4.MPI – interfața pentru programarea paralelă a sistemelor cu memorie distribuită

Specificaţia *MPI (Message Passing Interface*) cunoaşte diverse implementări (API-uri MPI) sub forma unor biblioteci ce conţin un set larg (peste 250) de rutine pentru schimbul de mesaje, managementul datelor şi proceselor, rutine ce pot fi utilizate pentru un spectru variat de aplicaţii ştiinţifice implementabile eficient pe diverse sisteme paralele şi/ sau distribuite.

4.1. Obiective

- Studiul bibliotecii de programare paralelă a sistemelor cu memorie distribuită bazată pe mesaje MPI
- Procese MPI şi tipuri de comunicaţie
- Comunicatori, grupuri de procese, topologii virtuale, comunicația colectivă-mecanisme suport pentru procesare paralelă performantă
- Implementarea unor algoritmi paraleli în modelul transferului de mesaje

4.2. Concepte

Standardul MPI defineşte o *interfaţă de programare paralelă prin schimb explicit de mesaje*, ce poate fi utilizată pentru scrierea de aplicaţii pentru sisteme de tip MIMD (multiple instructions multiple data) cu memorie distribuită / partajată, respectiv pentru sisteme SIMD (Single instruction multiple data) ce presupun paralelizare prin decompoziţia datelor. MPI este o bibliotecă de funcţii, ea defineşte nume de funcţii, secvenţe de apel şi rezultatele pentru un set de funcţii care pot fi apelate în programe C, C++ şi Fortran 90. Compilarea programelor poate fi realizată cu un compilator standard, sau cu diverse alte compilatoare performante, iar în faza de editare a legăturilor este introdusă şi biblioteca MPI.

Principalele caracteristici care au determinat o largă răspândire a standardului sunt:

- oferă o interfaţă de programare pentru aplicaţii paralele/distribuite
- oferă performantă, portabilitate şi uşurinţă în folosire, semantica sa permite implementări diverse;
- semantica interfeţei este independenta de limbaj (C, C++, Fortran)
- permite comunicaţie eficientă *punct-la-punct şi punct-multipunct* în medii eterogene (reţele diverse);
- conţine un set de rutine pentru lucrul cu *grupuri de procese* , permite scheme de adresare complexe.
- permite *comunicaţie colectivă şi operaţii de calcul global* ,ce conferă scalabilitate sistemului
- oferă contexte de comunicare şi comunicatori pentru dezvoltarea de biblioteci de funcţii necesare programării paralele;
- conţine rutine pentru definirea de topologii virtuale de grup;
- oferă simplitate dar şi completitudine bazat pe un număr extrem de mare de funcţii, însă aplicaţii relativ complexe pot fi programate folosind un *set minimal de 6 funcţii esenţiale.*

4.2.1. Modelul de comunicație

Soluţia oferită de MPI constă în specificarea mesajelor la un nivel înalt de abstractizare, capabil să reflecte faptul că transmisia unui mesaj necesită o structurare mai complexă decât un simplu string de biţi, astfel structura unui mesaj devine *adresa, contor, tip_date*. Termenul de *buffer de mesaj* poate fi utilizat cu semnificaţii multiple, astfel:

- poate referi o zonă de memorie *specificată în aplicaţie* pentru stocarea datelor sau utilizatorul poate seta o zonă de memorie utilizată ca o zonă intermediară pentru memorarea unor mesaje arbitrare ce apar în aplicaţie
- poate reprezenta o zonă de memorie creată şi gestionată de *sistemul MPI*

Sincron vs asincron. Mesajul poate fi transferat direct între bufferele declarate în aplicaţia utilizator, sau pentru comunicaţie asincronă, mesajul este *stocat temporar* într-un buffer creat dinamic de sistem înaintea depozitării sale în bufferul receptor, respectiv utilizatorul poate declara iniţial un bufer suficient de mare astfel încât să permită transferul oricărui mesaj ce necesită stocare intermediară şi utilizare în operaţia de transfer. Comunicare prin transfer sincron de mesaje defineşte modul de comunicare ce poate fi regăsit în limbajul teoretic de comunicare prin transfer de mesaje al lui Hoare- CSP. Cele două procese corespunzătoare transmiterii–recepţiei de mesaje se încheie doar atunci când mesajul transmis este şi recepţionat. Avantajele pe care le prezintă referă semantica simplă şi uşor de utilizat. Un proces care revine din subrutina send va şti cu certitudine că mesajul trimis a fost recepţionat. De asemenea un avantaj important îl constituie faptul că nici sistemul şi nici utilizatorul nu trebuie să menţină un buffer pentru mesaj, acesta poate fi copiat direct în spaţiul de adrese al procesului receptor. Datorită simplităţii ,un număr mare de sisteme paralele implementează diferite variante ale acestui mod de comunicare.

Comunicarea bazată pe emisie - recepţie blocantă este caracterizată de transmiterea unui mesaj în mod blocant, executată *fără a aştepta recepţia acestuia* de către procesul destinaţie, astfel este posibilă încheierea execuţiei unei operaţii de transmisie chiar înaintea începerii execuţiei operaţiei de recepţie corespunzătoare. Tipurile de comunicare sunt prezentate in tabelul 1.

