ROMANIAN NATIONAL COMPETENCE CENTRE
IN HPC - HIGH PERFORMANCE COMPUTING

CURS ONLINE

INTRODUCERE ÎN MPI

WWW.RONCC.RO

Cuprins

Tabel de figuri

I. Introducere în HPC:

Supercomputing (sisteme cu eficiență înaltă de procesare) sau High Performance Computing – HPC (calcul de înaltă performanță) înseamnă practica unificării puterii de calcul pentru a se obține o putere cumulată mult mai mare decât cea oferită de computerele și serverele tradiționale. HPC reprezintă un sistem informatic capabil să prelucreze cantități mari de date într-un timp foarte scurt, rezolvând probleme numerice complexe în diferite domenii, folosind mai multe calculatoare și dispozitive de stocare pe o infrastructură cu o rețea cu bandă largă de transfer de date și latență redusă. HPC face posibilă explorarea și găsirea răspunsurilor la unele dintre cele mai mari probleme la nivel mondial în domeniul științei, ingineriei și afacerilor.

HPC constituie o parte esențială a cercetării academice și a inovației din domeniu. HPC îi ajută pe cercetători, arhitecți, ingineri, designeri să rezolve probleme complexe întru-un interval de timp redus și cu costuri mai mici decât cele pe care le presupune metoda de calcul tradițională.

Principalele avantaje ale HPC sunt:

- **Testare fizică redusă**: HPC este utilizat intens în cadrul simulărilor, eliminând realizarea testelor fizice. HPC facilitează, de exemplu, testele privind accidente auto, fiind mult mai ușor și mai puțin costisitor de realizat aceste simulări în comparație cu testele fizice de impact;
- **Viteză**: Utilizând cele mai noi tipuri de CPU/GPU-uri și structuri de rețea cu latență scăzută, cum ar fi accesul direct la memorie de la distanță (RDMA), beneficiind de capacitate de stocare rapidă, HPC poate realiza calcule uriașe mult mai rapid, în câteva zile, comparativ cu un interval de câteva luni de zile;
- **Costuri**: Răspunsurile mai rapide înseamnă costuri mai mici, precum și reducerea timpului de procesare. În plus, pentru întreprinderile mici și startup-uri, folosirea HPC în Cloud înseamnă reducerea costurilor privind achiziția de echipamente beneficiind astfel de avantaje precum scalabilitatea și flexibilitatea în folosirea resurselor în funcție de necesitățile curente;
- **Inovație**: HPC stimulează inovația în majoritatea domeniilor de cercetare oferind pârghiile necesare descoperirilor științifice revoluționare ce contribuie semnificativ la creșterea calității vieții.

Pentru a putea beneficia la maxim de avantajele HPC trebuie corelate infrastructura hardware cu infrastructura software de bază iar dezvoltarea software-ului va avea în vedere atât cerințele de business cât și specificul infrastructurii existente, de exemplu CUDA care este o arhitectură software și hardware folosită pentru programare masiv paralelă utilizând procesoare grafice.

II. Paralelism

Unul dintre factorii cheie în HPC este paralelismul. Pentru a rezolva o problemă mai rapid sau pentru a calcula seturi de date cât mai mari, sarcinile pot fi împărțite în mai multe sub-sarcini și executate în paralel. De fapt, cheia supercomputerelor nu se află în microprocesoare ultra-puternice sau în componente foarte scumpe și diferite de cele pe care le folosim zilnic în casele noastre, ci în factorul paralelism.

Programarea în calcul paralel se axează pe partiționarea întregii probleme de rezolvat în sarcini separate (tasks), alocarea sarcinilor procesoarelor disponibile și sincronizarea lor pentru a obține rezultate concludente. Acest tip de programare se poate aplica numai problemelor care sunt în general paralelizabile. O problemă poate fi partiționată sau descompusă după domenii, funcții sau după o combinație a celor două.

