricei sunt inițializate de procesul cu rankul 0. Să fie utilizate funcțiile* **MPI_Reduce** *și operația* **MPI_MAX**..

Mai jos este prezentat codul programului în limbajul C++ în care se realizează cele menționate în exemplul 3.4.5.

```
#include<mpi.h>
#include<stdio.h>
int main(int argc, char *argv[])
{
int numtask,sendcount,reccount,source;
double *A,*Max_Col;
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==root)
{
printf("\n=====REZULTATUL PROGRAMULUI
         '%s' \n",argv[0]);
A=(double*)malloc(numtask*numtask*sizeo
        f
        (double));
Max_Col=(double*)malloc(numtask*sizeof
        (double));
        for(int i=0;i<numtask*numtask;i++)
        A[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[%d,%d]=%5.2f
        ",i,j,A[i*numtask+j]);
        }
        printf("\n");
        MPI_Barrier(MPI_COMM_WORLD);
}
else MPI_Barrier(MPI_COMM_WORLD);
MPI_Scatter(A,sendcount,MPI_DOUBLE,
        Rows,reccount,MPI_DOUBLE,root,MPI
        _COMM_WORLD);
MPI_Reduce(Rows,Max_Col,numtask,
        MPI_DOUBLE, MPI_MAX, root,
        MPI_COMM_WORLD);
if (myrank==root) {
for(int i=0;i<numtask;i++)
     {
      printf("\n");
      printf("Elementul maximal de pe
       coloana %d=%5.2f ",i,Max_Col[i]);
     }
     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_5.exe Exemplu _3_4_5.cpp
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpirun -n 6 -machinefile ~/nodes4 HB_Scater_MPI_MAX.exe

=====REZULTATUL PROGRAMULUI ' Exemplu _3_4_5.exe'
Tipar datele initiale

A[0,0]= 1.80 A[0,1]= 0.85 A[0,2]= 1.68 A[0,3]= 1.71 A[0,4]= 1.96 A[0,5]= 0.42
A[1,0]= 0.72 A[1,1]= 1.65 A[1,2]= 0.60 A[1,3]= 1.19 A[1,4]= 1.03 A[1,5]= 1.35
A[2,0]= 0.78 A[2,1]= 1.10 A[2,2]= 2.04 A[2,3]= 1.97 A[2,4]= 1.37 A[2,5]= 1.54
A[3,0]= 0.30 A[3,1]= 1.30 A[3,2]= 0.04 A[3,3]= 0.52 A[3,4]= 0.29 A[3,5]= 1.73
A[4,0]= 0.34 A[4,1]= 0.86 A[4,2]= 0.28 A[4,3]= 0.23 A[4,4]= 2.15 A[4,5]= 0.47
A[5,0]= 1.10 A[5,1]= 1.80 A[5,2]= 1.32 A[5,3]= 0.64 A[5,4]= 1.37 A[5,5]= 1.13

Elementul maximal de pe coloana 0= 1.80
Elementul maximal de pe coloana 1= 1.80
Elementul maximal de pe coloana 2= 2.04
Elementul maximal de pe coloana 3= 1.97
Elementul maximal de pe coloana 4= 2.15
Elementul maximal de pe coloana 5= 1.73
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpirun -n 8 -machinefile ~/nodes4 Exemplu _3_4_5.exe

=====REZULTATUL PROGRAMULUI ' Exemplu _3_4_5.exe'
Tipar datele initiale

A[0,0]= 1.80 A[0,1]= 0.85 A[0,2]= 1.68 A[0,3]= 1.71 A[0,4]= 1.96 A[0,5]= 0.42 A[0,6]= 0.72 A[0,7]= 1.65
A[1,0]= 0.60 A[1,1]= 1.19 A[1,2]= 1.03 A[1,3]= 1.35 A[1,4]= 0.78 A[1,5]= 1.10 A[1,6]= 2.04 A[1,7]= 1.97
A[2,0]= 1.37 A[2,1]= 1.54 A[2,2]= 0.30 A[2,3]= 1.30 A[2,4]= 0.04 A[2,5]= 0.52 A[2,6]= 0.29 A[2,7]= 1.73
A[3,0]= 0.34 A[3,1]= 0.86 A[3,2]= 0.28 A[3,3]= 0.23 A[3,4]= 2.15 A[3,5]= 0.47 A[3,6]= 1.10 A[3,7]= 1.80
A[4,0]= 1.32 A[4,1]= 0.64 A[4,2]= 1.37 A[4,3]= 1.13 A[4,4]= 1.06 A[4,5]= 2.09 A[4,6]= 0.63 A[4,7]= 1.66
A[5,0]= 1.13 A[5,1]= 1.65 A[5,2]= 0.86 A[5,3]= 1.91 A[5,4]= 0.61 A[5,5]= 0.76 A[5,6]= 1.73 A[5,7]= 1.97
A[6,0]= 0.15 A[6,1]= 2.04 A[6,2]= 1.13 A[6,3]= 0.18 A[6,4]= 0.41 A[6,5]= 1.42 A[6,6]= 1.91 A[6,7]= 0.75
A[7,0]= 0.14 A[7,1]= 0.04 A[7,2]= 0.98 A[7,3]= 0.14 A[7,4]= 0.51 A[7,5]= 2.08 A[7,6]= 1.94 A[7,7]= 1.83

Elementul maximal de pe coloana 0= 1.80
Elementul maximal de pe coloana 1= 2.04
Elementul maximal de pe coloana 2= 1.68
Elementul maximal de pe coloana 3= 1.91
Elementul maximal de pe coloana 4= 2.15
Elementul maximal de pe coloana 5= 2.09
Elementul maximal de pe coloana 6= 2.04
Elementul maximal de pe coloana 7= 1.97
[Hancu_B_S@hpc Finale]$
```

Exemplul 3.4.6 *Utilizând funcția* **`MPI_Reduce`** *și operațiile* **`MPI_MAX`**, **`MPI_MIN`**, *să se determine elementele maximale de pe liniile și coloanele unei matrice de dimensiuni arbitrare. Elementele matricei sunt inițializate de procesul cu rankul 0.*

Indicații: Fiecare proces MPI va executa o singură dată funcția **`MPI_Reduce`.** Pentru aceasta în programul de mai jos a fost elaborată

funcția **reduceLines** în care se realizează următoarele. Fie că dimensiunea matricei *A* este $n \times m$ și procesul cu rankul *k*, în baza funcției **MPI_Scatterv** a recepționat l_k linii, adică

a_{r1}	...	a_{rm}	a_{r+11}	...	a_{r+1m}	...	a_{r+l_k1}	...	a_{r+l_km}

Atunci procesul *k* construiește următorul vector de lungimea *m*:

$\max\{a_{r1}, a_{r+11}, \dots, a_{r+l_k1}\}$	...	$\max\{a_{rm}, a_{r+1m}, \dots, a_{r+l_km}\}$,

care și este utilizat în funcția **MPI_Reduce.**

Mai jos este prezentat codul programului în limbajul C++ în care se realizează cele menționate în exemplul 3.4.6.

```
#include<mpi.h>
#include<stdio.h>
#include <iostream>
using namespace std;
void calculateSendcountsAndDispls(int rows,
        int cols, int size, int sendcounts[], int
        displs[])
{
  int rowsPerProc = rows / size;
int remainRows = rows % size;
int currDispl = 0;
for (int i = 0; i < size; ++i)
{
        displs[i] = currDispl;
        if (i < remainRows)
        sendcounts[i] = (rowsPerProc +
        1)*cols; else
        sendcounts[i] = rowsPerProc * cols;
        currDispl += sendcounts[i];
        }
}
void invertMatrix(double *m, int mRows, int
        mCols, double *rez)
{
        for (int i = 0; i < mRows; ++i)
          for (int j = 0; j < mCols; ++j)
          rez[j * mRows + i] = m[i * mCols + j];
}
void reduceLines(double* Rows, int
        reccount, int cols, double
        myReducedRow[], bool min)
{
        int myNrOfLines = reccount / cols;
        for (int i = 0; i < cols; ++i)
         {
        double myMaxPerCol_i = Rows[i];
        for (int j = 1; j < myNrOfLines; ++j)
        {
        if (min)
         {
        if (Rows[j * cols + i]<myMaxPerCol_i)
        myMaxPerCol_i = Rows[j * cols + i];
          }
        else
          {
          if (Rows[j * cols + i] >
        myMaxPerCol_i)
        myMaxPerCol_i = Rows[j * cols + i];
          }
        }
        myReducedRow[i] = myMaxPerCol_i;
        }
}
int main(int argc, char *argv[])
{
int size,reccount, source;
double *A;
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==root)
        {
printf("\n=====REZULTATUL PROGRAMULUI
        '%s' \n",argv[0]);
        cout << "Introduceti nr. de rinduri a
        matricei: ";
        cin >> rows;
```

```
        cout << "Introduceti nr. de coloane a
        matricei: ";
        cin >> cols;
        A=(double*)malloc(rows*cols*sizeof
        (double));
       for(int i=0;i<rows*cols;i++)
        A[i]=rand()/1000000000.0;
             printf("Tipar datele initiale\n");
             for(int i=0;i<rows;i++)
             {
                        printf("\n");
                        for(int j=0;j<cols;j++)
        printf("A[%d,%d]=%5.2f ",i,j,A[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);
if (rows >= size)
        {
        calculateSendcountsAndDispls(rows,
        cols, size, sendcounts, displs);
        }
        else
        {
        cout << "Introduceti un numar de linii
        >= nr de procese." << endl;
        MPI_Finalize();
        return 0;
        }
reccount = sendcounts[myrank];
Rows = new double[reccount];
MPI_Scatterv(A, sendcounts, displs,
        MPI_DOUBLE, Rows, reccount,
        MPI_DOUBLE, root,
        MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);
if(myrank==root) cout << "Liniile matricei au
        fost repartizate astfel:" << endl;
cout << "\nProces " << myrank << " - " <<
        reccount / cols << " liniile" << endl;
double myReducedRow[cols];
reduceLines(Rows, reccount, cols,
        myReducedRow, false);
double* maxPerCols;
if (myrank == root)
maxPerCols = new double[cols];
MPI_Reduce(myReducedRow,maxPerCols,
        cols,MPI_DOUBLE,MPI_MAX,root,
        MPI_COMM_WORLD);
if (myrank == root)
{
printf("\nValorile de maxim pe coloanele
        matricii sunt:\n");
for (int i = 0; i < cols; ++i)
printf("Coloana %d - %.2f\n", i,
        maxPerCols[i]);
delete[] maxPerCols;
}
double *invMatr;
if (myrank == root)
{
invMatr = new double[cols * rows];
invertMatrix(A, rows, cols, invMatr);
}
if (cols >= size)
{
calculateSendcountsAndDispls(cols, rows,
        size, sendcounts, displs);
}
else
{
cout << "Introduceti un numar de coloane
        >= nr de procese." << endl;
MPI_Finalize();
return 0;
        }
reccount = sendcounts[myrank];
delete[] Rows;
Rows = new double[reccount];
MPI_Scatterv(invMatr, sendcounts, displs,
        MPI_DOUBLE, Rows, reccount,
        MPI_DOUBLE, root,
        MPI_COMM_WORLD);
double myReducedCol[rows];
reduceLines(Rows, reccount, rows,
        myReducedCol, true);
double* minPerRows;
if (myrank == root)
minPerRows = new double[rows];
MPI_Reduce(myReducedCol,minPerRows,
        rows,MPI_DOUBLE,MPI_MIN,root,
        MPI_COMM_WORLD);
if (myrank == root)
{
```

```
printf("\nValorile de minim pe liniile matricii
        sunt:\n");
for (int i = 0; i < rows; ++i)
printf("Rindul %d - %.2f\n", i,
        minPerRows[i]);
delete[] minPerRows;
free(A);
}
MPI_Finalize();
delete[] Rows;
return 0;
}
```

Rezultatele posibile ale executării programului:

```
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpiCC -o Exemplu _3_4_6.exe Exemplu _3_4_6.cpp
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpirun -n 3 -machinefile ~/nodes4  Exemplu _3_4_6.exe

=====REZULTATUL PROGRAMULUI 'Exemplu _3_4_6.exe'
Introduceti nr. de rinduri a matricei: 7
Introduceti nr. de coloane a matricei: 6
Tipar datele initiale
A[0,0]= 1.80 A[0,1]= 0.85 A[0,2]= 1.68 A[0,3]= 1.71 A[0,4]= 1.96 A[0,5]= 0.42
A[1,0]= 0.72 A[1,1]= 1.65 A[1,2]= 0.60 A[1,3]= 1.19 A[1,4]= 1.03 A[1,5]= 1.35
A[2,0]= 0.78 A[2,1]= 1.10 A[2,2]= 2.04 A[2,3]= 1.97 A[2,4]= 1.37 A[2,5]= 1.54
A[3,0]= 0.30 A[3,1]= 1.30 A[3,2]= 0.04 A[3,3]= 0.52 A[3,4]= 0.29 A[3,5]= 1.73
A[4,0]= 0.34 A[4,1]= 0.86 A[4,2]= 0.28 A[4,3]= 0.23 A[4,4]= 2.15 A[4,5]= 0.47
A[5,0]= 1.10 A[5,1]= 1.80 A[5,2]= 1.32 A[5,3]= 0.64 A[5,4]= 1.37 A[5,5]= 1.13
A[6,0]= 1.06 A[6,1]= 2.09 A[6,2]= 0.63 A[6,3]= 1.66 A[6,4]= 1.13 A[6,5]= 1.65
Liniile matricei au fost repartizate astfel:
Proces 2 - 2 liniile
Proces 0 - 3 liniile
Proces 1 - 2 liniile
Valorile de maxim pe coloanele matricii sunt:
Coloana 0 - 1.80
Coloana 1 - 2.09
Coloana 2 - 2.04
Coloana 3 - 1.97
Coloana 4 - 2.15
Coloana 5 - 1.73
Valorile de minim pe liniile matricii sunt:
Rindul 0 - 0.42
Rindul 1 - 0.60
Rindul 2 - 0.78
Rindul 3 - 0.04
Rindul 4 - 0.23
Rindul 5 - 0.64
Rindul 6 - 0.63
[Hancu_B_S@hpc Finale]$
```

Vom ilustra utilizarea operațiilor globale de reducere **`MPI_MAXLOC`** prin exemplul ce urmează.