Evenimentul de comunicaţie	**Sincron**	**Blocant**	**Nonblocant**
Revenirea din send este indicată	Mesaj recepţionat	Mesaj transmis	Transmitere de mesaj iniţiată
Bufferare mesaj	Nu e necesară	E necesară	E necesară
Verificarea stării	Nu e necesară	Nu e necesară	E necesară
Încărcare generată de aşteptare	Mare	Medie	Mică
Suprapunere comunicaţie şi procesare	Nu	Da	Da

Tabel 4.1. Moduri de comunicare aflate la baza primitivelor MPI

Comunicare bazată pe emisie - recepţie neblocantă. Transmisia neblocantă se încheie *imediat după ce a fost notificat sistemul* de existenţa unui mesaj de transmis, respectiv recepţionat, mesaj care poate fi în tranzit, deja sosit sau chiar netransmis încă. Pentru această variantă, sistemul trebuie să implementeze un *buffer temporar*. Implementările performante conţin funcţii specializate pentru verificarea *stării bufferului* sau pentru implementarea unor mecanisme de aşteptare pană ce procesul poate continua (tip *wait-for).* Bufferul necesar transferului de mesaje poate fi implementat de către *sistemul de transfer al mesajelor sau mod utilizator*, la nivelul aplicaţiei.

Avantajele oferite de comunicaţia neblocantă pot fi exploatate în contextul utilizării suportului hardware de comunicaţie dedicat, astfel încărcarea indusă de comunicaţie poate fi aproape integral mascată prin operaţii neblocante. Operaţiile neblocante pot fi utilizate şi cu un protocol bufferat, astfel emiţătorul iniţiază operaţia DMA şi revine, însă datele sunt disponibile accesului (sunt sigure) doar după ce operaţia DMA s-a încheiat.

Tipuri de date MPI. O caracteristică deosebit de puternică a bibliotecii o constituie posibilitatea introducerii de argumente sub forma unor tipuri de date variate, posibil combinate, pentru toate mesajele trimise şi recepţionate. MPI asigură un set deosebit de complex de tipuri de date predefinite, set care include toate tipurile de bază din C, C++ şi Fortran, alături de două tipuri de date specifice MPI:MPI_BYTE şi MPI_PACKED.

Context de comunicaţie. Fiecare comunicare de mesaj se derulează într-un anumit context. Mesajele sunt întotdeauna primite în contextul în care au fost transmise, iar mesaje transmise în contexte diferite nu interferă.

Proces şi grup de procese. Procesul MPI este unitatea fundamentală de calcul, fiind constituit dintr-un *fir de control independent* şi un *spaţiu de adrese privat*, astfel un proces nu poate accesa direct variabilele definite în spaţiul de adrese al altui proces, comunicarea fiind bazată pe transfer de mesaje. Contextul este partajat de un grup de procese. Procesele MPI se execută fără a avea însă mecanisme de încărcare a codului pe procesoare, atribuire a proceselor procesoarelor sau mecanisme de creare şi distrugere de procese. Aceste mecanisme sunt de obicei, în cadrul implementărilor curente, *preluate de la sistemul de operare pe care rulează* procesul MPI, fapt ce poate crea dificultăţi de portare a acestor biblioteci între sisteme de operare diferite. MPI este proiectat a fi thread-safe, iar grupurile de procese definite în MPI sunt dinamice, apartenenţa la grup este statică însă suportă ataşarea-detaşarea din grup. Grupurile se pot suprapune fapt ce permite unui proces să fi membru al mai multor grupuri simultan. Procesele sunt identificate în cadrul grupului printr-un întreg (rank), care ia valori între 0 şi n-1, unde n este numărul proceselor din grup. Iniţializarea unei aplicaţii MPI folosind rutina de iniţializare MPI_INIT, creează un singur grup, numit MPI_COMM_WORLD, care include toate procesele MPI ce formează aplicaţia.

Procesele pot utiliza *comunicaţie punct-la-punct* cu scopul transmisiei- recepţionării de mesaje utilă implementării comunicaţiilor locale nestructurate. De asemenea un grup de procese poate apela operaţii de tip *comunicaţii colective* pentru a simplifica implementarea unor operaţii globale, deşi standardul nu oferă în mod explicit suport pentru multithreading, unele implementări includ mecanisme de acest fel.

Comunicatorul integrează informaţiile de context şi informaţiile de identificare a grupului. MPI_INIT defineşte pentru fiecare proces apelat comunicatorul implicit MPI_COMM_WORLD. Toate comunicaţiile MPI necesită un comunicator ca argument, iar procesele MPI nu pot comunica decât dacă *partajează* un comunicator. Fiecare comunicator identifică un grup reprezentând o listă de procese, fiecare proces este numerotat, identificatorul fiecăruia numindu-se rang (*rank*). Acest rang identifică în mod unic un proces şi poate fi folosit pentru specificarea sursei sau destinaţiei unui mesaj. Folosind MPI_COMM_WORLD, fiecare proces poate comunica cu oricare altul, iar grupul MPI_COMM_WORLD reprezintă mulţimea tuturor proceselor MPI. Informaţiile specifice comunicatorului pot fi accesibile prin funcţiile:

MPI_COMM_RANK (MPI_comm comm, int *rank)
Funcţia MPI_COMM_RANK returnează rangul procesului apelant în grupul comunicatorului *comm*.
MPI_COMM_SIZE (MPI_comm comm, int *size)
Funcţia MPI_COMM_SIZE returnează în variabila *size* numărul de procese din grupul asociat comunicatorului *comm*. Fiecare proces aparţinând unui comunicatori este identificat prin rangul său.