Procesarea în paralel respectiv în serie descrie dacă un sistem computerizat poate separa sarcinile de calcul pentru a utiliza mai multe procesoare sau nuclee simultan sau dacă se bazează pe finalizarea sarcinilor cu un singur nucleu de procesor. Inițial toate procesoarele pentru utilizatorii normali erau procesoare în serie, apoi Intel a introdus primul procesor dual-core pentru consumatori. Mai multe procesoare single-core pot lucra împreună pentru a gestiona procesarea serială prin clustere de computer paralele în rețea sau care rulează mai multe procesoare pe o singură placă de bază.

Un computer modern tipic rulează zeci până la sute de sarcini la un moment dat; cu toate acestea, fiecare nucleu lucrează doar la un singur proces la un anumit moment dat. Procesorul comută constant între diferitele „fire”(threads) de procesare sau „fluxuri de instrucțiuni” pentru a rula mai multe programe simultane sub o iluzie în timp real numită concurență. În cele din urmă, computerul pierde ciclurile procesorului în timp ce comută între lucrări și nu rulează la o eficiență optimă atunci când se rulează mai multe activități.

Un mediu de procesare paralelă poate procesa sarcinile mai rapid atunci când programele sunt proiectate să utilizeze procesarea paralelă. Programele seriale aliniază toate instrucțiunile în aranjament serial și interfață cu procesorul folosind un singur fir. Programele paralele funcționează prin împărțirea sarcinilor în părți individuale care pot fi împărțite între mai multe nuclee de procesor și reasamblate ca sarcini finalizate. Procesoarele paralele pot multiplica puterea de procesare a procesoarelor în serie însă, cu toate acestea, un procesor serial cu o viteză de ceas mai mare poate depăși procesoarele paralele atunci când lucrează cu un singur fir.

Procesare în serie

Programele scrise pentru procesarea în serie utilizează un singur nucleu(core) la un moment dat și procesează sarcini în ordine secvențială. Un procesor serial funcționează precum o duzină de benzi deschise de plată la un magazin alimentar, cu o casieră care rulează între diferite benzi, verificând pe toată lumea în același timp. Casierul sau CPU-ul, sare de pe o bandă pe alta verificând câteva articole deodată, înainte de a trece la următorul, cu scopul de a termina toate comenzile în același timp.

Prelucrare paralelă

Ideea din spatele procesoarelor paralele este că mai multe nuclee(cores) care funcționează împreună au ca rezultat performanțe mai bune. Un procesor paralel se comportă ca și cum ar avea mai mult de un casier care operează o duzină de benzi de plată. Dacă un program este configurat pentru a profita de procesarea paralelă, „clientul” ar putea împărți comanda în grupuri mai mici și ar putea folosi mai multe benzi de plată simultan.

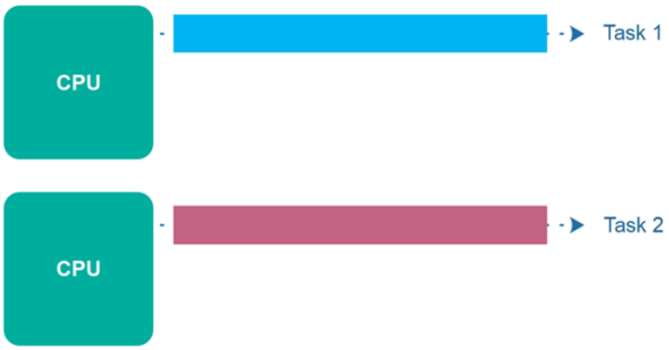


Figura 1. Prelucrarea paralelă

Avantajele calculului paralel constau în faptul că sistemele de calcul pot executa cod mai eficient, ceea ce poate conduce la economisirea de timp și bani prin sortarea „big data” mai rapid ca niciodată. Programarea paralelă poate rezolva și probleme mai complexe, folosind eficient mai multe resurse la un moment dat. Soluțiile de calcul paralel sunt capabile să scaleze mai eficient decât soluțiile secvențiale, deoarece pot gestiona mai multe instrucțiuni. În calculul paralel, procesoarele pot avea acces la o memorie partajată pentru a face schimb de informații între procesoare. O modalitate de a realiza paralelismul în calcul este folosirea mai multor procesoare pe un nod pentru a executa părți ale unui proces. De exemplu, puteți împărți o buclă în patru bucle mai mici și le puteți rula simultan pe procesoare separate. Acest proces se numește threading. Fiecare procesor procesează un thread.