Exemplu 3.4.7 *Utilizând funcția* **`MPI_Reduce`** *și operațiile* **`MPI_MAXLOC`** *să se determine elementele maximale de pe coloane și indicele liniei, unei matrice pătrate (dimensiunea este egală cu numărul de procese). Elementele matricei sunt inițializate de procesul cu rankul 0.*

Mai jos este prezentat codul programului în limbajul C++ în care se realizează cele menționate în exemplul 3.4.7.

```
#include<stdio.h>
int main(int argc, char *argv[])
{
 int numtask,sendcount,reccount,source;
double *A;
int i, myrank, root=1;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,
        &myrank);
MPI_Comm_size(MPI_COMM_WORLD,
        &numtask);
double ain[numtask], aout[numtask];
int ind[numtask];
struct {
double val;
int rank;
          } în[numtask], out[numtask];
             sendcount=numtask;
             reccount=numtask;
if(myrank==root)
 {
printf("===== Rezultatele programului '%s'
        =====\n",argv[0]);
A=(double*)malloc(numtask*numtask*sizeo
        f(double));
 for(int i=0;i<numtask*numtask;i++)
 A[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[%d,%d]=%.2f ",i,j,
        A[i*numtask+j]);
        }
        printf("\n");
        MPI_Barrier(MPI_COMM_WORLD);
 }
 else MPI_Barrier(MPI_COMM_WORLD);
MPI_Scatter(A, sendcount,
        MPI_DOUBLE,ain, reccount,
        MPI_DOUBLE, root,
        MPI_COMM_WORLD);
for (i=0; i<numtask; ++i)
        {
        în[i].val = ain[i];
        în[i].rank = myrank;
        }
MPI_Reduce(în,out,numtask,MPI_DOUBLE_
        INT, MPI_MAXLOC, root,
        MPI_COMM_WORLD);
if (myrank == root)
        {printf("\n");
        printf("Valorile maximale de pe
        coloane și indicele liniei:\n");
        for (i=0; i<numtask; ++i) {
        aout[i] = out[i].val;
        ind[i] = out[i].rank;
        printf("Coloana %d, valoarea=  %.2f,
        linia= %d\n",i, aout[i],ind[i]); }
        }
MPI_Finalize();
return 0;
}
```

Rezultatele posibile ale executării programului:

```
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpiCC -o Exemplu_3_4_7.exe Exemplu_3_4_7.cpp
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpirun -n 6 -machinefile ~/nodes4  Exemplu_3_4_7.exe
===== Rezultatele programului 'Exemplu_3_4_7.exe' =====
Tipar datele initiale

A[0,0]=1.80 A[0,1]=0.85 A[0,2]=1.68 A[0,3]=1.71 A[0,4]=1.96 A[0,5]=0.42
A[1,0]=0.72 A[1,1]=1.65 A[1,2]=0.60 A[1,3]=1.19 A[1,4]=1.03 A[1,5]=1.35
A[2,0]=0.78 A[2,1]=1.10 A[2,2]=2.04 A[2,3]=1.97 A[2,4]=1.37 A[2,5]=1.54
A[3,0]=0.30 A[3,1]=1.30 A[3,2]=0.04 A[3,3]=0.52 A[3,4]=0.29 A[3,5]=1.73
A[4,0]=0.34 A[4,1]=0.86 A[4,2]=0.28 A[4,3]=0.23 A[4,4]=2.15 A[4,5]=0.47
A[5,0]=1.10 A[5,1]=1.80 A[5,2]=1.32 A[5,3]=0.64 A[5,4]=1.37 A[5,5]=1.13
Valorile maximale de pe coloane și indicele liniei:
Coloana 0, valoarea=  1.80, linia= 0
Coloana 1, valoarea=  1.80, linia= 5
```

```
Coloana 2, valoarea=  2.04, linia= 2
Coloana 3, valoarea=  1.97, linia= 2
Coloana 4, valoarea=  2.15, linia= 4
Coloana 5, valoarea=  1.73, linia= 3
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpirun -n 8 -machinefile ~/nodes4  Exemplu_3_4_7.exe
===== Rezultatele programului 'Exemplu_3_4_7.exe' =====
Tipar datele initiale
A[0,0]=1.80 A[0,1]=0.85 A[0,2]=1.68 A[0,3]=1.71 A[0,4]=1.96 A[0,5]=0.42 A[0,6]=0.72 A[0,7]=1.65
A[1,0]=0.60 A[1,1]=1.19 A[1,2]=1.03 A[1,3]=1.35 A[1,4]=0.78 A[1,5]=1.10 A[1,6]=2.04 A[1,7]=1.97
A[2,0]=1.37 A[2,1]=1.54 A[2,2]=0.30 A[2,3]=1.30 A[2,4]=0.04 A[2,5]=0.52 A[2,6]=0.29 A[2,7]=1.73
A[3,0]=0.34 A[3,1]=0.86 A[3,2]=0.28 A[3,3]=0.23 A[3,4]=2.15 A[3,5]=0.47 A[3,6]=1.10 A[3,7]=1.80
A[4,0]=1.32 A[4,1]=0.64 A[4,2]=1.37 A[4,3]=1.13 A[4,4]=1.06 A[4,5]=2.09 A[4,6]=0.63 A[4,7]=1.66
A[5,0]=1.13 A[5,1]=1.65 A[5,2]=0.86 A[5,3]=1.91 A[5,4]=0.61 A[5,5]=0.76 A[5,6]=1.73 A[5,7]=1.97
A[6,0]=0.15 A[6,1]=2.04 A[6,2]=1.13 A[6,3]=0.18 A[6,4]=0.41 A[6,5]=1.42 A[6,6]=1.91 A[6,7]=0.75
A[7,0]=0.14 A[7,1]=0.04 A[7,2]=0.98 A[7,3]=0.14 A[7,4]=0.51 A[7,5]=2.08 A[7,6]=1.94 A[7,7]=1.83
Valorile maximale de pe coloane şi indicele liniei:
Coloana 0, valoarea=  1.80, linia= 0
Coloana 1, valoarea=  2.04, linia= 6
Coloana 2, valoarea=  1.68, linia= 0
Coloana 3, valoarea=  1.91, linia= 5
Coloana 4, valoarea=  2.15, linia= 3
Coloana 5, valoarea=  2.09, linia= 4
Coloana 6, valoarea=  2.04, linia= 1
Coloana 7, valoarea=  1.97, linia= 5
[Hancu_B_S@hpc Notate_Exemple]$
```

3.4.4 Exerciţii

1. Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++ în care se modelează funcţia **`MPI_Gather`** cu ajutorul funcţiilor de transmitere a mesajelor de tip proces-proces.
2. Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++ în care se modelează funcţia **`MPI_Allreduce`** cu ajutorul funcţiilor **`MPI_Reduce`** şi **`MPI_Bcast`**.
3. Utilizând funcţia **`MPI_Reduce`** şi operaţiile **`MPI_MAXLOC`**, să se determine elementele maximale de pe coloanele unei matrice de dimensiune arbitrară şi indicele liniei. Elementele matricei sunt iniţializate de procesul cu rankul 0.
4. Folosind funcţia **`MPI_Op_create`**, să se creeze operaţia MPI cu numele **`MPI_ALLMAXLOC`** care va determina toate elementele maximale şi indicele lor. Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++ în care, folosind operaţia **`MPI_ALLMAXLOC`**, să se determine toate elementele maximale şi

indicele liniei de pe coloanele unei matrice, matrice pătrate (dimensiunea este egală cu numărul de procese). Elementele matricei sunt inițializate de procesul cu rankul 0.

5. Fie dată o matrice $A = \|a_{ij}\|_{i=1,n}^{j=1,m}$ care se împarte în blocuri de dimensiunea $n_b \times m_b$. Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++ care să realizeze următoarele:
   a. fiecare proces primeşte un singur[1] bloc al matricei *A*;
   b. fiecare proces atribuie valoarea *0* numai acelor elemente ale submatricei sale care sunt elementele de pe diagonala principală a matricei *A*.

   Matricea *A* este iniţializată numai de procesul cu rankul *0*.

3.5 Rutine MPI de administrare a comunicatorului și a grupelor

3.5.1 Grupe și comunicatori

O grupă este o mulțime ordonată de procese. Fiecare proces dintr-un grup este asociat unui rang întreg unic. Valorile rangului pornesc de la 0 și merg până la *N – 1*, unde *N* este numărul de procese din grup.

În MPI un grup este reprezentat în memoria sistemului ca un obiect. El este accesibil programatorului numai printr-un „handle”. Există două grupe prestabilite: **`MPI_GROUP_EMPTY`** – grupul care nu conține niciun proces și **`MPI_GROUP_NUL`** – se returnează această valoare în cazul când grupul nu poate fi creat. Un grup este totdeauna asociat cu un obiect comunicator. Un comunicator cuprinde un grup de procese care pot comunica între ele. Toate mesajele MPI trebuie să specifice un comunicator. În sensul cel mai simplu, comunicatorul este o „etichetă” suplimentară care trebuie inclusă în apelurile MPI. Ca și grupele, comunicatorii sunt reprezentați în memoria sistemului ca obiecte și sunt accesibili programatorului numai prin „handles”. De exemplu, un handle pentru comunicatorul care cuprinde toate task-urile este **`MPI_COMM_WORLD`**. Rutinele grupului sunt utilizate în principal pentru a specifica procesele care trebuie utilizate pentru a construi un comunicator.

Scopurile principale ale obiectelor grup și comunicator:

[1] Dimensiunea blocului se alege astfel încât să se poată realiza acest lucru.

- Permit organizarea task-urilor, pe baza unor funcții, în grupuri de task-uri.
- Abilitează operațiile de comunicare colectivă într-un subset de task-uri într-un fel în relație.
- Asigură baza pentru implementarea topologiilor virtuale definite de utilizator.
- Garantează siguranța comunicării.

Restricții și alte considerații asupra programării:

- Grupurile/comunicatorii sunt obiecte dinamice – pot fi create și distruse în timpul execuției programului.
- Procesele pot fi în mai mult de un grup/comunicator. Ele vor avea un rang unic în fiecare grup/comunicator.

În MPI comunicarea poate avea loc în cadrul unui grup (intracomunicare) sau între două grupuri distincte (intercomunicare). Corespunzător, comunicatorii se împart în două categorii: **intracomunicatori** și **intercomunicatori**. Un intracomunicator descrie:

- ✓ un grup de procese;
- ✓ contexte pentru comunicare punct la punct și colectivă (aceste două contexte sunt disjuncte, astfel că un mesaj punct la punct nu se poate confunda cu un mesaj colectiv, chiar dacă au același tag);
- ✓ o topologie virtuală (eventual);
- ✓ alte atribute.

MPI are o serie de operații pentru manipularea grupurilor, printre care:.

- ✓ aflarea grupului asociat unui comunicator;
- ✓ găsirea numărului de procese dintr-un grup și a rangului procesului apelant;
- ✓ compararea a două grupuri;
- ✓ reuniunea, intersecția, diferența a două grupuri;
- ✓ crearea unui nou grup din altul existent, prin includerea sau excluderea unor procese;
- ✓ desființarea unui grup.

Funcții similare sunt prevăzute și pentru manipularea intracomunicatorilor:

- ✓ găsirea numărului de procese sau a rangului procesului apelant;
- ✓ compararea a doi comunicatori;
- ✓ duplicarea, crearea unui comunicator;

- ✓ partiționarea grupului unui comunicator în grupuri disjuncte;
- ✓ desființarea unui comunicator.

Un intercomunicator leagă două grupuri, împreună cu contextele de comunicare partajate de cele două grupuri. Contextele sunt folosite doar pentru operații punct la punct, neexistând comunicare colectivă intergrupuri. Nu există topologii virtuale în cazul intercomunicării. O intercomunicare are loc între un proces inițiator, care aparține unui grup local, și un proces țintă, care aparține unui grup distant. Procesul țintă este specificat printr-o pereche (comunicator, rang) relativă la grupul distant. MPI conține peste 40 de rutine relative la grupuri, comunicatori și topologii virtuale. Mai jos vom descrie o parte din aceste funcții (rutine).

3.5.2 Funcțiile MPI de gestionare a grupelor de procese

Funcția MPI_Group_size

Această funcție permite determinarea numărului de procese din grup. Prototipul acestei funcții în limbajul C++ este

```
int MPI_Group_size(MPI_Group group, int *size)
```

unde

IN **group** – numele grupei;
OUT **size** – numărul de procese din grup.

Funcția MPI_Group_rank

Această funcție permite determinarea rankului (un identificator numeric) al proceselor din grup. Prototipul acestei funcții în limbajul C++ este

```
int MPI_Group_rank(MPI_Group group, int *rank)
```

unde

IN **group** – numele grupei;
OUT **rank** – numărul procesului din grup.