4.2.2. Modelul computaţional

Stilul de programare care poate fi folosit cu MPI se bazează fie pe modelul **MIMD** (Multiple instructions multiple data) ce presupune existenţa de programe diferite pentru fiecare procesor, fie pe modelul **SPMD** (Single Program Multiple Data), caz în care există un singur program, astfel procesele MPI provin din acelaşi program, dar execută porţiuni diferite de cod selectate pe baza unor instrucţiuni de condiţie. Standardul MPI a fost proiectat astfel încât programatorul începător să nu simtă complexitatea de la primele programe. Programarea unei aplicaţii minimale MPI se poate face

cunoscând doar un set minimal de 6 funcţii ce reprezintă nucleul suficient pentru scrierea unei aplicaţii:

MPI_Init (int *argc, char**argv) este funcţia care iniţializează variabilele locale MPI şi înştiinţează sistemul global MPI despre execuţia unui nou proces MPI. Funcţia acţionează şi ca barieră de iniţializare, în sensul ca nici un proces MPI nu-şi poate continua execuţia până când toate celelalte procese nu au executat partea lor de iniţializare, argc/argv sunt argumentele la nivel de linie de comanda pentru programul C.
MPI_Comm_rank(), returnează identificatorul procesului MPI în cadrul grupului specificat în lista de parametri.
MPI_Comm_size**() î**ntoarce numărul de procese care fac parte din grupul specificat ca parametru
MPI_Send() şi MPI_Recv() sunt funcţiile de bază care implementează comunicaţia prin mesaje între procese**.**
MPI_Finalize() pentru anunţarea sistemului MPI despre terminarea procesului MPI cu succes (valoarea lui ierr=0), această rutină este apelată de fiecare proces MPI. După această funcţie nici o altă funcţie nu poate fi apelată, toate comunicaţiile în aşteptare trebuie încheiate înaintea apelului ei.

Apelul funcţiilor de bibliotecă necesită includerea antetului **mpi.h** ce conţine constantele referite în rutinele MPI (tipuri de date, comunicatori, erori) şi interfeţele funcţiilor MPI. O operaţie este considerată *terminată local* în cadrul unui anumit proces, dacă acesta a terminat partea sa ,atât în comunicaţia punct-la-punct, cât şi în cea colectivă. O operaţie de comunicare este *terminată global*, dacă toate procesele implicate în comunicare au *terminat local* părţile ce le revin din aceasta.

4.2.3. Modul de comunicaţie punct-la-punct

O aplicaţie poate reduce latenţa de comunicare presupunând că platformele hardware pe care este realizată implementarea includ mecanisme care permit *rutarea şi gestiunea mesajelor paralel cu alte procesări*. Biblioteca MPI garantează respectarea următorului set de proprietăţi (reguli semantice): pentru comunicarea punct la punct, astfel: garantarea preluării mesajelor în ordinea în care au fost trimise, perechile send şi receive nu pot rămâne în permanenţă nerezolvate (necorelate).
Mesajele transmise reprezintă o secvenţă de articole de acelaşi tip, identificarea mesajelor realizându-se prin identificatorul **sursei (rank) şi o etichetă (tag)** ataşată mesajului, valori interpretate relativ, în cadrul zonei de comunicaţie identificată de un anumit comunicator. Există situaţii în care este necesar ca recepţia să fie efectuată indiferent de sursă (MPI_ANY_SOURCE) sau de etichetă (MPY_ANY_TAG).

Programele paralele necesită implementarea unor soluţii algoritmice capabile să minimizeze timpul de răspuns, iar MPI oferă propriile mecanisme pentru instrumentarea codului astfel încât să permită analiza performanţei. Funcţia **MPI_Wtime**, returnează o valoare dublă precizie a numărului de secunde raportat la un anumit moment de timp trecut, garantat nemodificabil pe durata execuţiei programului, fapt ce permite inserarea diverselor apeluri ale rutinei în codul sursă pentru măsurarea unor intervale de timp de execuţie. Funcţiile bibliotecii MPI utilizate în analiza performanţelor bazat pe elementul timp de procesare sunt **MPI_Wtime (void), MPI_Wtick(void).** Lista parametrilor utilizaţi în rutinele MPI şi semnificaţia lor,respectiv lista codurilor de eroare sunt prezentate în tabelul 4.2.:

ierror	**Cod de eroare MPI**
Comm	Identificarea comunicatorului utilizat în transmisia mesajului
Rank	Rangul procesului apelant în grup sau comunicator
buffer	Adresa de start a buferului ce conţine mesajul(transmis-receptionat)
count	Număr de articole specificate în bufer (număr maxim de articole receopţionate)
datatype	Tipul de dată a fiecărui element transmis

parametru	Semnificaţie
Root	Rangul unui proces specific (sursa sau destinaţia pot fi identice)
Size	Numărul de procese din grup
Tag	Tagul de mesaj transmis-recepţionat
source	Rangul procesului sursă în comunicator
dest	Rangul procesului destinaţie în comunicator
status	Tablou de întregi ce conţine starea, referă o modul de recepţie a mesajului (sursa, tagul şi codul de eroare)

Tabel 4.2. Coduri de eroare şi parametrii rutinelor MPI

Pentru a putea construi o aplicaţie paralelă performantă este necesar un control performant asupra transmiterii mesajelor. În acest sens, MPI defineşte patru moduri de comunicare: **standard, sincron, bufferat şi ready** (vezi tabelul 4.3.). Funcţia de comunicare se obţine prin combinarea modului de comunicaţie cu un mod de blocare, moduri ce vor fi detaliate la secţiunea funcţii pentru comunicaţie. Toate modurile de comunicaţie există atât în forma blocantă cât şi neblocantă, în forma blocantă, întoarcerea dintr-o funcţie implică terminarea ei cu succes MPI nu impune restricţii asupra modului de potrivire a funcţiilor de transmisie cu cele de recepţie, existând astfel mai multe posibile combinaţii a căror funcţionare corectă intră însă în sarcina programatorului. MPI defineşte două moduri de comunicare:

- fără blocare (imediat), caz în care funcţiile de transmisie sau recepţie se termină înainte ca informaţia să fie trimisă/recepţionată, iar bufferul indicat într-o funcţie neblocantă nu trebuie folosit până la detectarea sfârşitului funcţiei iniţiate.
- cu blocare, caz în care după terminarea funcţiei, bufferul de transmisie sau recepţie poate fi refolosit.