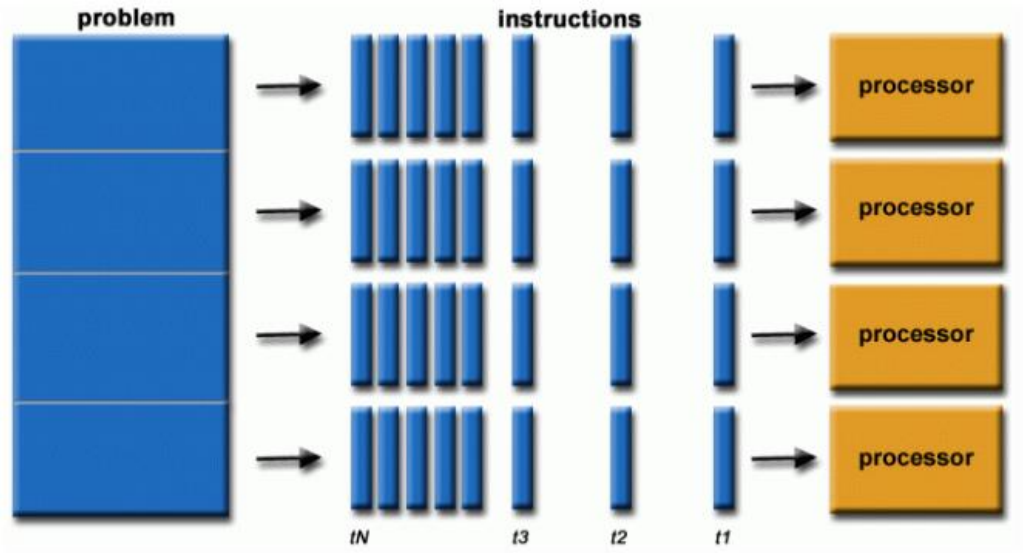


Figura 2. Threading

A. Proiectare algoritm paralel - PCAM

Nu există o schemă simplă care să poată fi aplicată pentru dezvoltarea programelor paralele. Totuși, se poate considera o abordare metodologică care să maximizeze numărul de opțiuni, să furnizeze mecanisme de evaluare a alternativelor și să reducă costurile necesare revenirii, în cazul luării unor decizii neperformante. O asemenea metodologie de dezvoltare permite programatorului să se concentreze în prima fază de dezvoltare asupra aspectelor independente de arhitectură, cum este concurența, iar analiza aspectelor dependente de hardware să fie amânată pentru finalul procesului de dezvoltare. O metodologie de dezvoltare a aplicațiilor paralele este propusă de către Ian Foster, în care procesul de dezvoltare este organizat în patru etape distincte: partiționarea, comunicația, aglomerarea și maparea (PCAM). Această metodologie definește aceste etape ca linii directoare în procesul de realizare a programelor paralele și nu ca niște etape stricte de dezvoltare.