Dacă procesul nu face parte din grupul indicat, atunci se returnează valoarea **MPI_UNDEFINED.**

Funcția MPI_Comm_group

Această funcție permite crearea unui grup de procese prin intermediul unui comunicator. Prototipul acestei funcții în limbajul C++ este

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group
    *group)
```

unde

IN **comm** – numele comunicatorului;

OUT **group** – numele grupului.

Astfel, se creează grupul cu numele **group** pentru mulțimea de procese care aparțin comunicatorului cu numele **comm.**

Funcțile MPI_Comm_union, MPI_Comm_intersection, MPI_Comm_difference

Următoarele trei funcții au aceeași sintaxă și se utilizează pentru crearea unui grup nou de procese MPI , ca rezultat al unor operații asupra mulțimilor de procese din două grupe. Prototipul acestor funcții în limbajul C++ este

```
int MPI_Group_union(MPI_Group group1, MPI_Group
    group2, MPI_Group *newgroup)
int MPI_Group_intersection(MPI_Group group1,
    MPI_Group group2, MPI_Group *newgroup)
int MPI_Group_difference(MPI_Group group1,
    MPI_Group group2, MPI_Group *newgroup)
```

unde

IN **group1** – numele primului grup de procese;

IN **group2** – numele grupului doi de procese;

OUT **newgroup** – numele grupului nou creat.

Operațiunile sunt definite după cum urmează:

Union – creează un nou grup din procesele grupei **group1** și din acele procese ale grupei **group2** care nu aparțin grupei **group1** (operația de *reuniune*).

Intersection – creează un nou grup din procesele grupei **group1** care aparțin și grupei **group2** (operația de *intersecție*).

Difference – creează un nou grup din procesele grupei **group1** care nu aparțin grupei **group2** (operația de *diferență*).

Noul grup poate fi și vid, atunci se returnează **MPI_GROUP_EMPTY**.

Funcțile MPI_Comm_incl, MPI_Comm_excl

Următoarele două funcții sunt de aceeași sintaxă, dar sunt complementare. Prototipul acestor funcții în limbajul C++ este

```
int MPI_Group_incl(MPI_Group group, int n, int
    *ranks, MPI_Group *newgroup)
int MPI_Group_excl(MPI_Group group, int n, int
    *ranks, MPI_Group *newgroup)
```

unde

IN **group** – numele grupei existente („părinte”);

IN **n** – numărul de elemente în vectorul **ranks** (care este de fapt egal cu numărul de procese din grupul nou creat);

IN **ranks** – un vector ce conține rankurile proceselor;

OUT **newgroup** – numele grupului nou creat.

Funcția **MPI_Group_incl** creează un nou grup, care constă din procesele din grupul **group**, enumerate în vectorul **ranks**. Procesul cu numărul *i* în noul grup **newgroup** este procesul cu numărul **ranks[i]** din grupul existent **group**.

Funcția **MPI_Group_excl** creează un nou grup din acele procese ale grupului existent **group** care nu sunt enumerate în vectorul **ranks**. Procesele din grupul **newgroup** sunt ordonate la fel ca și în grupul inițial **group**.

Vom exemplifica rutinele MPI descrise mai sus.

Exemplul 3.5.1 *Fie dat un grup „părinte” de procese MPI numerotate 0,...,size-1. Să se elaboreze un program MPI în care se creează un nou grup de k=size/2 procese, alegând aleator procese din grupul „părinte”. Fiecare proces din grupul creat tipărește informația despre sine în forma: rankul din grupul nou (rankul din grupul părinte).*

Mai jos este prezentat codul programului în limbajul C++ în care se realizează cele menționate în exemplul 3.5.1

```
#include<stdio.h>
#include <stdio.h>
#include <mpi.h>
int main(int argc,char *argv[])
{
int i,k,p,size,rank;
int rank_gr;
 char
      processor_name[MPI_MAX_PROCESS
      OR_NAME];
MPI_Status status;
MPI_Group MPI_GROUP_WORLD, newgr;
int* ranks;
int namelen;
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]);
MPI_Barrier(MPI_COMM_WORLD);
srand(time(0));
k=size / 2;
ranks = (int*)malloc(k*sizeof(int));
int rN = 0;
int repeat;
        for (i = 0; i < k; i++)
        {
             do
             {
             repeat = 0;
             rN = rand() % size;
             for (int j = 0; j < i; ++j)
             {
             if (rN == ranks[j])
                          {
                          repeat = 1;
                          break;
                          }
             }
             } while(repeat == 1);

           ranks[i] = rN;
        }
        if(rank==0)
        {
        printf("Au fost extrase aleator %d
        numere dupa cum urmeaza:\n",k);
             for (i = 0; i < k; i++)
             printf(" %d ",ranks[i]);
             printf(" \n");

             MPI_Barrier(MPI_COMM_WORLD)
        ;
        }
        else
        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Comm_group(MPI_COMM_WOR
        LD,&MPI_GROUP_WORLD);
        MPI_Group_incl(MPI_GROUP_WORLD
        ,k,ranks,&newgr);
        MPI_Group_rank(newgr,&rank_gr);
        if (rank_gr != MPI_UNDEFINED)
        printf ("Sunt procesul cu rankul %d
        (%d) de pe nodul %s. \n", rank_gr,
        rank, processor_name);
MPI_Group_free(&newgr);
MPI_Finalize();
return 0;
}
```

Rezultatele posibile ale executării programului:

```
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpiCC -o Exemplu _3_5_1.exe Exemplu _3_5_1.cpp
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpirun -n 16 -machinefile ~/nodes
HB_Grup_Proc_Aleatoare.exe
=====REZULTATUL PROGRAMULUI 'Exemplu _3_5_1.exe'
Au fost extrase aleator 8 numere dupa cum urmeaza:
 8  2  6  1  14  3  10  13
Sunt procesul cu rankul 2 (6) de pe nodul compute-0-4.local.
Sunt procesul cu rankul 7 (13) de pe nodul compute-0-8.local.
Sunt procesul cu rankul 0 (8) de pe nodul compute-0-6.local.
Sunt procesul cu rankul 3 (1) de pe nodul compute-0-2.local.
Sunt procesul cu rankul 4 (14) de pe nodul compute-0-8.local.
Sunt procesul cu rankul 6 (10) de pe nodul compute-0-6.local.
Sunt procesul cu rankul 1 (2) de pe nodul compute-0-2.local.
Sunt procesul cu rankul 5 (3) de pe nodul compute-0-2.local.
[Hancu_B_S@hpc]$
```

3.5.3 Funcțiile MPI de gestionare a comunicatoarelor

În acest paragraf vom analiza funcțiile de lucru cu comunicatorii. Acestea sunt împărțite în funcții de acces la comunicator și funcții utilizate pentru a crea un comunicator. Funcțiile de acces sunt locale și nu necesită comunicații, spre deosebire de funcțiile de creare a comunicatoarelor, care

sunt colective și pot necesita comunicații interprocesor. Două funcții de acces la comunicator **MPI_Comm_size** și **MPI_Comm_rank** au fost analizate deja mai sus.

Funcția MPI_Comm_compare

Este utilizată pentru a compara două comunicatoare. Prototipul funcției în limbajul C++ este

```
int MPI_Comm_compare(MPI_Comm comm1,MPI_Comm
    comm2, int *result)
```

unde

IN **comm1** – numele primului comunicator;
IN **comm2** – numele comunicatorului al doilea;
OUT **result** – rezultatul comparației.

Valorile posibile ale variabilei **result**:

MPI_IDENT Comunicatoarele sunt identice, reprezintă același mediu de comunicare.

MPI_CONGRUENT Comunicatoarele sunt congruente, două medii de comunicare cu aceeași parametri de grup.

MPI_SIMILAR Comunicatoarele sunt similare, grupele conțin aceleași procese, dar cu o altă distribuire a rankurilor.

MPI_UNEQUAL Toate celelalte cazuri.

Crearea unor noi medii de comunicare, comunicatoare, se poate face cu una din următoarele funcții: **MPI_Comm_dup, MPI_Comm_create, MPI_Comm_split.**

Funcția MPI_Comm_dup

Crearea unui comunicator ca și copie a altuia. Prototipul funcției în limbajul C++ este

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm
    *newcomm))
```

unde

IN **comm** – numele comunicatorului părinte;
OUT **newcomm** – numele comunicatorului nou creat.

Funcția MPI_Comm_create

Funcția este utilizată pentru crearea unui comunicator pentru un grup de procese. Prototipul funcției în limbajul C++ este

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group
    group, MPI_Comm *newcomm)
```

unde

IN **comm** – numele comunicatorului părinte;
IN **group** – numele grupei;
IN **newcomm** – numele comunicatorului nou creat.

Pentru procesele care nu fac parte din grupul **group** se returnează valoarea **MPI_COMM_NULL**. Funcția va returna un cod de eroare dacă **group** nu este un subgrup al comunicatorului părinte.

Vom exemplifica rutinele MPI pentru gestionarea comunicatoarelor descrise mai sus.

Exemplul 3.5.2 *Să se elaboreze un program MPI în limbajul C++ în care se creează un nou grup de procese care conține câte un singur proces de pe fiecare nod al clusterului. Pentru acest grup de procese să se creeze un comunicator. Fiecare proces din comunicatorul creat tipărește informația despre sine în forma: rankul din comunicatorul nou (rankul din comunicatorul părinte).*

Elaborarea comunicatoarelor de acest tip poate fi utilizat pentru a exclude transmiterea mesajelor prin rutine MPI între procese care aparțin aceluiași nod. Mai jos este prezentat codul programului în limbajul C++ în care se realizează cele menționate în exemplul 3.5.2.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main(int argc, char *argv[])
{
int i, p, k=0, size, rank, rank_new;
int Node_rank;
int Nodes; //numarul  de noduri
int local_rank =
        atoi(getenv("OMPI_COMM_WORLD_L
        OCAL_RANK"));
char processor_name[MPI_MAX_
        PROCESSOR_NAME];
MPI_Status status;
MPI_Comm com_new, ring1;
MPI_Group MPI_GROUP_WORLD, newgr;
int *ranks,*newGroup;
int namelen;
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("=====REZULTATUL PROGRAMULUI
        '%s' \n", argv[0]);
printf ("Rankurile proceselor din comuni
        catorului 'MPI_COMM_WOLD' au fost
        repartizate astfel: \n"); }
MPI_Barrier(MPI_COMM_WORLD);
 // Se determina numarul de noduri (egal cu
        numarul de procese în grupul creat
```

```
printf ("Rankul %d de pe nodul %s. \n",rank,
        processor_name);
if (local_rank == 0) k = 1;
MPI_Allreduce(&k, &Nodes, 1, MPI_INT,
        MPI_SUM, MPI_COMM_WORLD);
newGroup=(int *)malloc(Nodes*sizeof(int));
ranks = (int *) malloc(size * sizeof(int));
int r;
// Se construieste vectorul newGroup
  if (local_rank == 0)
        ranks[rank] = rank;
  else
        ranks[rank] = -1;
  for (int i = 0; i < size; ++i)
        MPI_Bcast(&ranks[i], 1, MPI_INT, i,
        MPI_COMM_WORLD);
  for (int i = 0, j = 0; i < size; ++i)
  {
        if (ranks[i] != -1)
        {
              newGroup[j] = ranks[i];
              ++j;
        }
  }
MPI_Comm_group(MPI_COMM_WORLD,
        &MPI_GROUP_WORLD);
MPI_Group_incl(MPI_GROUP_WORLD,
        Nodes, newGroup, &newgr);
MPI_Comm_create(MPI_COMM_WORLD,
        newgr, &com_new);
MPI_Group_rank(newgr, &rank_new);
if (rank_new!= MPI_UNDEFINED)
printf ("Procesul cu rankul %d al com.
        'com_new' (%d com.
        'MPI_COMM_WOLD') de pe nodul %s.
        \n", rank_new,rank,processor_name);
MPI_Finalize();
return 0;
}
```

Rezultatele posibile ale executării programului:

```
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpiCC -o Exemplu_3_5_2.exe Exemplu_3_5_2.cpp
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpirun -n 28 -host compute-0-1,compute-0-2,compute-0-
4,compute-0-6,compute-0-8,compute-0-9,compute-0-10 Exemplu_3_5_2.exe
=====REZULTATUL PROGRAMULUI 'Exemplu_3_5_2.exe'
Rankurile proceselor din comunicatorului 'MPI_COMM_WOLD' au fost repartizate astfel:
Rankul 18 de pe nodul compute-0-8.local.
Rankul 19 de pe nodul compute-0-9.local.
Rankul 14 de pe nodul compute-0-1.local.
Rankul 6 de pe nodul compute-0-10.local.
Rankul 22 de pe nodul compute-0-2.local.
Rankul 2 de pe nodul compute-0-4.local.
Rankul 10 de pe nodul compute-0-6.local.
Rankul 11 de pe nodul compute-0-8.local.
Rankul 12 de pe nodul compute-0-9.local.
Rankul 0 de pe nodul compute-0-1.local.
Rankul 20 de pe nodul compute-0-10.local.
Rankul 8 de pe nodul compute-0-2.local.
Rankul 9 de pe nodul compute-0-4.local.
Rankul 3 de pe nodul compute-0-6.local.
Rankul 25 de pe nodul compute-0-8.local.
Rankul 26 de pe nodul compute-0-9.local.
Rankul 7 de pe nodul compute-0-1.local.
Rankul 27 de pe nodul compute-0-10.local.
Rankul 15 de pe nodul compute-0-2.local.
Rankul 16 de pe nodul compute-0-4.local.
Rankul 17 de pe nodul compute-0-6.local.
Rankul 4 de pe nodul compute-0-8.local.
Rankul 5 de pe nodul compute-0-9.local.
Rankul 21 de pe nodul compute-0-1.local.
Rankul 13 de pe nodul compute-0-10.local.
```

```
Rankul 1 de pe nodul compute-0-2.local.
Rankul 23 de pe nodul compute-0-4.local.
Rankul 24 de pe nodul compute-0-6.local.
Pprocesul cu rankul 3 al com. 'com_new'(3 com. 'MPI_COMM_WOLD') de pe nodul compute-0-6.local.
Procesul cu rankul 5 al com. 'com_new'(5 com. 'MPI_COMM_WOLD') de pe nodul compute-0-9.local.
Procesul cu rankul 6 al com. 'com_new'(6 com. 'MPI_COMM_WOLD') de pe nodul compute-0-10.local.
Procesul cu rankul 1 al com. 'com_new'(1 com. 'MPI_COMM_WOLD') de pe nodul compute-0-2.local.
Procesul cu rankul 2 al com. 'com_new'(2 com. 'MPI_COMM_WOLD') de pe nodul compute-0-4.local.
Procesul cu rankul 4 al com. 'com_new'(4 com. 'MPI_COMM_WOLD') de pe nodul compute-0-8.local.
Procesul cu rankul 0 al com. 'com_new'(0 com. 'MPI_COMM_WOLD') de pe nodul compute-0-1.local.
[Hancu_B_S@hpc]$
```

3.5.4 Topologii virtuale. Rutine de creare a topologiei carteziene

În termenii MPI, o topologie virtuală descrie o aplicație/ordonare a proceselor MPI într-o „formă” geometrică. Cele două tipuri principale de topologii suportate de MPI sunt cea *carteziană (grilă)* și cea sub formă *de graf*. Topologiile MPI sunt virtuale – poate să nu existe nicio relație între structura fizică a unei mașini paralele și topologia proceselor. Topologiile virtuale sunt construite pe grupuri și comunicatori MPI și trebuie să fie programate de cel care dezvoltă aplicația. Topologiile virtuale pot fi utile pentru aplicații cu forme de comunicare specifice – forme (patterns) care se potrivesc unei structuri topologice MPI. De exemplu, o topologie carteziană se poate dovedi convenabilă pentru o aplicație care reclamă comunicare cu *4* din vecinii cei mai apropiați pentru date bazate pe grile.