Tip	Descriere
Send sincron	operaţia de trimitere poate începe oricând, dar nu se poate termina decât atunci când mesajul a ajuns la receptor
Send buferat	utilizatorul poate furniza sistemului un buffer pentru a permite ca operaţia de trimitere să se termine întotdeauna înainte ca mesajul să fie recepţionat la destinaţie.
Send standard	operaţia de trimitere poate începe chiar dacă operaţia corespunzătoare de recepţie nu a început.
Send ready	operaţia send poate începe doar dacă operaţia corespunzătoare de recepţie a început.
Receive	Se completează la sosirea unui mesaj

Tabel 4.3. Moduri de comunicare MPI

Sintaxa rutinelor pentru transmisia mesajelor este:

MPI_Send(void*buffer, int count, MPI_datatype datatype, int dest, int tag, MPI_comm comm)

MPI_Recv(void* buffer, int count, MPI_datatype datatype, int source, int tag, MPI_comm comm, MPI_status status)

După ce mesajul a fost transmis, variabila Status poate fi utilizată pentru a colecta informaţii referitoare la operaţia MPI_recv, aceasta este o structură de date cu trei câmpuri: sursa mesajului, tagul şi cod de eroare, iar lungimea mesajului transmis poate fi verificată cu ajutorul funcţiei MPI_get_count.

Transmisie blocantă standard (Standard send). Forma *Standard send* se încheie odată ce mesajul a fost trimis, mesaj care poate să ajungă sau nu la destinaţie, iar ignorarea acestui fapt poate duce la comportări inconsistente ale programului. Sintaxa:

int MPI_SEND (void *buffer, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

unde:

buffer este adresa datei ce va fi trimise;
count este numărul de elemente de tipul *datatype* conţinute de *buffer*
dest reprezintă procesul destinaţie specificată prin intermediul rangului din cadrul comunicatorului *comm*;
tag este un marcator la dispoziţia programatorului pentru a distinge între diferite tipuri de mesaje;

Transmisie sincronă (Synchronous send). *Synchronous send* se foloseşte atunci când se doreşte confirmarea primirii mesajului, astfel destinaţia trimite spre sursă o confirmare iar transmisia este considerată încheiată numai după primirea confirmării. Primitiva send se consideră încheiată atunci când o operaţie receive a fost lansată, astfel bufferul aplicaţiei este disponibil pentru a recepţiona mesajul. Este posibilă o transmisie sincronă neblocantă ce nu presupune transmiterea mesajului, ci doar inexistenţa bufferării suplimentare la receptor, dar un buffer sistem la emiţător poate fi necesar, iar pentru a elimina copieri multiple ale mesajului se poate utiliza emisia blocantă. Această formă de transfer se foloseşte atunci când este nevoie de siguranţă a transmisiei, pentru a oferi comportament determinist. Sintaxa:

int MPI_SSEND (void * buffer, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

Transmisie buferată (Buffered send). Forma *Buffered send* foloseşte un buffer pentru trimiterea mesajului, dacă acesta nu poate fi trimis la momentul dorit. Avantajul faţă de metoda clasică îl constituie predictibilitatea recepţiei mesajului. Ataşarea unui buffer se realizează cu ajutorul funcţiei:

int MPI_BUFFER_ATTACH (void *buffer, int size)

Numai un singur buffer poate fi ataşat unui proces la un moment dat. Detaşarea se realizează cu:

int MPI_BUFFER_DETACH (void *buffer, int size)

Observaţii:

- după detaşare, utilizatorul poate reutiliza sau dealoca spaţiul ocupat de tampon
- unele operaţii, de ataşare şi detaşare, au un argument de tip void*; ele sunt folosite diferit – un pointer la tampon este transmis la *attach*; adresa pointerului este utilizată la *detach*, astfel că acest apel întoarce valoarea pointerului. Argumentele sunt definite ambele ca void* (şi nu void* respectiv void **) pentru a evita conversii forţate de tip (cast).

Transmisie ready (Ready send). Formele *Ready send*, ca şi *buffered send* se termină imediat, iar comunicarea este garantată a se efectua cu succes dacă o operaţie *receive* corespunzătoare a fost apelată. La expedierea mesajului, procesul expeditor pune mesajul în comunicator, sperând ca destinatarul să îl aştepte şi să îl primească. Dacă acesta nu îl acceptă, mesajul poate fi abandonat sau se generează o eroare.

int MPI_RSEND (void *buffer, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

Recepţie blocantă (Blocking receive). Sintaxa:

int MPI_RECV (void *buffer, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status status)

unde:

- *source* este rangul expeditorului în grupul asociat comunicatorului *comm*; ca şi sursă se poate specifica MPI_ANY_SOURCE, ceea ce este echivalent cu orice sursă;
- *tag* specifică tipul mesajelor ce vor fi acceptate, se poate de asemenea specifica MPI_ANY_TAG ca fiind identificator universal de mesaj.

Informaţia este returnată de către funcţia MPI_RECV în variabila *status*. Acest argument poate fi testat direct pentru a afla sursa mesajului, dimensiunea şi identificatorul de mesaj (de exemplu poate fi necesară identificarea sursei unui mesaj recepţionat folosind MPI_ANY_SOURCE).

4.2.4. Comunicaţii colective,comunicatori şi topologii virtuale

Mecanismele abstracte folosite în comunicaţie sunt : comunicatori, context de comunicare, topologii virtuale.În multe aplicaţii paralele este necesar ca operaţiile de comunicaţie să fie restricţionate la un set limitat de procese. MPI defineşte o serie de concepte şi introduce diverse mecanisme pentru lucrul cu procese structurate în grupuri şi organizate folosind comunicatorii. MPI introduce noţiunea de *comunicator* pentru a defini cadrul unei comunicaţii colective, în particular procesele care sunt implicate în această comunicare, dar şi pentru a permite construcţia modulară a programelor paralele în mod: secvenţial, paralel sau-şi concurent.