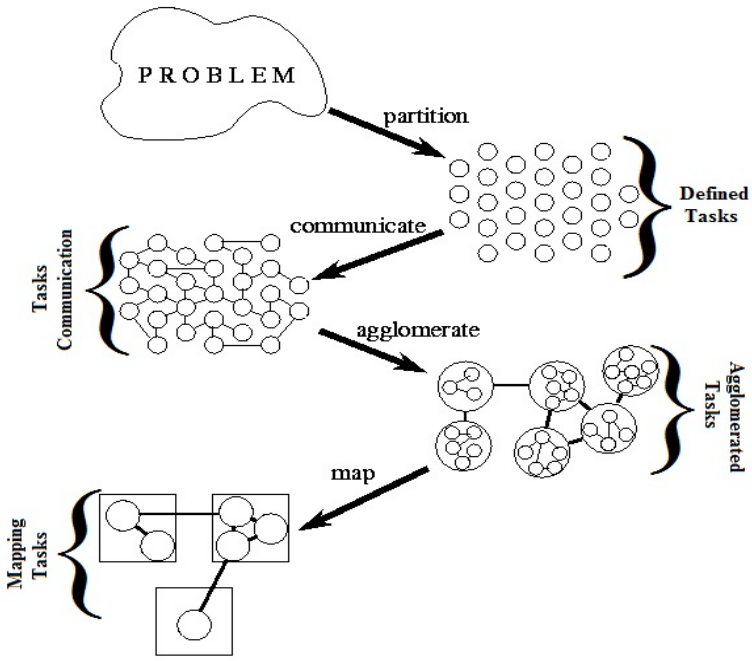


Figura 3. Algoritmul PCAM

Partiționarea

Problema partiționării are în vedere împărțirea problemei de programare în componente de calcul care se pot executa concurent. Aceasta nu implică o divizare directă a programului într-un număr de componente egal cu numărul de procesoare disponibile. Cele mai importante scopuri ale partiționării sunt legate de scalabilitate, abilitatea de a ascunde întârzierea (latency) datorată rețelei și realizarea unei granularități cât mai mari. Sunt de preferat partiționările care furnizează mai multe componente decât procesoare, astfel încât să se permită ascunderea întârzierii.

Întârzierea(Delay) este definită ca timpul necesar unui mesaj pentru a traversa o arhitectură și a ajunge la destinație. O componentă va fi blocată, sau va aștepta, până când mesajul care conține informația dorită va ajunge. Dacă există și alte componente de calcul disponibile, procesorul poate continua calculul; acest concept se numește multiprogramare și corespunde execuției concurente pe același calculator.

Comunicația

Componentele de calcul rezultate în urma partiționării nu pot fi executate în general independent; ele necesită anumite comunicații. Specificarea și analiza acestor operații se face în această fază. Comunicația este strâns legată de partiționare, mai exact modul de partiționare implică structura comunicației.

Există diferite tipuri de comunicații: locale și globale, structurate și nestructurate, statice și dinamice, sincrone și asincrone.

Comunicațiile sunt locale dacă fiecare proces comunică cu un număr mic de alte procese, numite vecini și avem comunicații globale dacă toate procesele sunt implicate în operația de comunicație. Într-o comunicație structurată, un proces și vecinii săi formează o structură regulată, spre deosebire de cazul comunicațiilor nestructurate când nu se poate stabili o structură între vecini.

O comunicație statică implică faptul că vecinii unui proces nu se modifică în timp, iar în cazul unei comunicații dinamice, vecinii unui proces se cunosc doar la execuție și se pot modifica în timpul execuției.

Comunicarea asincronă este orice tip de comunicație în care un proces comunică informația și apoi există un decalaj de timp înainte ca destinatarul să preia informația.

În cazul comunicației asincrone, procesul care comunică informația continuă execuția după ce aceasta a fost transmisă la destinație, în timp ce procesul care recepționează mesajul poate suferi o întârziere din cauza așteptării. Comunicarea asincronă nu are loc în timp real.

De exemplu, colegul de serviciu este ocupat și nu poate înțelege corect informațiile pe care le furnizați atunci când îi vizitați biroul. În schimb, vă cere să treceți la o anumită formă de comunicare asincronă - și anume Slack sau e-mail - astfel încât să poată primi, prelua și răspunde la informațiile în timpul liber.

Comunicația sincronă cere ca atât procesul care trimite, cât și cel care recepționează mesajul să fie disponibile până în momentul când transmisia s-a terminat, ambele procese putând apoi să-și continue lucrul. În felul acesta, nu este necesară stocarea mesajelor (lucru care poate fi necesar în cazul transmiterii asincrone).

În esență, în comunicarea sincronă, emițătorul și receptorul sunt sincronizați în timp real.