Funcția MPI_Cart_create

Pentru a crea un comunicator (un mediu virtual de comunicare) cu topologie carteziană este folosită rutina **`MPI_Cart_create`**. Cu această funcție se poate crea o topologie cu număr arbitrar de dimensiuni, și pentru fiecare dimensiune în mod izolat pot fi aplicate condiții – limită periodice. Astfel, în funcție de care condiții la limită se impun, pentru o topologie unidimensională obținem sau structură liniară, sau un inel (cerc). Pentru topologie bidimensională, respectiv, fie un dreptunghi sau un cilindru sau tor. Prototipul funcției în limbajul C++ este

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

unde

IN **comm_old** – numele comunicatorului părinte;
IN **ndims** – dimensiunea (numărul de axe);

IN **dims** – un vector de dimensiunea **ndims** în care se indică numărul de procese pe fiecare axă;

IN **periods** – un vector logic de dimensiunea **ndims** în care se indică condițiile la limită pentru fiecare axă (**true-** condițiile sunt periodice, adică axa este „închisă”, **false-** condițiile sunt neperiodice, adică axa nu este „închisă”);

IN **reorder** – o variabilă logică, care indică dacă se va face renumerotarea proceselor (**true**) sau nu (**false**);

OUT **comm_cart** – numele comunicatorului nou creat.

Funcția este colectivă, adică trebuie să fie executată pe toate procesele din grupul de procese ale comunicatorului **comm_old.** Parametrul **reorder=false** indică faptul că ID-urile proceselor din noul grup vor fi aceleași ca și în grupul vechi. Dacă **reorder=true**, MPI va încerca să le schimbe cu scopul de a optimiza comunicarea.

Următoarele funcții descrise în acest paragraf au un caracter auxiliar sau informațional.

Funcția MPI_Dims_create

Această funcție se utilizează pentru determinarea unei configurații optimale ale rețelei de procese. Prototipul funcției în limbajul C++ este

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

unde

IN **nnodes** – numărul total de noduri în rețea;

IN **ndims** – dimensiunea (numărul de axe);

IN/OUT **dims** – un vector de dimensiunea **ndims** în care se indică numărul recomandat de procese pe fiecare ax.

La intrare în procedură în vectorul **dims** trebuie să fie înregistrate numere întregi non-negative. În cazul în care elementul **dims[i]** este un număr pozitiv, atunci pentru această axă (direcție) nu este realizat niciun calcul (numărul de procese de-a lungul acestei direcții este considerat a fi

specificat). Se determină (calculează) numai acele **dims[i]**, care înainte de aplicarea procedurii sunt setate la *0*. Funcția are ca scop crearea unei distribuții cât mai uniforme a proceselor de-a lungul axei, aranjându-le în ordine descrescătoare. De exemplu, pentru 12 procese, aceasta va construi o grilă tridimensională *4×3×1*. Rezultatul acestei rutine poate fi folosită ca un parametru de intrare pentru funcția **MPI_Cart_create**.

Funcția MPI_Cart_get

Această funcție se utilizează pentru a obține o informație detaliată despre comunicatorul cu topologie carteziană. Prototipul funcției în limbajul C++ este

```
int MPI_Cart_get(MPI_Comm comm, int ndims, int
    *dims, int *periods, int *coords)
```

unde

IN **comm** – comunicatorul cu topologie carteziană;

IN **ndims** – dimensiunea (numărul de axe);

OUT **dims** – un vector de dimensiunea **ndims** în care se returnează numărul de procese pe fiecare axă;

OUT **periods** – un vector logic de dimensiunea **ndims** în care se returnează condițiile la limită pentru fiecare axă (**true-** condițiile sunt periodice, **false-** condițiile sunt neperiodice);

OUT **coords** – coordonatele carteziene ale proceselor care apelează această funcție.

Următoarele două funcții realizează corespondența dintre rankul (identificatorul) procesului în comunicatorul pentru care s-a creat topologia carteziană și coordonatele sale într-o grilă carteziană.

Funcția MPI_Cart_rank

Această funcție se utilizează pentru a obține rankul proceselor în baza coordonatelor sale. Prototipul funcției în limbajul C++ este

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

unde

IN **comm** – comunicatorul cu topologie carteziană;

IN **coords** – coordonatele în sistemul cartezian);

OUT **rank** – rankul (identificatorul) procesului.

Funcția MPI_Cart_coords

Această funcție se utilizează pentru a obține coordonatele carteziene ale proceselor în baza rankurilor (identificatoarelor) sale. Prototipul funcției în limbajul C++ este

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

unde

IN **comm**	– comunicatorul cu topologie carteziană;
IN **rank**	– rankul (identificatorul) procesului;
IN **ndims**	– numărul de axe (direcții);
OUT **coords**	– coordonatele în sistemul cartezian.

Funcția MPI_Cart_shift

În mulți algoritmi numerici se utilizează operația de deplasare a datelor de-a lungul unor axe ale grilei carteziene. În MPI există funcția **MPI_Cart_shift** care realizează această operație. Mai precis, deplasarea datelor este realizată folosind **MPI_Sendrecv**, iar funcția **MPI_Cart_shift** calculează pentru fiecare proces parametrii funcției **MPI_Sendrecv funcția** (sursa și destinația). Prototipul funcției în limbajul C++ este

```
int MPI_Cart_shift(MPI_Comm comm, int direction,
    int disp, int *rank_source, int *rank_dest)
```

unde

IN **comm**	– comunicatorul cu topologie carteziană;
IN **direction**	– numărul axei (direcției) de-a lungul căreia se realizează deplasarea;
IN **disp**	– valoarea pasului de deplasare (poate fi pozitivă – deplasare în direcția acelor cronometrului, sau negative – deplasare în direcția inversă acelor cronometrului);
OUT **rank_source**	– rankul procesului de la care se vor recepționa datele;
OUT **rank_dest**	– rankul procesului care va recepționa datele.

Vom exemplifica rutinele MPI pentru gestionarea comunicatoarelor cu topologie cartezianã.

Exemplul 3.5.3 *Sã se elaboreze un program MPI în limbajul C++ în care se creeazã un nou grup de procese care conține câte un singur proces de pe fiecare nod al clusterului. Pentru acest grup de procese sã se creeze un comunicator cu topologie cartezianã de tip cerc (inel) și sã se realizeze transmiterea mesajelor pe cerc.*

Mai jos este prezentat codul programului în limbajul C++ în care se realizează cele indicate în exemplul 3.5.3.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main(int argc, char *argv[])
{
int i, p, k=0, size, size_new, rank, rank_new,
        sour, dest;
int Node_rank, rank_gr;
int Nodes;
int local_rank =
        atoi(getenv("OMPI_COMM_WORLD_
        LOCAL_RANK"));
char
        processor_name[MPI_MAX_PROCESS
        OR_NAME];
MPI_Status status;
MPI_Comm com_new, ring1;
MPI_Group MPI_GROUP_WORLD, newgr;
int dims[1], period[1], reord;
int *ranks,*newGroup;
int namelen;
int D1 = 123, D2;
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("=====REZULTATUL PROGRAMULUI
        '%s' \n", argv[0]);
   MPI_Barrier(MPI_COMM_WORLD);
   if (local_rank == 0) k = 1;
   MPI_Allreduce(&k, &Nodes, 1, MPI_INT,
        MPI_SUM, MPI_COMM_WORLD);
   newGroup=(int *) malloc(Nodes *
        sizeof(int));
   ranks = (int *) malloc(size * sizeof(int));
    int r;
   if (local_rank == 0)
        ranks[rank] = rank;
   else
        ranks[rank] = -1;

   for (int i = 0; i < size; ++i)
        MPI_Bcast(&ranks[i], 1, MPI_INT, i,
        MPI_COMM_WORLD);
    for (int i = 0, j = 0; i < size; ++i)
   {
        if (ranks[i] != -1)
        {
             newGroup[j] = ranks[i];
             ++j;
        }
   }
MPI_Comm_group(MPI_COMM_WORLD,
        &MPI_GROUP_WORLD);
MPI_Group_incl(MPI_GROUP_WORLD,
        Nodes, newGroup, &newgr);
MPI_Comm_create(MPI_COMM_WORLD,
        newgr, &com_new);
MPI_Group_rank(newgr, &rank_gr);
 if (rank_gr != MPI_UNDEFINED) {
      MPI_Comm_size(com_new,
        &size_new);
      MPI_Comm_rank(com_new,
        &rank_new);
      dims[0] = size_new;
      period[0] = 1;
      reord = 1;
      MPI_Cart_create(com_new, 1, dims,
        period, reord, &ring1);
      MPI_Cart_shift(ring1, 1, 2, &sour,
        &dest);
      D1 = D1 + rank;
```

```
      MPI_Sendrecv(&D1, 1, MPI_INT, dest,
        12, &D2, 1, MPI_INT, sour, 12, ring1,
        &status);
      if (rank_new == 0) {
     printf("===Rezultatul
        MPI_Sendrecv:\n");
         MPI_Barrier(com_new);
      }
      else MPI_Barrier(com_new);
printf ("Proc. %d (%d from %s), recv. from
       proc. %d the value %d and send to
       proc. %d the value %d\n", rank_new,
       rank,processor_name, sour,
       D2,dest,D1);
     MPI_Barrier(com_new);
     MPI_Group_free(&newgr);
     MPI_Comm_free(&ring1);
     MPI_Comm_free(&com_new);
   }
MPI_Finalize();
return 0;
}
```

Rezultatele posibile ale executării programului:

```
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpiCC -o Exemplu_3_5_3.exe Exemplu_3_5_3.cpp
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpirun -n 44 -machinefile ~/nodes o Exemplu_3_5_3.exe
=====REZULTATUL PROGRAMULUI o Exemplu_3_5_3.exe
===Rezultatul MPI_Sendrecv:
Proc.0 (0 from compute-0-0.local), recv. from proc. 9 the value 159 and send to proc. 2 the value 123
Proc.1 (4 from compute-0-1.local), recv. from proc. 10 the value 163 and send to proc. 3 the value 127
Proc.5 (20 from compute-0-6.local), recv. from proc. 3 the value 135 and send to proc. 7 the value 143
Proc.2 (8 from compute-0-2.local), recv. from proc. 0 the value 123 and send to proc. 4 the value 131
Proc.3 (12 from compute-0-4.local), recv. from proc. 1 the value 127 and send to proc. 5 the value 135
Proc.4 (16 from compute-0-5.local), recv. from proc. 2 the value 131 and send to proc. 6 the value 139
Proc.7 (28 from compute-0-8.local), recv. from proc. 5 the value 143 and send to proc. 9 the value 151
Proc.6 (24 from compute-0-7.local), recv. from proc. 4 the value 139 and send to proc. 8 the value 147
Proc.8 (32 from compute-0-9.local), recv. from proc. 6 the value 147 and send to proc. 10 the value
155
Proc.9 (36 from compute-0-10.local), recv. from proc. 7 the value 151 and send to proc. 0 the value
159
Proc.10 (40 from compute-0-12.local), recv. from proc. 8 the value 155 and send to proc. 1 the value
163
[Hancu_B_S@hpc]$
```

Exemplul 3.5.4 *Să se elaboreze un program MPI în limbajul C++ în care se creează un comunicator cu topologie carteziană de tip cub și să se determine pentru un anumit proces vecinii săi pe fiecare axă de coordonate.*

Mai jos este prezentat codul programului în limbajul C++ în care se realizează cele indicate în exemplul 3.5.4.

```
#include <mpi.h>
#include <stdio.h>
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
int rank,test_rank=2;
int size;
int ndims = 3;
int source, dest;
int up_y,down_y,right_x,left_x,up_z,
      down_z;
int dims[3]={0,0,0},coords[3]={0,0,0};
int periods[3]={0,0,0},
reorder = 0;
MPI_Comm comm;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD,
      &size);
MPI_Comm_rank(MPI_COMM_WORLD,
      &rank);
```

```
MPI_Dims_create(size, ndims, dims);
if(rank == 0)
        {
        printf("\n=====REZULTATUL
        PROGRAMULUI '%s' \n",argv[0]);
        for ( int i = 0; i < 3; i++ )
        cout << "Numarul de procese pe axa
        "<< i<< " este "<< dims[i] << "; ";
        cout << endl;
   }
MPI_Barrier(MPI_COMM_WORLD);
MPI_Cart_create(MPI_COMM_WORLD,
        ndims, dims, periods,
        reorder,&comm);
MPI_Cart_coords(comm, rank, ndims,
        coords);
cout << "Procesul cu rankul " << rank << "
        are coordonatele (" << coords[0] << ","
        << coords[1] << "," << coords[2]
        <<")"<< endl;
MPI_Barrier(MPI_COMM_WORLD);
if(rank == test_rank)
        {
        MPI_Cart_shift(comm,0,1,&left_x,
        &right_x);
        MPI_Cart_shift(comm,1,1,&up_y,
        &down_y);
       MPI_Cart_shift(comm,2,1,&up_z,
        &down_z);
  printf("Sunt procesul cu rankul %d, vecinii
        mei sunt: \n",rank);
  printf("   pe directia axei X : stanga %d
        dreapta %d \n",left_x,right_x);
  printf("   pe directia axei Y : stanga %d
        dreapta %d \n",up_y,down_y);
  printf("   pe directia axei Z : stanga %d
        dreapta %d \n",up_z,down_z);
  printf("Valorile negative semnifica lipsa
        procesului vecin!\n");
      }
MPI_Finalize();
return 0;
}
```

Rezultatele posibile ale executării programului:

```
 [Hancu_B_S@hpc]$ /opt/openmpi/bin/mpiCC -o Exemplu_3_5_4.exe Exemplu_3_5_4.cpp
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpirun -n 16 -machinefile ~/nodes4  Exemplu_3_5_4.exe
=====REZULTATUL PROGRAMULUI ' Exemplu_3_5_4.exe'
Numarul de procese pe axa 0 este 4; Numarul de procese pe axa 1 este 2; Numarul de procese pe axa
2 este 2;
Procesul cu rankul 14 are coordonatele (3,1,0)
Procesul cu rankul 5 are coordonatele (1,0,1)
Procesul cu rankul 0 are coordonatele (0,0,0)
Procesul cu rankul 10 are coordonatele (2,1,0)
Procesul cu rankul 12 are coordonatele (3,0,0)
Procesul cu rankul 4 are coordonatele (1,0,0)
Procesul cu rankul 2 are coordonatele (0,1,0)
Procesul cu rankul 9 are coordonatele (2,0,1)
Procesul cu rankul 13 are coordonatele (3,0,1)
Procesul cu rankul 6 are coordonatele (1,1,0)
Procesul cu rankul 1 are coordonatele (0,0,1)
Procesul cu rankul 11 are coordonatele (2,1,1)
Procesul cu rankul 3 are coordonatele (0,1,1)
Procesul cu rankul 8 are coordonatele (2,0,0)
Procesul cu rankul 15 are coordonatele (3,1,1)
Procesul cu rankul 7 are coordonatele (1,1,1)
Sunt procesul cu rankul 2, vecinii mei sunt:
  pe directia axei X : stanga -2 dreapta 6
  pe directia axei Y : stanga 0 dreapta -2
  pe directia axei Z : stanga -2 dreapta 3
Valorile negative semnifica lipsa procesului vecin!
[Hancu_B_S@hpc]$
```

3.5.5 Exerciţii

1. Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++ în care se creează un grup de procese, al căror rank *k* se împarte fără rest la 3. Procesele din grupul nou creat tipăresc rankul lor şi numele nodului.
2. Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++ în care se creează un comunicator cu topologie de tip cerc pentru un grup de procese extrase aleator din grupul părinte. Procesele din comunicatorul nou creat transmit unul altuia rankul lor.
3. Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++ în care se creează două grupe de procese, încât întrun grup se realizează transmiterea datelor pe cerc şi în altul – în baza schemei master-slave.
4. Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++ în care se creează un comunicator cu topologie de tip cub. Să se realizeze comunicarea pe cerc a proceselor care aparţin unei şi aceleiaşi faţete ale cubului.
5. 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 $m_k \times m_p$. Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++ în care se creează un comunicator cu topologie carteziană de dimensiunea *2* şi procesul cu rankul *0*, care iniţializează matricea, trimite procesului cu coordonatele (k, p) submatricea A_{kp}. Fiecare proces din topologia nou creată tipăreşte submatricea primită.

3.6 Accesul distant la memorie. Funcțiile RMA

3.6.1 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.

3.6.2 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*;
IN	**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;
IN	**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 **assert** – un număr întreg;
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 3.6.1 *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 =
        3.14159265358979238462643;
int n, numprocs, myid, i,done = 0;
double mypi, pi, h, sum, x;
int  namelen;
char
        processor_name[MPI_MAX_PROCESS
        OR_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
[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):
0
[Hancu_B_S@hpc]$
```

3.6.3 Exerciţii

1. Să se realizeze o analiză comparativă a timpului de execuţie pentru programele MPI descrise în exemplul 3.6.1 şi exemplul 3.4.4.
2. De ce orice apel al funcțiilor RMA trebuie „bordat” cu funcția **`MPI_Win_fence`**?
3. Ce funcţii RMA se pot utiliza pentru înlocuirea funcţiei **`MPI_Sendrecv`**?
4. Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++ în care se realizează condițiile enunțate în exemplul 3.3.1, utilizând comunicări unic-direcţionate, adică funcţiile RMA.
5. Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++ în care se realizează condițiile enunțate în exemplul 3.4.,5 utilizând comunicări unic-direcţionate, adică funcţiile RMA.

3.7 Generarea dinamică a proceselor MPI

3.7.1 Modalităţi de generare dinamică a proceselor

În toate exemplele descrise mai sus s-a folosit un singur mod de generare a proceselor MPI. Și anume prin utilizarea comenzii **mpirun.** Acest mod de generare a proceselor se numește static.

MPI presupune existența unui mediu de execuție a proceselor, cu care interacțiunea este păstrată la un nivel scăzut pentru a evita compromiterea portabilității. Interacțiunea se limitează la următoarele aspecte:

- ✓ Un proces poate porni dinamic alte procese prin funcția **`MPI_Comm_spawn`** și **`MPI_Comm_spawn_multiple.`**
- ✓ Poate comunica printr-un argument **`info`** informații despre unde și cum sa pornească procesul.

✓Un atribut **MPI_UNIVERSE_SIZE** al **MPI_COMM_WORLD** precizează câte procese pot rula în total, deci câte pot fi pornite dinamic, în plus faţă de cele în execuție.

Funcția MPI_Comm_spawn

Această funcție se utilizează pentru generarea unui număr de procese MPI, fiecare dintre acestea va executa același cod de program. Prototipul funcției în limbajul C++ este

```
int MPI_Comm_spawn(char *command, char *argv[],
    int maxprocs, MPI_Info info, int root,
    MPI_Comm comm, MPI_Comm *intercomm,int
    array_of_errcodes[])
```

unde

IN **command** – specifică numele codului de program care va fi executat de procesele MPI generate;

IN **argv[]** – conține argumentele transmise programului în forma unui tablou de șiruri de caractere;

IN **maxprocs** – numărul de procese generate care vor executa programul MPI specificat de **command**;

IN **info** – conține informații adiționale pentru mediul de execuție în forma unor perechi de șiruri de caractere (cheie, valoare);

IN **root** – rankul procesului pentru care sunt descrise argumentele anterioare (**info** etc.);

IN **comm** – intracomunicatorul pentru grupul de procese care conține procesul generator (de procese MPI);

OUT **intercomm** – un intercomunicator pentru comunicare între părinți și fii;

OUT **array_of_errcodes[]** – conține codurile de erori.

Astfel funcția **MPI_Comm_spawn** întoarce un intercomunicator pentru comunicare între procese „părinţi” și procese „fii”. Acesta conține procesele „părinte” în grupul local și procesele „fii” în grupul distant.

Funcția MPI_Comm_ get_parent

Intercomunicatorul poate fi obținut de procesele „fii” apelând această funcție. Prototipul funcției în limbajul C++ este

```
int MPI_Comm_get_parent(MPI_Comm *parent)
```

unde

OUT **parent** – specifică numele intercomunicatorului „părinte” ;

În cazul în care un proces a fost generat utilizând funcția **MPI_Comm_spawn** sau **MPI_Comm_spawn_multiple**, atunci funcția **MPI_Comm_get_parent** returnează intercomunicatorul părinte pentru procesul curent. În cazul în care procesul nu a fost generat, **MPI_Comm_get_parent** returnează valoarea **MPI_COMM_NULL**.

Vom ilustra utilizarea rutinelor MPI descrise mai sus prin următorul exemplu.

Exemplul 3.7.1 *Să se elaboreze un program MPI în limbajul C++ pentru generarea dinamică a proceselor care, la rândul lor, vor executa același cod de program.*

Mai jos este prezentat codul programului în limbajul C++ în care se realizează cele indicate în exemplul 3.7.1. Codul programului **Exemplu_3_7_1.cpp** în care se generează procesele.

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
int world_size, universe_size,
      *universe_sizep, flag, err[4], namelen,
      rank;
MPI_Comm everyone;
char
    worker_program[100]="./HB_MPI_Spaw
    n_Worker.exe";
char
      processor_name[MPI_MAX_PROCESS
      OR_NAME];
MPI_Init(&argc, &argv);
MPI_Get_processor_name(processor_name
      ,&namelen);
MPI_Comm_size(MPI_COMM_WORLD,
      &world_size);
MPI_Comm_rank(MPI_COMM_WORLD,
      &rank);
if (world_size != 1)  printf("Top heavy with
      management");
MPI_Attr_get(MPI_COMM_WORLD,
      MPI_UNIVERSE_SIZE,&universe_sizep,
      &flag);
if (!flag) {
    printf("This MPI does not support
      UNIVERSE_SIZE. How many n
      processes total?\n");
    scanf("%d", &universe_size);
```

```
        }
else universe_size = *universe_sizep;
if (universe_size == 1)  printf("No room to
        start workers");
universe_size=9;
MPI_Comm_spawn(worker_program,
        MPI_ARGV_NULL, universe_size-1,
        MPI_INFO_NULL, 0,
        MPI_COMM_SELF, &everyone, err);
printf("===I am Manager ('%s'), run on the
        node '%s' with rank %d and generate
        %d proceses that run the program '%s'
        ===\n",argv[0],processor_name,rank,
        universe_size-1,worker_program);
MPI_Finalize();
return 0;
}
```

Codul programului **`HB_MPI_Spawn_Worker.cpp`** care va fi executat de procesele generate:

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
int size,size1,rank,namelen,t,incep=3;
char
        processor_name[MPI_MAX_PROCESS
        OR_NAME];
MPI_Comm parent;
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_get_parent(&parent);
MPI_Get_processor_name(processor_name
        ,&namelen);
if (parent == MPI_COMM_NULL)
printf("=== Intercomunicatorul parinte nu a
        fost creat!\n");
MPI_Comm_remote_size(parent, &size);
MPI_Comm_size(MPI_COMM_WORLD,
        &size1);
MPI_Comm_rank(MPI_COMM_WORLD,
        &rank);
if (size != 1) printf("Something's wrong with
        the parent");
printf("Module '%s'. Start on the processor
        rank %d of the node name '%s'of
        world_size  %d \n",argv[0], rank,
        processor_name, size);
MPI_Barrier(MPI_COMM_WORLD);
if(rank==incep)
{
MPI_Send(&rank,1,MPI_INT, (rank + 1) %
        size1, 10, MPI_COMM_WORLD);
MPI_Recv(&t,1,MPI_INT, (rank+size1-1) %
        size1,10,MPI_COMM_WORLD,&status
        );
}
Else
{
MPI_Recv(&t,1,MPI_INT, (rank+size1-
        1)%size1, 10, MPI_COMM_WORLD,
        &status);
MPI_Send(&rank,1,MPI_INT,(rank+1)%size1,
        10,MPI_COMM_WORLD);
}
printf("proc num %d@%s rcvd=%d from
        %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_7_1.exe  Exemplu _3_7_1.cpp
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpiCC -o HB_MPI_Spawn_Worker.exe
HB_MPI_Spawn_Worker.cpp
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpirun -n 1 -machinefile ~/nodes Exemplu _3_7_1.exe
===I am Manager (' Exemplu _3_7_1.exe'), run on the node 'compute-0-0.local' with rank 0 and
generate 8 proceses that run the program './HB_MPI_Spawn_Worker.exe' ===
Module './HB_MPI_Spawn_Worker.exe'. Start on the processor rank 3 of the node name 'compute-0-
1.local'of world_size  1
Module './HB_MPI_Spawn_Worker.exe'. Start on the processor rank 7 of the node name 'compute-0-
2.local'of world_size  1
```

```
Module './HB_MPI_Spawn_Worker.exe'. Start on the processor rank 4 of the node name 'compute-0-
1.local'of world_size  1
Module './HB_MPI_Spawn_Worker.exe'. Start on the processor rank 6 of the node name 'compute-0-
1.local'of world_size  1
Module './HB_MPI_Spawn_Worker.exe'. Start on the processor rank 0 of the node name 'compute-0-
0.local'of world_size  1
Module './HB_MPI_Spawn_Worker.exe'. Start on the processor rank 1 of the node name 'compute-0-
0.local'of world_size  1
Module './HB_MPI_Spawn_Worker.exe'. Start on the processor rank 2 of the node name 'compute-0-
0.local'of world_size  1
Module './HB_MPI_Spawn_Worker.exe'. Start on the processor rank 5 of the node name 'compute-0-
1.local'of world_size  1
proc num 4@compute-0-1.local rcvd=3 from 3
proc num 7@compute-0-2.local rcvd=6 from 6
proc num 0@compute-0-0.local rcvd=7 from 7
proc num 6@compute-0-1.local rcvd=5 from 5
proc num 1@compute-0-0.local rcvd=0 from 0
proc num 5@compute-0-1.local rcvd=4 from 4
proc num 2@compute-0-0.local rcvd=1 from 1
proc num 3@compute-0-1.local rcvd=2 from 2
[Hancu_B_S@hpc]$
```

Funcția MPI_Comm_spawn_multiple

Această funcție se utilizează pentru generarea unui număr de procese MPI, fiecare dintre acestea pot executa coduri diferite de program. Prototipul funcției în limbajul C++ este

```
int MPI_Comm_spawn_multiple(int count, char
    *array_of_commands[], char
    **array_of_argv[], int array_of_maxprocs[],
    MPI_Info array_of_info[], int root, MPI_Comm
    comm, MPI_Comm *intercomm, int
    array_of_errcodes[])
```

unde

IN **count** – numărul de programe care urmează a fi executate (este relevant numai pentru procesul **root**);

IN **array_of_commands[]** – vector de lungimea **count** în care elementul **i** specifică numele codului de program care vor fi executate de procesele MPI generate (este relevant numai pentru procesul **root**);

IN **array_of_argv[]** – vector de lungimea **count** în care

	se specifică argumentele transmise programelor, în forma unui tablou de șiruri de caractere (este relevant numai pentru procesul **root**);
IN **array_of_maxprocs[]**	– vector de lungimea **count** în care elementul **i** specifică numărul maximal de procese care vor executa programul indicat de vectorul **array_of_commands[]** (este relevant numai pentru procesul **root**);
IN **array_of_info[]**	– vector de lungimea **count** în care elementul **i** conține informații adiționale pentru mediul de execuție în forma unor perechi de șiruri de caractere (cheie, valoare) (este relevant numai pentru procesul **root**) ;
IN **root**	– rankul procesului pentru care sunt descrise argumentele anterioare;
IN **comm**	– intracomunicatorul pentru grupul de procese care conține procesul generator (de procese MPI) ;
OUT **intercomm**	– un intercomunicator pentru comunicare între părinţi şi fii.
OUT **array_of_errcodes[]**	– conține codurile de erori

Vom ilustra utilizarea rutinei **MPI_Comm_spawn_multiple** prin următorul exemplu.

Exemplul 3.7.2 *Să se elaboreze un program MPI în limbajul C++ pentru generarea dinamică a proceselor, care la rândul lor, vor executa coduri diferite de program.*

Mai jos este prezentat codul programului în limbajul C++ în care se realizează cele indicate în exemplul 3.7.2.