Un comunicator defineşte un *context de comunicare alături de o colecţie ordonată de procese* implicate într-un subset al calculului de efectuat. Prin introducerea comunicatorilor, comunicaţia între subseturi de procese ale programului poate fi structurată. Comunicaţia între procese din grupuri distincte este posibilă prin definirea unui *inter-comunicator*.

Comunicaţia inter-grupuri este, de obicei, gestionată de un conducător al grupului. Grupurile pot fi create şi prin intermediul reuniunii, intersecţiei, diferenţei între grupuri deja existe. Noţiunea de comunicator este fundamentală pentru definirea într-un mod modular a bibliotecilor de funcţii care pot fi invocate de procesele utilizator. Putându-şi defini propriul comunicator (alocat de către sistem, cu asigurarea unicităţii), procesele sistem pot schimba mesaje fără a interfera cu comunicaţia dintre procesele aplicaţiei utilizatorului.

Este necesară distincţia clară între diferitele tipuri de comunicatori. Astfel un *comunicator* este un obiect cu atribute şi reguli pentru creare, utilizare şi distrugere. Comunicatorul specifică un domeniu de comunicare ce poate fi utilizat de comunicarea punct la punct sau colectivă

- Un *intracomunicator* este utilizat pentru comunicarea în interiorul unui grup (comunicare intra-grup); el are două atribute: *grupul de procese şi topologia grupului.*
- Un *intercomunicator* este utilizat pentru comunicarea punct la punct între două grupuri diferite (comunicare inter-grup); atributele lui sunt cele două grupuri, fără a specifica topologia.
- Un *domeniu de comunicare* este un set de comunicatori. Dacă acest domeniu este pentru comunicare intra-grup atunci toţi comunicatorii sunt intracomunicatori şi au aceleaşi atribute.

Unui comunicator i se pot ataşa informaţii adiţionale alături de cele de grup şi context, informaţie numită **topologie.** Topologia reprezintă un mecanism ce realizează asocierea diverselor *scheme de adresare proceselor* aflate într-un grup. Grupurile şi comunicatorii sunt obiecte opace, detaliile reprezentării lor interne sunt specifice implementării MPI, iar accesarea lor este realizată folosind un handle. Contextele nu sunt utilizate explicit în funcţiile MPI, ci sunt asociate în mod *implicit* grupurilor de procese la crearea lor.

Procesele unui grup au ranguri de la 0 la n-1 ,n fiind numărul de procese din grup. În multe aplicaţii, ordonarea liniară a proceselor, după rang, nu reflectă tiparul de comunicare între procese.

Adesea, procesele trebuie aranjate în topologii cu două sau mai multe dimensiuni. În cazul general, tiparul de comunicare între procese corespunde unui graf. Un astfel de aranjament al proceselor, care reflectă comunicările punct la punct dintre ele reprezintă topologia virtuală a grupului de procese. Trebuie făcută distincţie între topologia virtuală a proceselor şi topologia reală a sistemului pe care acestea sunt executate. Topologia virtuală poate fi exploatată de sistem în plasarea proceselor pe procesoare, pentru a ameliora performanţele.

Trebuie reţinut faptul că topologia virtuală reflectă caracteristicile aplicaţiei şi poate fi folosită în *îmbunătăţirea performantelor.* Sunt cunoscute tehnici standard de mapare a unor topologii grilă / tor pe hipercuburi sau grile de procesoare. Pentru topologiile graf metodele sunt mai complexe şi se bazează adesea pe euristici. În afara informaţiei furnizate algoritmului de plasare a proceselor pe procesoare, topologiile virtuale au ca rol facilitarea scrierii programelor, într-o formă mai uşor de înţeles. Topologia virtuală poate fi considerată ca un atribut deosebit de puternic al unui comunicator, atribut ce permite descrierea structurii de interconectare a proceselor dintr-un grup, ea este folosită pentru a mapa procesele pe arhitectura fizică.

Topologia poate fi definită prin grupul de procese, numărul de conexiuni al fiecărui proces şi partenerul pentru fiecare conexiune Topologia virtuală este distinctă de graful programului, pentru că două procese pot comunica chiar dacă în topologie nu sunt conectate. Topologia virtuală descrie acei comunicatori care trebuie consideraţi de maparea grupului de procese pe arhitectura fizică. Noţiunea de topologie virtuala este deosebit de utilă, atunci când nu este disponibil un mecanism eficient de mapare a topologiei virtuale pe un calculatomasiv paralel. Astfel, dacă procesele utilizatorului vor comunica în principal cu cei mai apropiaţi vecini după modelul unui grid bidimensional, se va putea crea o topologie virtuală care să reflecte acest fapt. In acest mod este realizat accesul la rutinele convenabile, de exemplu, calculând rangul oricărui proces dat prin coordonatele sale în grid, luând în considerare condiţiile limită şi returnând MPI_NULL_PROC dacă se iese din grid. În particular există rutine care calculează rangurile celor mai apropiaţi vecini, rang ce poate fi folosit ca argument pentru primitivele de transmisie /recepţie MPI_SEND, MPI_RECV, MPI_SENDRECV etc. Deşi o topologie virtuală scoate în evidenţă şabloanele utilizate în comunicare într-un comunicator pentru o "conexiune", orice proces din comunicator poate comunica cu oricare altul.