Aglomerarea

Algoritmul, rezultat în urma celor două etape precedente, este abstract, în sensul că nu este specializat pentru o execuție eficientă pe un anumit sistem paralel. Prin cea de-a treia etapă – aglomerarea – se trece de la abstract la concret prin luarea unor decizii care să conducă la un algoritm eficient pe un sistem dat. Operația presupune combinarea componentelor determinate în faza de partiționare cu scopul de reducere a numărului de comunicații și de asemenea de obținerea a unei granularități optime. Deciziile de combinare vor fi luate în funcție de rețeaua de comunicație a

sistemului ales și de granularitatea sistemului, astfel încât să se ajungă la o implementare eficientă, analizându-se de asemenea, dacă este cazul, replicarea anumitor calcule cu scopul reducerii comunicațiilor.

Maparea

După ce o problemă este partiționată, componentele trebuie să fie asignate, sau altfel spus mapate pe procesoare, pentru execuție. Un important aspect al acestei activități este menținerea localizării, mai precis plasarea componentelor problemei care schimbă informații, cât mai aproape posibil, pentru a reduce costul comunicațiilor.

I. Arhitectura supercomputerelor

Noțiunea de supercomputer a apărut pentru prima dată în anii 1960, când Seymour Cray a început designul celui mai rapid computer din lume în cadrul firmei Control Data Corporation. În 1964, Cray a introdus CDC 6600, care conținea inovații precum trecerea tranzistoarelor de germaniu în favoarea siliciului și un sistem de răcire bazat pe Freon. Mai important, a funcționat la o viteză de 40 MHz, executând aproximativ trei milioane de operații în virgulă mobilă pe secundă (3 MFLOPS), ceea ce l-a făcut cel mai rapid computer din lume. În 1975 a fost lansat Cray-1, având un procesor de 80 MHz și o unitate SIMD încorporată cu precizie pe 64 de biți. Noul sistem reprezintă un salt uriaș de la 3 MFLOPS de putere ale CDC 6600 la 160 MFLOPS ale Cray-1.

La începutul anilor ‘80 au apărut computerele masiv paralele, alimentate de mii de procesoare care lucrează în tandem pentru a depăși barierele de performanță. Așadar, un supercomputer grafic LINKS-1 de la Universitatea Osaka alcătuit din 257 microprocesoare Zilog Z8001 și 257 FPU Intel iAPX 86/20, obținea performanțe bune pentru acea vreme și putea reda grafică 3D realistă, cu 1.7 GFLOPS.

În 1999 IBM a decis să-și folosească procesoarele PowerPC pentru crearea Blue Gene, un proiect încheiat în noiembrie 2004. Primul BlueGene, cunoscut sub numele de BlueGene / L, a fost format din 131072 de procesoare, o cifră astronomică care i-a permis să ajungă la 70.72 TFLOPS de putere.