Codul programului *exemplul_3_7_2.cpp* în care se generează procesele:

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
int world_size, *universe_sizep, flag,
        err[4],namelen,rank;
char
        processor_name[MPI_MAX_PROCESS
        OR_NAME];
MPI_Comm everyone
const int count = 3;
int universe_size[count] = {5, 4, 3};
char *worker_program[count] =
        {"./HB_MPI_Spawn_Worker1.exe",
        "./HB_MPI_Spawn_Worker2.exe",
        "./HB_MPI_Spawn_Worker3.exe"};
char **args[count];
char *argv0[] = {NULL};
char *argv1[] = {NULL};
char *argv2[] = {NULL};
args[0] = argv0;
args[1] = argv1;
args[2] = argv2;
MPI_Info hostinfo[count];
for(int i = 0; i < count; i++)
        {
        hostinfo[i] = MPI_INFO_NULL;
        }
MPI_Init(&argc, &argv);
MPI_Get_processor_name(processor_name
        ,&namelen);
MPI_Comm_size(MPI_COMM_WORLD,
        &world_size);
MPI_Comm_rank(MPI_COMM_WORLD,
        &rank);
if (world_size != 1)  printf("Top heavy with
        management");
printf("===I am Manager ('%s'), run on the
        node '%s' with rank %d and generate
        the following proceses:
        \n",argv[0],processor_name,rank);
for(int i = 0; i < count; i++)
{
printf("%d proceses run the module '%s'\n",
        universe_size[i],worker_program[i]);
}
printf("===\n");

MPI_Comm_spawn_multiple(count,
        worker_program,  args, universe_size,
        hostinfo, 0 ,MPI_COMM_SELF,
        &everyone, err);
MPI_Finalize();
return 0;
}
}
```

Codul programului *HB_MPI_Spawn_Worker1.cpp* care va fi executat de procesele generate. Programele *HB_MPI_Spawn_Worker2.cpp*, *HB_MPI_Spawn_Worker3.cpp* sunt similare.

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
int size,size1,rank,namelen,t,incep=3;
char
        processor_name[MPI_MAX_PROCESS
        OR_NAME];
MPI_Comm parent;
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_get_parent(&parent);
MPI_Get_processor_name(processor_name
        ,&namelen);
if (parent == MPI_COMM_NULL)
printf("=== Intercomunicatorul parinte nu a
        fost creat!\n");
MPI_Comm_remote_size(parent, &size);
MPI_Comm_size(MPI_COMM_WORLD,
        &size1);
MPI_Comm_rank(MPI_COMM_WORLD,
        &rank);
if (size != 1) printf("Something's wrong with
        the parent");
printf("Module '%s'. Start on the processor
        rank %d of the node name '%s'of
        world_size  %d \n",argv[0], rank,
        processor_name, size);
MPI_Barrier(MPI_COMM_WORLD);
if(rank==incep)
{
MPI_Send(&rank,1,MPI_INT, (rank + 1) %
        size1, 10, MPI_COMM_WORLD);
MPI_Recv(&t,1,MPI_INT, (rank+size1-1) %
        size1,10,MPI_COMM_WORLD,&status
        );
}
Else
{
```

```
MPI_Recv(&t,1,MPI_INT, (rank+size1-
        1)%size1, 10, MPI_COMM_WORLD,
        &status);
MPI_Send(&rank,1,MPI_INT,(rank+1)%size1,
        10,MPI_COMM_WORLD);
}
```

```
printf("proc num %d@%s rcvd=%d from
        %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_7_2.exe  Exemplu _3_7_2.cpp
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpiCC -o HB_MPI_Spawn_Worker1.exe
HB_MPI_Spawn_Worker1.cpp
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpiCC  -o HB_MPI_Spawn_Worker2.exe
HB_MPI_Spawn_Worker2.cpp
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpiCC  -o HB_MPI_Spawn_Worker3.exe
HB_MPI_Spawn_Worker3.cpp
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpirun -n 1 -machinefile ~/nodes Exemplu _3_7_2.exe
===I am Manager (' Exemplu _3_7_2.exe'), run on the node 'compute-0-0.local' with rank 0 and
generate yhe following proceses:
5 procese run the module './HB_MPI_Spawn_Worker1.exe'
4 procese run the module './HB_MPI_Spawn_Worker2.exe'
3 procese run the module './HB_MPI_Spawn_Worker3.exe'
===
Module './HB_MPI_Spawn_Worker1.exe'. Start on the processor rank 2 of the node name 'compute-0-
0.local'of world_size  1
Module './HB_MPI_Spawn_Worker2.exe'. Start on the processor rank 8 of the node name 'compute-0-
2.local'of world_size  1
Module './HB_MPI_Spawn_Worker1.exe'. Start on the processor rank 0 of the node name 'compute-0-
0.local'of world_size  1
Module './HB_MPI_Spawn_Worker3.exe'. Start on the processor rank 10 of the node name 'compute-
0-2.local'of world_size  1
Module './HB_MPI_Spawn_Worker1.exe'. Start on the processor rank 4 of the node name 'compute-0-
1.local'of world_size  1
Module './HB_MPI_Spawn_Worker3.exe'. Start on the processor rank 11 of the node name 'compute-
0-4.local'of world_size  1
Module './HB_MPI_Spawn_Worker1.exe'. Start on the processor rank 1 of the node name 'compute-0-
0.local'of world_size  1
Module './HB_MPI_Spawn_Worker2.exe'. Start on the processor rank 7 of the node name 'compute-0-
2.local'of world_size  1
Module './HB_MPI_Spawn_Worker2.exe'. Start on the processor rank 6 of the node name 'compute-0-
1.local'of world_size  1
Module './HB_MPI_Spawn_Worker3.exe'. Start on the processor rank 9 of the node name 'compute-0-
2.local'of world_size  1
Module './HB_MPI_Spawn_Worker1.exe'. Start on the processor rank 3 of the node name 'compute-0-
1.local'of world_size  1
Module './HB_MPI_Spawn_Worker2.exe'. Start on the processor rank 5 of the node name 'compute-0-
1.local'of world_size  1
==Module './HB_MPI_Spawn_Worker1.exe'. proc num 1 @compute-0-0.local rcvd=0 from 0
==Module './HB_MPI_Spawn_Worker1.exe'. proc num 3 @compute-0-1.local rcvd=2 from 2
==Module './HB_MPI_Spawn_Worker1.exe'. proc num 2 @compute-0-0.local rcvd=1 from 1
==Module './HB_MPI_Spawn_Worker2.exe'. proc num 7 @compute-0-2.local rcvd=6 from 6
==Module './HB_MPI_Spawn_Worker1.exe'. proc num 4 @compute-0-1.local rcvd=3 from 3
==Module './HB_MPI_Spawn_Worker2.exe'. proc num 8 @compute-0-2.local rcvd=7 from 7
```

```
==Module './HB_MPI_Spawn_Worker2.exe'. proc num 5 @compute-0-1.local rcvd=4 from 4
==Module './HB_MPI_Spawn_Worker3.exe'. proc num 9 @compute-0-2.local rcvd=8 from 8
==Module './HB_MPI_Spawn_Worker2.exe'. proc num 6 @compute-0-1.local rcvd=5 from 5
==Module './HB_MPI_Spawn_Worker3.exe'. proc num 10 @compute-0-2.local rcvd=9 from 9
==Module './HB_MPI_Spawn_Worker1.exe'. proc num 0 @compute-0-0.local rcvd=11 from 11
==Module './HB_MPI_Spawn_Worker3.exe'. proc num 11 @compute-0-4.local rcvd=10 from 10
[Hancu_B_S@hpc]$
```

3.7.2 Comunicarea între procesele generate dinamic

MPI permite stabilirea unor „canale” de comunicare între procese, chiar dacă acestea nu împart un comunicator comun. Aceasta este utilă în următoarele situații:

- ✓Două părți ale unei aplicații, pornite independent, trebuie să comunice între ele.
- ✓Un instrument de vizualizare vrea să se atașeze la un proces în execuție.
- ✓Un server vrea să accepte conexiuni de la mai mulți clienți; serverul și clienții pot fi programe paralele.

Mai jos vom prezenta modalitățile de comunicare între procesul părinte și procesele fii generate.

Vom exemplifica comunicarea între procesul părinte şi procesele fiu prin intermediul intercomunicatorului (adică ei fac parte din grupuri diferite de procese) în cele ce urmează.

Exemplul 3.7.3 *Să se elaboreze un program MPI în limbajul C++ pentru generarea dinamică a proceselor, astfel încât procesul fiu va trimite rankul său procesului părinte utilizând un mediu de comunicare de tip intercomunicator.*

Mai jos este prezentat codul programului în limbajul C++ în care se realizează cele indicate în exemplul 3.7.3.

Codul programului e*xemplu_3_7_3.cpp* în care se generează procesele.

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>
int main(int argc, char** argv)
{
int rank, size, namelen, version, subversion,
    universe_size;
MPI_Comm family_comm;
char
    processor_name[MPI_MAX_PROCESS
    OR_NAME],
worker_program[100];
int rank_from_child,ich;
MPI_Status status;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,
    &rank);
MPI_Comm_size(MPI_COMM_WORLD,
    &size);
```

```
MPI_Get_processor_name(processor_name
        ,&namelen);
MPI_Get_version(&version,&subversion);
printf("I'm manager %d of %d on %s running
        MPI %d.%d\n", rank, size,
        processor_name, version, subversion);
if (size != 1) printf("Error: Only one manager
        process should be running, but %d
        were started.\n", size);
universe_size = 4;
strcpy(worker_program,"./HB_Woker_Com
        unication.exe");
printf("Spawning %d worker processes
        running %s\n", universe_size-1,
        worker_program);
MPI_Comm_spawn(worker_program,
        MPI_ARGV_NULL, universe_size-1,
        MPI_INFO_NULL,0,MPI_COMM_SELF,
        &family_comm,MPI_ERRCODES_IGNO
        RE);
/* comunicare cu procesul fiu */
for(ich=0;ich<(universe_size-1);ich++)
{
MPI_Recv(&rank_from_child,1,MPI_INT,ich,
        0,family_comm,MPI_STATUS_IGNORE)
        ;
printf("Received rank %d from child %d
        \n",rank_from_child,ich);
}
MPI_Bcast(&rank,1,MPI_INT,MPI_ROOT,
        family_comm);
MPI_Comm_disconnect(&family_comm);
MPI_Finalize();
return 0;
}
```

Codul programului *HB_Woker_Comunication.cpp* care va fi executat de procesele generate.

```
#include <mpi.h>
#include <stdio.h>
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main(int argc, char** argv)
{
int rank, size, namelen, version, subversion,
        psize;
int parent_rank;
MPI_Comm parent;
char
        processor_name[MPI_MAX_PROCESS
        OR_NAME];
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,
        &rank);
MPI_Comm_size(MPI_COMM_WORLD,
        &size);
MPI_Get_processor_name(processor_name
        ,&namelen);
MPI_Get_version(&version,&subversion);
printf("I'm worker %d of %d on %s running
        MPI %d.%d\n", rank, size,
        processor_name, version, subversion);
MPI_Comm_get_parent(&parent);
if (parent == MPI_COMM_NULL) {
        printf("Error: no parent process
        found!\n");
exit(1);
}
MPI_Comm_remote_size(parent,&psize);
if
(psize!=1)
{
printf("Error: number of parents (%d) should
        be 1.\n", psize);
exit(2);
}
/* comunicae cu procesul parinte */
 Int sendrank=rank;
printf("Worker %d:Success!\n", rank);
MPI_Send(&rank,1,MPI_INT,0,0,parent);
MPI_Bcast(&parent_rank,1,MPI_INT,0,
        parent);
printf("For Woker %d value of rank received
        from parent is %d \n", rank,
        parent_rank);
MPI_Comm_disconnect(&parent);
MPI_Finalize();
return 0;
}
```

Rezultatele posibile ale executării programului:

```
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpiCC -o Exemplu _3_7_3.exe Exemplu _3_7_3.cpp
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpiCC -o HB_Spawn_Comunication.exe
HB_Spawn_Comunication.cpp
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpirun -n 1 -machinefile ~/nodes Exemplu _3_7_3.exe
I'm manager 0 of 1 on compute-0-0.local running MPI 2.1
Spawning 3 worker processes running ./HB_Woker_Comunication.exe
I'm worker 0 of 3 on compute-0-0.local running MPI 2.1
Worker 0:Success!
I'm worker 2 of 3 on compute-0-0.local running MPI 2.1
Worker 2:Success!
Received rank 0 from child 0
I'm worker 1 of 3 on compute-0-0.local running MPI 2.1
Worker 1:Success!
Received rank 1 from child 1
Received rank 2 from child 2
For Woker 0 value of rank received from parent is 0
For Woker 2 value of rank received from parent is 0
For Woker 1 value of rank received from parent is 0
[Hancu_B_S@hpc]$
```

Pentru realizarea comunicării între procesul „părinte” și procesele „fii” prin intermediul uni mediu de comunicare de tip intracomunicator trebuie utilizată funcția **`MPI_Intercomm_merge.`** Prototipul funcției în limbajul C++ este

```
int MPI_Intercomm_merge(MPI_Comm intercomm, int
    high,MPI_Comm *newintracomm)
```

unde

IN **`intercomm`**	– numele intercomunicatorului;
IN **`high`**	– o variabilă logică care indică modul de reuniune a grupelor de procese;
OUT **`newintracomm`**	– numele intracomunicatorului.

Vom ilustra comunicarea între procesul părinte și procesele fii prin intermediul intracomunicatorului (adică ei fac parte din același grup de procese) prin următorul exemplu.

Exemplul 3.7.4 *Să se elaboreze un program MPI în limbajul C++ pentru generarea dinamică a proceselor, astfel încât procesul „fiu” va trimite rankul său procesului „părinte” utilizând un mediu de comunicare de tip intracomunicator.*

Mai jos este prezentat codul programului în limbajul C++ în care se realizează cele indicate în exemplul 3.7.4.

Codul programului *exemplu_3_7_4.cpp* în care se generează procesele.