O topologie virtuală poate fi modelată ca un graf în care nodurile reprezintă procese, iar arcele perechile de procese care comunică. Nu trebuie ca o comunicare să fie precedată de o deschidere explicită de canal. Ca urmare, absenţa unui arc între două noduri din graf nu înseamnă că procesele respective nu vor putea comunica între ele, aci că o astfel de comunicare nu contează la maparea topologiei virtuale de procese pe o topologie reală de procesoare. Arcele grafului de comunicare nu sunt ponderate. Deşi specificarea topologiei în termeni de graf este foarte generală, în multe aplicaţii specificarea unei topologii mai simple, regulate, este mai convenabilă. De exemplu inel, grile bi- sau tri-dimensionale, tor. Aceste topologii sunt complet definite prin numărul de dimensiuni şi prin numărul de procese pe fiecare axă de coordonate. Maparea unor topologii grilă sau tor este mai simplă decât maparea unui graf, motivând astfel interesul MPI pentru tratarea separată, explicită a acestora.

MPI permite o serie largă de topologii virtuale între care cele mai uzuale sunt topologiile de tip graf virtual, topologiile carteziene, s. a. Două noduri în graf sunt conectate dacă pot comunica. Grafurile de procese pot fi utilizate pentru a specifica orice topologie, cele mai utilizate fiind gridurile. O topologie este cartezianã dacă fiecare proces este conectat cu vecinii săi printr-un grid virtual.Mecanismul utilizat de MPI pentru a asigna ranguri proceselor dintr-un anumit domeniu de comunicaţie nu utilizează informaţie despre reţeaua de comunicaţie, ci oferă un set de funcţii ce permit programatorului *aranjarea proceselor în diverse topologii fără a specifica explicit maparea lor la procesoare*, responsabilitatea unei mapări capabile sa reducă costurile transmisiei de mesaje este în totalitate a bibliotecii MPI.

Comunicaţia colectivă permite difuzarea datelor către toate procesele unui grup, sincronizarea la bariere ca mecanism de control al execuţiei, distribuţia colectarea datelor de la procesele unui grup, operaţii globale de reducţie pe grupuri de procese (suma, minim, maxim) sau comunicaţie de tip all-to-all. Toate aceste operaţii se pot împărţi în două mari clase: *operaţii de transfer de date şi operaţii de calcul global.*

Deşi operaţiile de comunicaţie colectivă nu acţionează asemeni unei bariere ele realizează o *sincronizare virtuală*, iar programul necesită o semantică corectă astfel încât să se manifeste determinist chiar dacă este necesară includerea unor operaţii de sincronizare globală înainte respectiv după apelul funcţiei colective. Deoarece operaţiile colective sunt virtual sincrone, ele *nu necesită marcajul de mesaj*(tag-ul). Pentru cele mai multe operaţii colective există două variante astfel: transfer de date de aceeaşi dimensiune, respectiv transfer de date de dimensiuni diferite, funcție ce ce tip și volum de date necesită transfer.

Rutinele de transfer de date referă *difuzarea datelor (*broadcast), *distribuirea* (scatter) şi *colectarea rezultatelor* (gather) spre sau de la procesele unui grup. Colectarea de rezultate se poate realiza de către *toate* procesele (de ex. funcția MPI all-gather) sau de către *un singur* proces. De asemenea, există operaţii *all-to-all*, în care fiecare proces trimite date, respectiv recepţionează date de la celelalte procese, prin utilizarea unei singure funcții cu semnătură complexă.

Operaţiile de calcul global pot fi împărţite în operaţiile de reducere şi operaţii de scanare. Pentru operaţiile de reducere funcția de reducere se aplică pe toate datele corespunzătoare fiecărui proces din grup, iar pentru operaţiile de scanare, funcția de reducere se aplică pe datele proceselor cu rang mai mic decât cel al procesului care execută operația.

Funcţia de reducere trebuie să fie asociativă şi comutativă, iar rezultatul unei operaţii globale poate fi cunoscut de *toate procesele sau doar de un singur proces*. În operaţiile globale pot fi folosite atât operaţiile predefinite în MPI cât şi operaţii definite de utilizator. În cadrul comunicaţiei colective, nu se folosesc etichete de mesaj, coordonarea comunicaţiei proceselor este implicită şi toate rutinele de comunicaţie colectivă se blochează până când comunicaţia este terminată local.

Operaţii colective. Comunicarea colectivă implică un grup de procese între membrii căruia se desfăşoară comunicaţia. Biblioteca MPI suportă comunicaţii colective cum ar fi difuzarea, *dispersarea/adunarea (scatter/gather), schimb total de date, agregare şi bariera*. Pentru toate formele de comunicare colectivă, toate procesele unui grup trebuie să realizeze apelul corespunzător, cu argumente compatibile. Orice omisiune în acest sens constituie o eroare.

Caracteristici ale comunicaţiilor colective:

- comunicaţiile colective nu pot interfera cu comunicaţiile punct la punct şi vice-versa;
- comunicaţie *colectivă poate sau nu să realizeze sincronizarea* proceselor participante;
- comunicaţiile colective sunt blocante;
- *toate procesele participante într-un comunicator trebuie să apeleze comunicaţiile colective*;
- mesajul transmis este un vector cu elemente de un anumit tip
- tipul de date trebuie să fie acelaşi pentru transmisie şi recepţie

Sincronizarea la barieră este cea mai simplă operaţie de comunicare colectivă care nu implică transfer de date. Forma sa este: **MPI_BARRIER (MPI_comm comm)**

Unicul argument este comunicatorul *comm* ce defineşte grupul de procese sincronizate, astfel procesul apelant se blochează până când toţi membrii grupului execută apelul funcţiei (ating bariera).