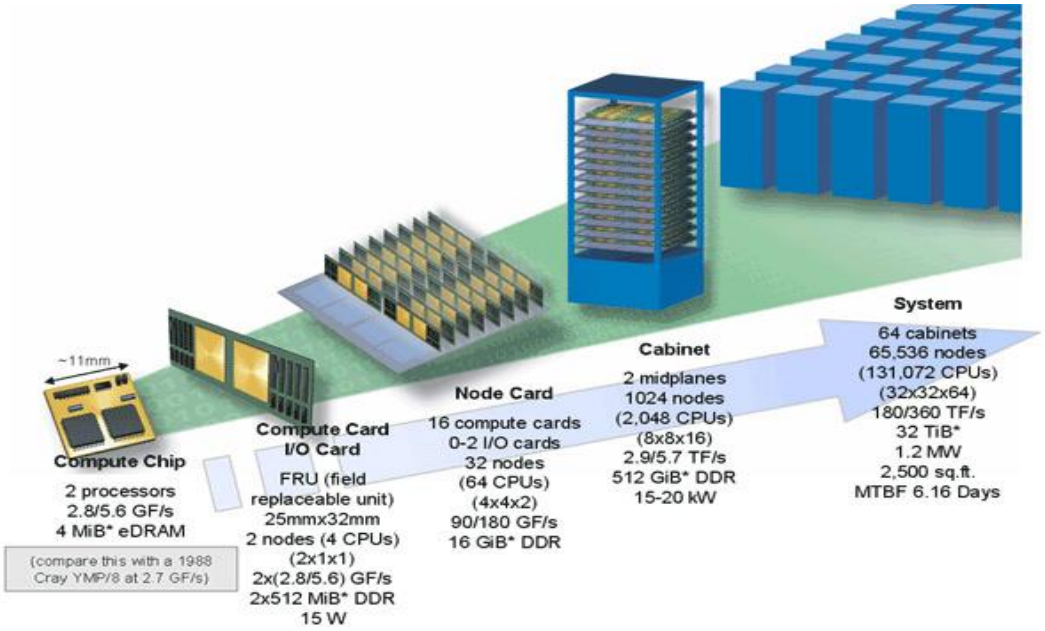


Figura 4. IBM Blue Gene Supercomputer

În zilele noastre, majoritatea supercomputerelor sunt proiectate având CPU-uri și GPU-uri. Tokyo Institute of Technology a creat TSUBAME și a folosit prima generație de NVIDIA Tesla pentru a ajunge la 170 TFLOPS, învingând astfel Blue Gene și Earth Simulator. Procesoarele grafice începuseră să aibă capacitatea de a rula algoritmi din ce în ce mai complecși de la implementarea unităților shader, iar odată cu arhitectura NVIDIA G80 acestea au fost folosite pentru a accelera algoritmi științifici de calcul. Cu toate acestea, un supercomputer bazat pe GPU nu a obținut primul loc până în 2013, moment în care Cray a lansat sistemul Titan, ce folosea procesoare AMD Opteron împreună cu GPU-uri Nvidia Tesla, având o putere de calcul de 10 PetaFLOPS, un salt de peste 50 de ori în ceea ce privește puterea de calcul în doar 5 ani, demonstrând eficiența enormă a GPU-urilor.

Pe baza algoritmului din standardul de testare cunoscut sub numele de High Performance Linpack (HPL) a fost concepută lista TOP500 unde se regăsesc cele mai rapide 500 de supercomputere din lume, precum și o descriere a acestora. Această listă se actualizează bi-anual și poate fi consultată la adresa https://www.top500.org/.

A. Arhitectură simetrică de multiprocesare

În prezent, HPC se bazează pe multiprocesarea simetrică sau multiprocesarea cu memorie partajată (SMP) ce implică o arhitectură hardware cu mai multe procesoare conectate la o singură memorie principală partajată, cu acces deplin la toate dispozitivele de intrare și ieșire și care sunt controlate printr-o singură instanță de sistemul de operare. Majoritatea sistemelor multiprocesor de astăzi folosesc o arhitectură SMP. În cazul procesoarelor cu mai multe nuclee, arhitectura SMP se aplică nucleelor, tratându-le ca procesoare separate.

Conceptul de SMP are la bază un multiprocesor cu memorie partajată în care costul accesării unei locații de memorie este același pentru toate procesoarele, adică are costuri uniforme de acces atunci când accesul este de fapt la memorie. Dacă locația este în cache, accesul va fi mai rapid, dar timpii de acces la cache și timpii de acces la memorie sunt aceeași pe toate procesoarele.

Sistemele SMP sunt sisteme multiprocesor strâns cuplate, cu procesoare omogene care rulează independent unul de celălalt. Fiecare procesor, executând diferite programe și lucrând pe diferite seturi de date, are capacitatea de a partaja resurse comune (memorie, dispozitiv I/O) care sunt conectate folosind o magistrală de sistem. Sistemele SMP au memorie partajată, centralizată, numită memorie principală (MM) care funcționează sub un singur sistem de operare cu două sau mai multe procesoare omogene.