```
#include <stdlib.h>
#include <string.h>
#include <mpi.h>
int main(int argc, char** argv)
{
int rank, size, namelen, version, subversion,
        universe_size;
int globalrank,sumrank;
MPI_Comm family_comm,allcomm;
char
        processor_name[MPI_MAX_PROCESS
        OR_NAME],
worker_program[100];
int rank_from_child,ich;
MPI_Status status;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,
        &rank);
MPI_Comm_size(MPI_COMM_WORLD,
        &size);
MPI_Get_processor_name(processor_name
        ,&namelen);
MPI_Get_version(&version,&subversion);
printf("I'm manager %d of %d on %s running
        MPI %d.%d\n",rank, size,
        processor_name,  version,
        subversion);
if (size != 1) printf("Error: Only one manager
        process should be running, but %d
        were started.\n", size);
universe_size = 4;
strcpy(worker_program,"./HB_Woker_Com
        unication_V1.exe");
printf("Spawning %d worker processes
        running %s\n", universe_size-1,
        worker_program);
MPI_Comm_spawn(worker_program,
        MPI_ARGV_NULL,universe_size-1,
        MPI_INFO_NULL,0,MPI_COMM_SELF,
        &family_comm,MPI_ERRCODES_IGNO
        RE);
MPI_Intercomm_merge(family_comm,1,
        &allcomm);
MPI_Comm_rank(allcomm, &globalrank);
printf("manager: global rank is %d,rank is
        %d \n", globalrank,rank);
MPI_Allreduce(&globalrank,&sumrank,1,
        MPI_INT,MPI_SUM,allcomm);
printf("sumrank after allreduce on process
        %d is %d \n", rank,sumrank);
MPI_Comm_disconnect(&family_comm);
MPI_Finalize();
return 0;
}
```

Codul programului *HB_Woker_Comunication_V1.cpp* care va fi executat de procesele generate.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main(int argc, char** argv)
{
int rank, size, namelen, version, subversion,
        psize;
int parent_rank;
int globalrank,sumrank;
MPI_Comm parent,allcom;
char
        processor_name[MPI_MAX_PROCESS
        OR_
        NAME];
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,
        &rank);
MPI_Comm_size(MPI_COMM_WORLD,
        &size);
MPI_Get_processor_name(processor_name
        ,&namelen);
MPI_Get_version(&version,&subversion);
printf("I'm worker %d of %d on %s running
        MPI %d.%d\n", rank, size,
        processor_name, version, subversion);
MPI_Comm_get_parent(&parent);
if (parent == MPI_COMM_NULL)
{
printf("Error: no parent process found!\n");
exit(1);
}
MPI_Comm_remote_size(parent,&psize);
if
(psize!=1)
{
printf("Error: number of parents (%d) should
        be 1.\n", psize);
exit(2);
}
MPI_Intercomm_merge(parent,1,&allcom);
MPI_Comm_rank(allcom,      &globalrank);
```

```
 printf("worker: globalrank is %d,rank is %d
        \n",globalrank, rank);
MPI_Allreduce(&globalrank,&sumrank,1,
        MPI_INT,MPI_SUM,allcom);
 printf("sumrank after allreduce on  process
        %d  is  %d  \n", rank,sumrank);
MPI_Comm_disconnect(&parent);
MPI_Finalize();
return 0;
}
```

Rezultatele posibile ale executării programului:

```
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpiCC -o Exemplu _3_7_4.exe Exemplu _3_7_4.cpp
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpiCC -o HB_Spawn_Comunication_V1.exe
HB_Spawn_Comunication_V1.cpp
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpirun -n 1 -machinefile ~/nodes Exemplu _3_7_4.exe
I'm manager 0 of 1 on compute-0-0.local running MPI 2.1
Spawning 3 worker processes running ./HB_Woker_Comunication_V1.exe
I'm worker 0 of 3 on compute-0-0.local running MPI 2.1
I'm worker 1 of 3 on compute-0-0.local running MPI 2.1
manager: global rank is 0,rank is 0
worker: globalrank is 1,rank is 0
sumrank after allreduce on  process  0  is  6
I'm worker 2 of 3 on compute-0-0.local running MPI 2.1
worker: globalrank is 3,rank is 2
worker: globalrank is 2,rank is 1
sumrank after allreduce on  process  2  is  6
sumrank after  allreduce  on  process  0  is  6
sumrank after allreduce on  process  1  is  6
[Hancu_B_S@hpc Finale]$
```

3.7.3 Exerciţii

1. Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++, prin care să se verifice dacă un process „fiu” poate, la rândul său, să genereze procese MPI.
2. Care este criteriul de verificare dacă au fost sau nu generate procesele de către procesul „părinte”?
3. Ce mediu de comunicare trebuie utilizat între procesul „părinte” şi procesele „fiu” pentru ca să nu existe două procese cu rankul *0*?
4. Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++, prin care să se verifice dacă procesul „părinte” şi procesele „fiu” pot utiliza aceeaşi operaţie de reducere.
5. Fie dat un şir de vectori X_k de lungimea *n*. Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++ în care:
   a. procesul „părinte” generează dinamic un număr l de procese, iniţializează vectorii $X_k, k = \overline{1,l}$ şi trimite procesului „fiu” vectorul X_k;

b. fiecare proces generat calculează norma vectorului X_k, adică $\|X_k\| = \sqrt{\sum_{i=1}^{n} x_i^2}$, după ce o trimite procesului părinte;
c. procesul părinte determină vectorul cu valoarea minimală a normei.

3.8 Tipuri de date MPI

3.8.1 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.

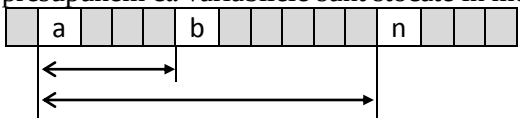
Typemap = {(type0 , disp0), . . . , (typen-1, dispn-1)}

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 foloseşte adresarea absolută în operațiile de comunicare.

3.8.2 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 3.8.1 *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 3.8.1.

```
#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
[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
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]$
```

3.8.3 Exerciţii

1. Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++ în care se creează un tip de date care reprezintă o linie a

unui masiv şi se distribuie o linie diferită tuturor proceselor comunicatorului **MPI_COMM_WORLD**.

2. Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++ în care procesul cu rankul 0 recepţionează de la toate procesele comunicatorului **MPI_COMM_WORLD** date de tip structură care constă din rankul procesului şi numele nodului pe care procesul este executat.
3. Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++ în care se construieşte o matrice transpusă utilizând proceduri de generare a tipurilor de date.
4. Fie dată o matrice pătratică de orice dimensiune. Să se creeze următoarele 3 tipuri de date:
   - elementele de pe diagonala principală;
   - elementele de pe diagonala secundară de jos;
   - elementele de pe diagonala secundară de sus.

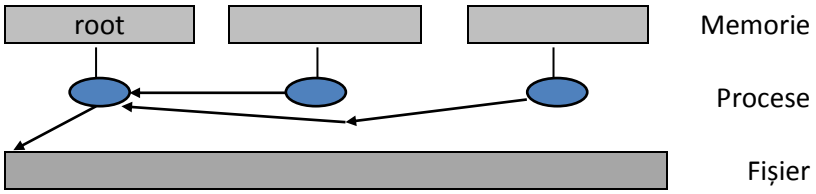
   Matricea este iniţializată de procesul cu rankul *0* şi prin funcţia **MPI_Brodcast** se transmit aceste tipuri de date tuturor proceselor.
5. Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++ în care să se copieze submatricea triunghiulară de jos a matricei *A* în submatricea triunghiulară de jos a matricei *B* utilizând funcţia **MPI_Type_indexed**.
6. Fie dată o matrice $A=\|a_{ij}\|_{\substack{i=1,m\\ j=1,n}}$ care este divizată în blocuri A_{kp} de dimensiunea $m_k \times n_p$. Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++ în care se creează un nou tip de date corespunzător submatricei A_{kp} şi procesul cu rankul 0 transmite acest tip de date procesului cu rankul $k \times p$.

3.9 Utilizarea fișierelor în MPI

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

În acest paragraf vom analiza următoarele modalităţi de utilizare a fişierelor în programarea paralelă.

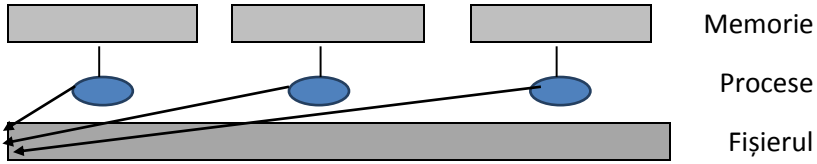
1. Utilizarea neparalelă a fișierelor – procesul root recepționează datele de la procese utilizând funcțiile MPI de transmitere/recepționare a mesajelor și apoi le scrie/citește în fișier. Acest mod schematic poate fi reprezentat astfel:



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;
}
```

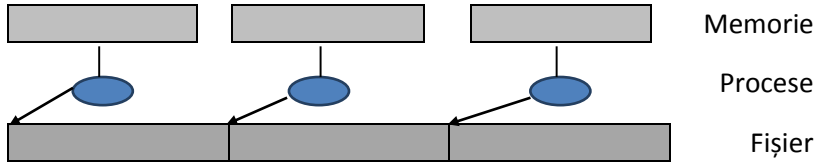
2. Fiecare proces utilizează în paralel fișierul său propriu. Acest mod schematic poate fi reprezentat astfel:



Această modalitate de utilizare a fişierelor 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.

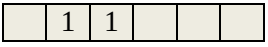
Vedere fișier (file view) – un set de date vizibile și accesibile dintr-un fișier deschis ca un set ordonat de e-tipuri. Fiecare proces are propria sa vedere fișier specificată de următorii trei parametri: offset, e-tip și f-tip. Șablonul descris de f-tip se repetă începând cu poziția offset.

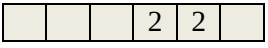
Offset – aceasta este poziția în fișier în raport cu vederea curentă, prezentă ca un număru de **e-tip**-uri. „Găurile” în **f-tipe** sunt omise la calcularea numărului poziției. Zero este poziția primului **e-tip** vizibil în vederea fișierului.

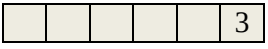
Referințe de fișier individuale – referințe de fișier care sunt locale pentru fiecare proces într-un fișier deschis.

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

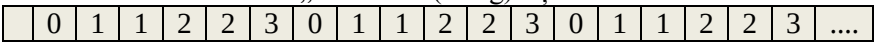
Vom reprezenta grafic noțiunile definite mai sus.



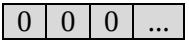


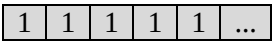


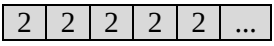
„Placarea” (tiling) fișierului

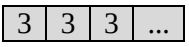


Astfel fiecare proces „va vedea” și deci va avea acces la următoarele date:









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**).

3.9.2 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 **comm**	– nume comunicator;
IN **filename**	– numele fișierului;
IN **amode**	– tipul de operații care se pot executa asupra fișierului;
IN **info**	– obiect informațional;
OUT **fh**	– descriptorul de fișier.

Valorile posibile ale parametrului **amode**:

MPI_MODE_RDONLY – accesibil numai la citire;
MPI_MODE_RDWR – accesibil la citire și înscriere;
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 **fh** – descriptorul de fișier;
IN **disp** – valoarea deplasării;
IN **etype** – tipul elementar de date;
IN **filetype** – tipul fișier de date;
IN **datarep** – modul de reprezentare a datelor din fișier;
IN **info** – 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 **fh** – descriptorul de fișier;
OUT **buf** – adresa de start a tamponului de date;
IN **int** – numărul de elemente din tamponul de date;
IN **datatype** – tipul de date din tamponul **buf**;
OUT **status** – 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 **fh** – descriptorul de fișier;
IN **buf** – adresa de start a tamponului de date;
IN **int** – numărul de elemente din tamponul de date;
IN **datatype** – tipul de date din tamponul **buf**;
OUT **status** – 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	**MPI_File_read_at** **MPI_File_write_at**	**MPI_File_read_at_all** **MPI_File_write_at_all**
	cu nonblocare	**MPI_File_iread_at** **MPI_File_iwrite_at**	**MPI_File_read_at_all_begin** **MPI_File_read_at_all_end** **MPI_File_write_at_all_begin** **MPI_File_write_at_all_end**
Poziționare individuală	cu blocare	**MPI_File_read** **MPI_File_write**	**MPI_File_read_all** **MPI_File_write_all**
	cu nonblocare	**MPI_File_iread** **MPI_File_iwrite**	**MPI_File_read_all_begin** **MPI_File_read_all_end** **MPI_File_write_all_begin** **MPI_File_write_all_end**
Poziționare colectivă	cu blocare	**MPI_File_read_shared** **MPI_File_write_shared**	**MPI_File_read_ordered** **MPI_File_write_ordered**
	cu nonblocare	**MPI_File_iread_shared** **MPI_File_iwrite_shared**	**MPI_File_read_ordered_begin** **MPI_File_read_ordered_end** **MPI_File_write_ordered_begin** **MPI_File_write_ordered_end**

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

Exemplul 3.9.1 *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.*

Mai jos este prezentat codul programului în limbajul C++ în care se realizează cele indicate în exemplul 3.9.1.