Transmisia broadcast. In transmisia *one –to-all*, procesul root transmite acelaşi mesaj stocat în bufferul *buffer* (un număr de *count* de intrări de date având tipul *datatype)* tuturor proceselor inclusiv lui însuşi, procese incluse în comunicatorul comm. Toate procesele implicate în comunicaţie trebuie să specifice aceeaşi rădăcină (root, procesul cu identificatorul 0 de regulă).

int MPI_BCAST (void *buffer, int count, MPI_datatype datatype, int source, MPI_comm comm)

- *buffer* este adresa de început a buffer-ului, *count* este numărul intrărilor în buffer
- *datatype* este tipul datelor din buffer, source (poate fi cazul particular root) este rangul procesului sursă, iar *comm* este comunicatorul.

Conţinutul mesajului este identificat de tripletul *buffer, count, datatype,* pentru procesul sursă acest triplet specifică bufferele de emisie şi recepţie, iar pentru celelalte procese buferul de recepţie. Volumul de date transferat de procesul sursă trebuie să fie identic cu volumul de date recepţionat de fiecare proces.

Funcţia Gather. Procesul ţintă (poate fi chiar rădăcina pentru topologii de tip arbore) recepţionează mesaje personalizate de la fiecare din cele n procese (inclusiv de la el însuşi). Cele n mesaje sunt *concatenate în ordinea rangurilor proceselor* şi stocate în bufferul de recepţie al procesului receptor. Fiecare buffer emiţător este identificat de tripletul *sendbuf, sendcount, sendtype,* iar bufferul de recepţie este ignorat pentru procesele nonroot, iar pentru procesul root este identificat de tripletul *recvbuf, recvcount, recvtype*.

Funcţia implementează comunicaţia de tip *all-to-one,* toate procesele transferă datele aflate în *inbuf* procesului rădăcină, care plasează datele în locaţii contigue nesuprapuse în buferul *outbuf.*

int MPI_GATHER (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int target, MPI_comm comm)

unde:

- *sendbuf* este adresa de start a send buffer
- *sendcount* este numărul elementelor din send buffer
- *sendtype* este tipul datelor din send buffer
- *recvbuf* este adresa lui receive buffer
- *recvcount* este numărul elementelor pentru fiecare recepţie
- *target* este rangul procesului receptor

Funcţia Scatter descrie operaţia inversă operaţiei GATHER. Sintaxa sa este:

int MPI_SCATTER (void *sendbuf, int sendcount, MPI_Datatype sendtype, void recvbuf, int recvcount, MPI_Datatype recvtype, source, MPI_comm comm).

Pentru lista completă a funcţiilor de comunicaţie colectivă se vor consulta resursele [1][3].

Pentru *analiza de performanţă* şi instrumentarea codului aplicaţiei cu scopul de a identifica timpul necesar procesării este utilă funcţia *MPI_wtime()* ce returnează valoarea timpului raportat la ceasul sistem.Deasemenea este util Analizorul de concurenţă pentru Visual Studio care poate fi descărcat ca şi componentă distinctă de la adresa https://docs.microsoft.com/en-us/visualstudio/profiling/concurrency-visualizer?view=vs-2019 şi care va fi instalat pentru integrare în Visual Studio- IDE, cu scopul de a vizualiza variaţia gradului de paralelism pentru fiecare aplicaţie şi consumul de memorie la execuţia codului.

4.3.Exemple

4.3.1. Realizaţi o transformare a programului secvenţial pentru calculul lui PI bazat pe integrarea numerică a funcţiei f(x)=4/(1-x*x) în intervalul [0, 1]. Să se ofere o soluţie de paralelizare prin împărţirea intervalului de integrare aferent numărului de procesoare din sistem. Pentru algoritm se poate folosi aproximarea funcţiei cu sumarea k=1.. n (4/(1+((k-. 5)/n)**2) şi fiecare nod va primi numărul de dreptunghiuri folosite în aproximare, va calcula aria dreptunghiului şi apoi se sincronizează pentru calculul sumei globale. Implementaţi un program pentru calculul valorii lui PI ce poate rula pe un sistem cu un număr arbitrar de procesoare, astfel va trebui ca indiferent de numărul de procesoare, valoarea calculată să fie aceeaşi (sursa [5]).

```
   /* Calculates the PI. It is the area of f(x)=4/(1+x^2) in interval from 0 to 1.The
interval in splitted into n intervals which are distributed equaly to all  available
processes. The result consists in a sum of areas calculated by each processor !!! */
```

```
#include <stdio.h>
#include <math.h>
#include "mpi.h"

/* Function definition */
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=0.0, endwtime;
    int  namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    /* Initialize MPI */
    MPI_Init(&argc,&argv);

    /* Get no. of processes */
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);

    /* Get current process ID */
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    /* Get current processor name */
    MPI_Get_processor_name(processor_name,&namelen);

    /* Print info about MPI WORLD and about current process */
    printf("MPI_WORLD has %d processes\n", numprocs);
    printf("Process %d on %s\n",myid, processor_name);

    /* Initialize no. of intervals to 0 */
    n = 0;
    while (!done) {
        /* What root process does: */
   if (myid==0) {

            /* Read no. of intervals prom keyboard */
       printf("Enter the number of intervals: (0 quits) ");
            scanf("%d",&n);

              /* Next line should only be used if you are too lazy to enter no.
       of intervals */
       /* if (n==0) n=100; else n=0; */

       /* Get current time */
       startwtime = MPI_Wtime(); }

   /* Send no. of intervals to all processes. Only root sends the message
   and the others receive the message */
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

   /* In no intervals defined then stop program */
   if (n==0)
            done=1;
        /* If intervals defined */
   else {
            /* Calculate f(x) where x=1/n*(i-0.5)*/
       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);}

       /* Area is 1/n*f(x) */
            mypi=h*sum;
```

```
     /* Send calculated area to root through a SUM function. I don't
    know yet if root sends it's value, but it's for sure he receives
    as result the SUM of all numbers from the other processes including
    his own. */
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

     /* What root does */
    if (myid==0)
    {
            /* Print the result of PI*/
       printf("pi is approximately %.16f, Error is %.16f\n",
                    pi, fabs(pi - PI25DT));

       /* Get current time and print total time needed */
       endwtime = MPI_Wtime();
       printf("wall clock time = %f\n",
              endwtime-startwtime);}}}

/* Terminate MPI */
MPI_Finalize();
/* Success */
return 0;
```

4.3.2.Se vor testa exemplele ce ilustrează modul de utilizare a API-ului MPI, exemple pe care le puteți descărca de la adresa https:/ftp.utcluj.ro /~civan/CPD/1_LABORATOR/04_MPI

4.4.Întrebări teoretice

4.4.1.Ce tipuri de comunicație oferă interfața MPI? Explicați-le succint semnificația și aplicabilitatea.