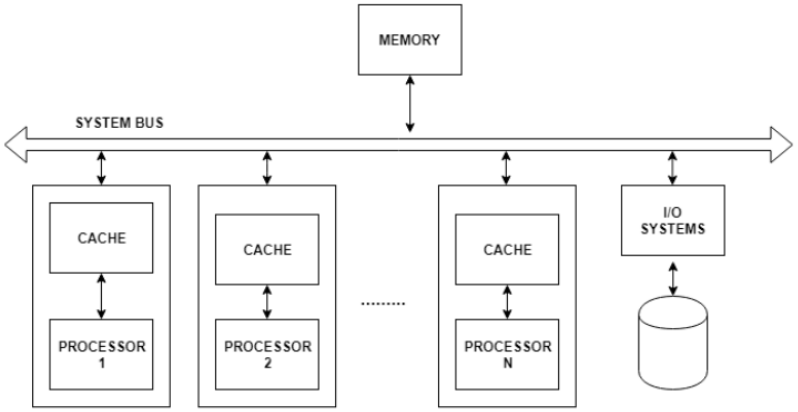


Figura 5. SMP - diagramă de sistem multiprocesor simetric

Așa cum se poate vedea în figura de mai sus, toate procesoarele din arhitectura multiprocesor simetrică conțin o magistrală comună și o memorie principală. De aceea, multiprocesarea simetrică este cunoscută sub numele de multiprocesare strâns cuplată. Fiecare dintre procesoarele din multiprocesare simetrică sunt egale și pot executa procese diferite după cum este necesar, indiferent de locul în care aceste procese sunt stocate în memorie. Aceasta este o diferență majoră față de multiprocesarea asimetrică.

Toate procesoarele conțin un cache individual în plus față de o memorie principală partajată. Acest lucru permite procesoarelor să acceseze datele mult mai rapid dacă acestea sunt disponibile în cache. De asemenea, reduce sarcina asupra magistralei de sistem, deoarece majoritatea solicitărilor sunt satisfăcute de memoria cache.

Multiprocesarea simetrică este folosită în:

- Sistemele de tip time-sharing, deoarece acestea au mai multe procese care rulează în paralel. Astfel, aceste procese pot fi programate pe procesoare paralele folosind multiprocesare simetrică.
- Procesarea simetrică nu este atât de utilă pentru computerele personale decât dacă se ia în considerare programarea cu mai multe „fire” (multithreading). Aceste „fire” multiple pot fi programate pe procesoarele paralele.
- Sistemele de partajare a timpului care utilizează programarea multithreading pot folosi, de asemenea, multiprogramarea simetrică.

B. Arhitectura de tip Cluster

Unul dintre cele mai cunoscute exemple de soluții HPC este supercomputerul, care conține mii de noduri de calcul care lucrează împreună pentru a finaliza una sau mai multe sarcini. Aceasta se numește procesare paralelă, similară cu a avea mii de PC-uri conectate împreună, putând face față unor serii de provocări privind procesarea cantităților masive de date și efectuarea de calcule complexe.

Arhitectura unui cluster este destul de simplă. Niște servere (noduri) care servesc diferite roluri într-un cluster sunt conectate printr-un fel de rețea. Este indicat însă nu obligatoriu ca nodurile de procesare să fie cât mai asemănătoare posibil. Figura următoare este o ilustrare simplă a arhitecturii de bază.