```
#include "mpi.h"
#include <mpi.h>
#include <stdio.h>
int main(int argc,char**argv) {
int rank,count,x,amode,i,j;
MPI_File OUT;
MPI_Aint rowsize;
MPI_Datatype etype,ftype0,ftype1,ftype2;
MPI_Status *status;
int value[4][8];
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,&ra
        nk);
if (rank ==0)
printf("\n=====REZULTATUL PROGRAMULUI
        '%s' \n",argv[0]);
MPI_Barrier(MPI_COMM_WORLD);
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);
if (rank==0) {
MPI_File_set_view(OUT,0,etype,ftype0,
        "native",MPI_INFO_NULL);
                        }
if (rank==1) {
MPI_File_set_view(OUT,0,etype,ftype1,
        "native",MPI_INFO_NULL);
                        }
if (rank==2) {
MPI_File_set_view(OUT,0,etype,ftype2,
        "native",MPI_INFO_NULL);
                        }
for (i=1;i<=4;++i)
for (j=1;j<=8;++j)
value[i-1][j-1]= 10*i+j;
count=0;
if (rank==0)
{
MPI_File_write(OUT,&value[rank][0],1,
        MPI_ROW,&state);
MPI_Get_count(&state,MPI_ROW,&x);
count=count+x;
MPI_File_write(OUT,&value[rank+1][0],1,
        MPI_ROW,&state);
MPI_Get_count(&state,MPI_ROW,&x);
count=count+x;
printf("Procesul %d a realizat %d
        inscrieri(etypes) \n",rank,count);
}
if(rank > 0 && rank <3)
{
```

```
MPI_File_write(OUT,&value[rank+1][0],1,
        MPI_ROW,&state);
MPI_Get_count(&state,MPI_ROW,&count);
printf("Procesul %d a realizat %d
        inscrieri(etypes) \n",rank,count);
}
if (rank==3) {
MPI_File_set_view(OUT,0,etype,ftype0,
        "native",MPI_INFO_NULL);
int *buff1 = (int*)malloc(sizeof(int)*16);
MPI_File_read(OUT,buff1,2,MPI_ROW,
        status);
printf("===Procesul %d a citit din fisier
        rindurile\n",rank);
for(int i = 0; i!=2; ++i)
{
        for(int j = 0; j != 8; ++j)
             printf("%5d",buff1[i*8 + j]);
        printf("\n");
}
                        }
if (rank==4) {
MPI_File_set_view(OUT,0,etype,ftype1,
        "native",MPI_INFO_NULL);
int buff2[8];
MPI_File_read(OUT,buff2,1,MPI_ROW,
        status);
printf("===Procesul %d a citit din fisier
        rindurile\n", rank);
        for(int j = 0; j != 8; ++j)
             printf("%5d",buff2[j]);
        printf("\n");
                        }
if (rank==5) {
MPI_File_set_view(OUT,0,etype,ftype2,
        "native",MPI_INFO_NULL);
int *buff3 = (int*)malloc(sizeof(int)*8);
MPI_File_read(OUT,buff3,1,MPI_ROW,
        status);
printf("===Procesul %d a citit din fisier
        rindurile\n",rank);
        for(int j = 0; j != 8; ++j)
             printf("%5d",buff3[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.cpp
[Hancu_B_S@hpc]$ /opt/openmpi/bin/mpirun -n 6 -machinefile ~/nodes4  Exemplu_3_9_1.exe
=====REZULTATUL PROGRAMULUI 'Exemplu_3_9_1.exe'
Procesul 1 a realizat 1 inscrieri(etypes)
Procesul 2 a realizat 1 inscrieri(etypes)
Procesul 0 a realizat 2 inscrieri(etypes)
===Procesul 4 a citit din fisier rindurile
  31  32  33  34  35  36  37  38
===Procesul 5 a citit din fisier rindurile
  41  42  43  44  45  46  47  48
===Procesul 3 a citit din fisier rindurile
  11  12  13  14  15  16  17  18
  21  22  23  24  25  26  27  28
```

Fișierul array.dat

```
[Hancu_B_S@hpc]$ od -d array.dat
0000000   11    0   12    0   13    0   14    0
0000020   15    0   16    0   17    0   18    0
0000040   21    0   22    0   23    0   24    0
0000060   25    0   26    0   27    0   28    0
0000100   31    0   32    0   33    0   34    0
0000120   35    0   36    0   37    0   38    0
0000140   41    0   42    0   43    0   44    0
0000160   45    0   46    0   47    0   48    0
```

3.9.3 Exerciţii

1. Care sunt etapele principale pentru utilizarea fişierelor în programe MPI?
2. Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++ în care se realizează condiţiile enunţate în exemplul 3.4.2 şi procesul de distribuire a liniilor matricei este substituit prin operaţiile I/O, adică utilizarea fişierelor.
3. Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++ în care se realizează condiţiile enunţate în exemplul 3.4.3 şi procesul cu rankul 0 citeşte dimensiunea şi elementele matricei dintr-un fişier. Numele fişierului se indică ca parametru utilitarului **mpirun**.
4. Fie dată o matrice $A=\|a_{ij}\|_{\substack{i=1,m\\ j=1,n}}$ care este divizată în blocuri A_{kp} de dimensiunea $m_k \times n_p$. Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++ în care fiecare proces cu rankul $k \times p$ din comunicatorul **`MPI_COMM_WORLD`** înscrie în acelaşi fişier submatricea A_{kp}. Procesul cu rankul *0* iniţializează matricea *A*.
5. Fie dat un fişier care conţine blocuri A_{kp} de dimensiunea $m_k \times n_p$ (a se vedea exerciţiul 4). Să se elaboreze şi să se execute pe clusterul USM un program MPI în limbajul C++ în care fiecare proces cu rankul $k \times p$ din comunicatorul **`MPI_COMM_WORLD`** citeşte din acelaşi fişier submatricea A_{kp}. Procesul cu rankul *0* iniţializează matricea *A* şi culege de la celelalte procese submatricele A_{kp} după ce tipăreşte matricea obţinută.

Bibliografie

1. B. Wilkinson, M. Allen. *Parallel Programming*. Printice-Hall, 1999.
2. B. Dumitrescu. *Algoritmi de calcul paralel*. Bucureşti, 2001.
3. Gh. Dodescu, B. Oancea, M. Raceanu. *Procesare paralelă.* Bucureşti: Economica, 2002.
4. Gh.M. Panaitescu. *Arhitecturi paralele de calcul*. Ploieşti: Universitatea "Petrol-Gaze", 2007.
5. R.W. Hockney, C.R. Jesshope. *Calculatoare paralele*: *Arhitectură, programare şi algoritmi*. Bucureşti, 1991.
6. http://www.mpi-forum.org/docs/docs.html.
7. *MPI-2. Extension to the Message-Passing Interface*: http://www.mpi-forum.org/docs/mpi20-html/
8. P.S. Pacheco. *An Introduction to Parallel Programming*. 2011. pp. 370. www.mkp.com or www.elsevierdirect.com
9. Ph.M. Papadopoulos. *Introduction to the Rocks Cluster Toolkit – Design and Scaling*. San Diego Supercomputer Center, University of California, San Diego, http://rocks.npaci.edu
10. А.А. Букатов, В.Н. Дацюк, А.И. Жегуло. *Программирование многопроцессорных вычислительных систем*. Ростов-на-Дону: Издательство ООО ЦВВР, 2003.
11. А.С. Антонов. *Параллельное программирование с использованием технологии MPI*. Издательство Московского университета, 2004.
12. В.П. Гергель, Р.Г. Стронгин. *Основы параллельных вычислений для многопроцессорных вычислительных систем*. Нижний Новгород: Издательство Нижегородского госуниверситета, 2003.
13. В.В. Воеводин. *Парралельные вычисления*. Москва, 2000.
14. Г.И. Шпаковски, Н.В. Сериков. *Программирование для многопроцессорных системах в стандарте MPI*, 2003.
15. В.П. Гергель. *Теория и практика параллельных вычислений*. http://www.software.unn.ac.ru/ccam/kurs1.htm
16. *Краткое руководство пользователяпо работе с вычислительным кластером ТТИ ЮФУ*. Таганрог, 2010. http://hpc.tti.sfedu.ru
17. М.Л. Цымблер, Е.В. Аксенова, К.С. Пан. *Задания для практических работ и методические указания по их выполнению по дисциплине „Технологии параллельного программирования"*. Челябинск, 2012.

Anexă

Vom prezenta o mostră de lucrare de laborator pentru implementarea soft pe clusterul USM a unui algoritm paralel.

Lucrare de laborator. ***Să se elaboreze un program MPI în limbajul C++ pentru determinarea în paralel a mulţimii tuturor situaţiilor de echilibru în strategii pure pentru un joc bimatriceal.***

Fie dat un joc bimatriceal $\Gamma = \langle I, J, A, B \rangle$ unde I – mulţimea de indici ai liniilor matricelor, J – mulţimea de indici ai coloanelor matricelor, iar $A = \|a_{ij}\|_{\substack{i \in I \\ j \in J}}$, $B = \|b_{ij}\|_{\substack{i \in I \\ j \in J}}$ reprezintă matricele de câştig ale jucătorilor.

Situaţia de echilibru este perechea de indici (i^*, j^*), pentru care se verifică sistemul de inegalităţi:

$$(i^*, j^*) \Leftrightarrow \begin{cases} a_{i^* j^*} \geq a_{ij^*} \ldots \forall i \in I \\ b_{i^* j^*} \geq b_{i^* j} \ \forall j \in J \end{cases}$$

Vom spune că *linia i strict domină linia k* în matricea *A* dacă şi numai dacă $a_{ij} > a_{kj}$ pentru orice $j \in J$. Dacă există *j* pentru care inegalitatea nu este strictă, atunci vom spune că linia *i* domină linia *k*. Similar, vom spune: *coloana j strict domină coloana l* în matricea *B* dacă şi numai dacă $b_{ij} > b_{il}$ pentru orice $i \in I$. Dacă există i pentru care inegalitatea nu este strictă, atunci vom spune: *coloana j domină coloana l.*

Algoritmul de determinare a situaţiei de echilibru

a) Eliminarea, în paralel, din matricea *A* şi *B* a liniilor care sunt dominate în matricea *A* şi din matricea *A* şi *B* a coloanelor care sunt dominate în matricea *B*.

b) Se determină situaţiile de echilibru pentru matricea (A', B'), $A' = \|a'_{ij}\|_{\substack{i \in I' \\ j \in J'}}$ şi $B' = \|b'_{ij}\|_{\substack{i \in I' \\ j \in J'}}$ obţinută din pasul a). Este clar că $|I'| \leq |I|$ si $|J'| \leq |J|$.

- Pentru orice coloană fixată în matricea A' notăm (evidenţiem) toate elementele maximale după linie. Cu alte cuvinte, se determină $i^*(j) = \arg\max_{i \in I'} a'_{ij}$ pentru orice $j \in J'$. Pentru aceasta

se va folosi funcţia **MPI_Reduce** şi operaţia **ALLMAXLOC**[2] (a se vedea exerciţiul 4 din paragraful 3.4.4).

- Pentru orice linie fixată în matricea B' notăm toate elementele maximale de pe coloane. Cu alte cuvinte, se determină $j^*(i) = \arg\max_{j \in J'} b'_{ij}$ pentru orice $i \epsilon I'$. Pentru aceasta se va folosi funcţia **MPI_Reduce** şi operaţia **ALLMAXLOC** (a se vedea Exerciţiul 4 din paragraful 3.4.4).
- Selectăm acele perechi de indici care concomitent sunt selectate atât în matricea A' cât şi în matricea B'. Altfel spus, se determină $\begin{cases} i^* \equiv i^*(j^*) \\ j^* \equiv j^*(i^*) \end{cases}$.

c) Se construiesc situaţiile de echilibru pentru jocul cu matricele iniţiale A şi B.

Vom analiza următoarele exemple.

Exemplul 1. Situaţia de echilibru se determină numai în baza eliminării liniilor şi a coloanelor dominate. Considerăm următoarele matrici:

$$A = \begin{pmatrix} 400 & 0 & 0 & 0 & 0 & 0 \\ 300 & 300 & 0 & 0 & 0 & 0 \\ 200 & 200 & 200 & 0 & 0 & 0 \\ 100 & 100 & 100 & 100 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -100 & -100 & -100 & -100 & -100 & -100 \end{pmatrix}.$$

$$B = \begin{pmatrix} 0 & 200 & 100 & 0 & -100 & -200 \\ 0 & 0 & 100 & 0 & -100 & -200 \\ 0 & 0 & 0 & 0 & -100 & -200 \\ 0 & 0 & 0 & 0 & -100 & -200 \\ 0 & 0 & 0 & 0 & 0 & -200 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

[2] În cazul utilizării operaţiei MAXLOC rezultatele pot fi incorecte.

Vom elimina liniile şi coloanele dominate în următoarea ordine: *linia 5, coloana 5, linia 4, coloana 4, coloana 3, linia 3, coloana 0, linia 0, coloana 1, linia 1*. Astfel obţinem matricele $A' = (200)$, $B' = (0)$ şi situaţia de echilibru este $(i^*, j^*) = (2,2)$ şi câştigul jucătorului *1* este *200*, al jucătorului *2* este *0*.

Exemplul 2. Considerăm următoarele matrice $A = \begin{pmatrix} 2 & 0 & 1 \\ 1 & 2 & 0 \\ 0 & 1 & 2 \end{pmatrix}, B = \begin{pmatrix} 1 & 0 & 2 \\ 2 & 1 & 0 \\ 0 & 2 & 1 \end{pmatrix}$. În matricea *A* nu există linii dominate, în matricea *B* nu există colane dominate. Pentru comoditate, vom reprezenta acest joc astfel: $AB = \begin{pmatrix} (\underline{2},1) & (0,0) & (1,\underline{2}) \\ (1,\underline{2}) & (\underline{2},1) & (0,0) \\ (0,0) & (1,\underline{2}) & (\underline{2},1) \end{pmatrix}$. Uşor se observă că în acest joc nu există situaţii de echilibru în strategii pure.

Exemplul 3. Considerăm următoarele matrice: $A = \|a_{ij}\|_{i \in I}^{j \in J}$, $B = \|b_{ij}\|_{i \in I}^{j \in J}$,

unde $a_{ij} = c, b_{ij} = k$ pentru orice $i \in I, j \in J$ şi orice constante *c,k*. Atunci mulţimea de situaţii de echilibru este $\{(i, j) : \forall i \in I, \forall j \in I\}$.

Pentru realizarea acestui algoritm pe clustere paralele sunt obligatorii următoarele:

1) Paralelizarea la nivel de date se realizează astfel:
   a) Procesul cu rankul 0 iniţializează valorile matricelor *A* şi *B*. Dacă *n>5, m>5*, atunci procesul cu rankul *0* citeşte matricele dintr-un fişier text.
   b) Distribuirea matricelor pe procese se face astfel încât să se realizeze principiul *load balancing*.
2) Paralelizarea la nivel de operaţii se realizează şi prin utilizarea funcţiei **`MPI_Reduce`** şi a operaţiilor nou create.

Boris HÂNCU, Elena CALMÎȘ

MODELE DE PROGRAMARE PARALELĂ PE CLUSTERE
PARTEA I. PROGRAMARE MPI

Note de curs

Redactare: Ana Ferafontov
Machetare computerizată:

Bun de tipar . Formatul
Coli de tipar . Coli editoriale .
Comanda 53. Tirajul 50 ex.

Centrul Editorial – Poligrafic al USM
str. Mateevici, 60, Chișinău, MD-2009