4.4.2.Identificați minim 3 tipuri de operații de tip SEND și ilustrați modul de utilizare.

4.4.3.Care sunt funcțiile MPI pentru inițializarea/închiderea contextului de execuție

4.4.4.Ce rol joacă un comunicator?Dar un grup de procese?

4.4.5.Ce proces joacă un rol distinct în structurarea și execuția codului? Explicați prin exemplificare cu o secvență de cod ,posibila execuție paralelă pe o mașină cu memorie distribuită ce dispune de mai multe procesoare.

4.5.Probleme propuse

4.5.1.Să se implementeze o aplicaţie simplă folosind transmiterea de mesaje în accepţiunea în care se definesc un proces receptor şi n-1 procese emiţător. Acestea îşi transmit identificatorul de proces şi numele hostului procesului receptor care le tipăreşte.

4.5.2. Să se implementeze un program în care sunt definite un număr impar de procese. Procesul zero iniţializează un tablou de întregi şi distribuie tabloul tuturor proceselor folosind primitiva Scatter. Procesele receptor îşi primesc părţile corespunzătoare din tabloul sursă. Fiecare proces răspunde cu un mesaj ce conţine numele hostului pe care procesul se execută şi partea sa din tabloul distribuit anterior, iar procesul rădăcină recepţionează şi tipăreşte mesajele.

4.5.3.Să se calculeze produsul a doi vectori (x, y de dimensiune n) distribuiţi unui grup de procese. Tablourile ce conţin vectorii sunt alocate static, iar dimensiunea vectorilor este divizibilă cu p numărul de procesoare din sistem. Propuneţi două soluţii de implementare pentru operaţia de reducţie şi analizaţi modul de execuţie şi performanţa folosind analizorul de concurenţă ce poate fi descărcat de la adresa http://msdn.microsoft.com/en-us/library/dd537632.aspx şi integrat în Visual Studio.

4.5.4.Implementaţi un program care să determine timpul necesar execuţiei operaţiei MPI_Barrier. Utilizaţi comunicatorul implicit MPI_COMM_WORLD. Tipăriţi dimensiunea comunicatorului

asigurându-vă ca atât emiţătorul cât şi receptorul sunt pregătiţi la începutul testului. Cum variază performanţa primitivei variind dimensiunea comunicatorului?

4.5.5.Propuneţi un program care sa realizeze recepţia corectă a mesajelor de la toate procesele emiţătoare, astfel încât toate cele 100 procese să transmită procesului 0, utilizând recepţia nonblocantă şi primitiva Wait_some.

4.5.6.Modificaţi problema conectării procesoarelor unei maşini paralele într-o topologie de tip inel înlocuind referirea proceselor vecine folosind rangurile rank-1 şi rank+1 folosind rutinele MPI specific topologice (topologii catreziene, liniare) utile pentru " lumi" de procese ce necesită comunicatori distincti.

4.6. Miniproiect

Sa se descrie în pseudocod şi să se implementeze şi evalueze performanța implementării pentru soluții paralele ale algoritmilor de înmulțire matrici optimizati pentru implementare paralelă folosind transferul de mesaje. (algoritmul Fox (https://www.cs.usfca.edu/~peter/ipp/) și algoritmul Cannon (https://bit.ly/2DdmHgr). Pentru dezvoltarea aplicașiei este acceptată orice implementare a bibliotecii din cele existente (MS-MPI, OpenMPI, MPICH2).

Deasemenea, se vor reprezenta grafic evaluările de performanță (Accelerarea S și Eficiența E) realizând un set de execuții consecutive și modificând corespunzător numărul de procese și/sau numărul de procesoare. Se vor analiza rezultatele obținute și se va evalua scalabilitatea.

4.7. Referințe bibliografice

1. API MPI https://www.mpich.org/static/docs/v3.2/www3/index.htm (similar MS-MPI)
2. Standard MPI https://www.mpi-forum.org/docs/
3. Tutorial MPI https://computing.llnl.gov/tutorials/mpi/
4. Implementări diverse :

   Microsoft MPI v6 https://www.microsoft.com/en-us/download/details.aspx?id=47259

   Compilare si executie
   http://blogs.technet.com/b/windowshpc/archive/2015/02/02/how-to-compile-and-run-a-simple-ms-mpi-program.aspx
   MPI project template (VS2015)
   https://visualstudiogallery.msdn.microsoft.com/90fb60d4-0b8c-472f-8135-683d3c45f45a

   OpenMPI : http://www.open-mpi.org/

   Instructiuni de instalare
   http://programmers-journal.blogspot.ro/2013/09/netbeans-ide-configuration-for-openmpi.html

   MPICH2: http://www-unix.mcs.anl.gov/mpi/mpich2

   Instructiuni de instalare
   http://yazid.blog.umt.edu.my/files/2010/09/05-2-MPICH_VS2008manual.pdf

5. Programare paralelă- manual open source Universitatea Princeton :
   https://princetonuniversity.github.io/PUbootcamp/sessions/parallel-programming/Intro_PP_bootcamp_2018.pdf