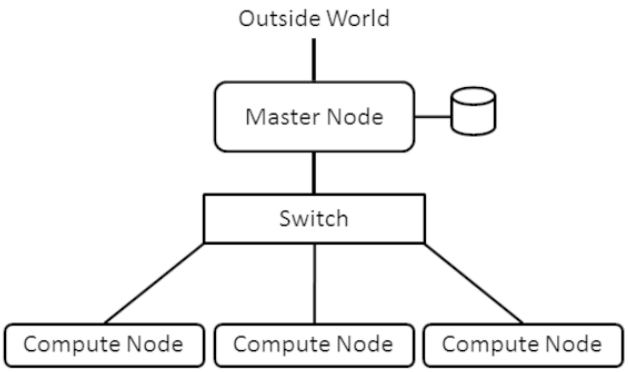


Figura 6. Arhitectura generală de tip cluster

Aproape întotdeauna există un nod care servește rolul unui „nod principal” (master node). Nodul principal este nodul „controller” sau nodul „management” pentru cluster. Controlează și efectuează întreținerea clusterului și de multe ori este nodul de conectare pentru ca utilizatorii să ruleze aplicații. În cazurile în care clustere sunt formate din mai puține noduri, nodul principal poate fi folosit atât pentru procesare, cât și pentru management, însă pe măsură ce numărul de noduri din cluster crește, nodul principal devine specializat și nu va fi folosit pentru procesare. Celelalte noduri din cluster ocupă rolul de noduri de procesare, acestea nu efectuează nicio funcție de gestionare a clusterului; ele doar procesează. Nodurile de procesare au sisteme de operare cu o amprentă cât mai mică – ceea ce înseamnă că demonii care nu sunt necesari sunt opriți și pachetele inutile nu sunt instalate.

Pe măsură ce clusterul crește, apar de obicei și alte roluri, necesitând adăugarea de noduri:

- De exemplu, servere de stocare pot fi adăugate la cluster. Aceste noduri nu rulează aplicații, ele stochează și furnizează date către restul clusterului;
- Nodurile suplimentare pot oferi capabilități de vizualizare a datelor în cadrul clusterului (de obicei, vizualizare la distanță);
- Clusterele foarte mari ar putea avea nevoie de noduri dedicate monitorizării clusterului sau conectării utilizatorilor la cluster.

Topologia de rețea este importantă deoarece poate avea un efect asupra performanței sistemului HPC iar pentru a evita blocaje de rețea (bottleneck-uri) este recomandată conectarea clusterului la o rețea privată folosind un spațiu de adrese nerutabil, separând în mod logic clusterul de o rețea publică.

Totuși clusterul trebuie conectat la o rețea publică, iar modalitatea de a face acest lucru atunci când clusterul se află într-o rețea privată este adăugarea unei interfațe de rețea la nodul principal. Acesta interfață va avea o adresă IP publică/routată care va permite conectarea la cluster din exterior. Doar nodul master ar trebui să aibă adresa IP publică/routată, deoarece nu există niciun motiv pentru ca nodurile de calcul să fie accesate direct.

O altă caracteristică cheie a unei arhitecturi de tip cluster este reprezentată de existența unui spațiu de stocare partajat între noduri. Strict vorbind, acest lucru nu este necesar, dar, fără el, unele aplicații MPI nu ar rula. SAN (Storage Area Network) permite transferul de date între servere și dispozitive de stocare cu ajutorul canalelor și comutatoarelor din fibră. NAS (Network Attached Storage) este o tehnologie de stocare la nivel de fișiere care oferă o facilitate de partajare a fișierelor cu ajutorul rețelei locale ce implică o rețea partajată în loc de o rețea dedicată, spre deosebire de SAN.

C. Architecturi State-of-the-Art

Infrastructurile HPC devin din ce în ce mai complexe iar până acum, CPU-urile au reprezentat componenta cea mai importantă în cadrul sistemelor HPC. CPU-urile se ocupă de toate calculele din sistem fie că implică gestionarea datelor de pe hard disk, afișarea rezultatelor pe ecran sau rularea aplicațiilor din RAM. CPU-urile au devenit extrem de puternice, având din ce în ce mai multe nuclee (core-uri) ajungând până la 128 în prezent. Concomitent cu creșterea puterii de procesare individuală a CPU-urilor, clusterele au permis cumularea puterii de procesare individuală a nodurilor, iar pentru a beneficia de paralelism în procesare s-a folosit cu precădere programarea având la bază MPI și OpenMP .

Ultimii ani au cunoscut o schimbare majoră către GPU-uri (Graphics Processing Unit). Dezvoltate inițial în principal pentru aplicații grafice și video, GPU-urile sunt acum suficient de puternice pentru a face mai mult decât mutarea imaginilor pe ecran, fiind din ce în ce mai mult valorificate folosind limbaje de programare precum CUDA și OpenCL pentru a oferi viteze mari de performanță pentru aplicațiile de uz general.

Conceptul de GPGPU (General Purpose Graphics Processing Unit) se referă la tendința din ce în ce mai comună și modernă de a folosi GPU-uri pentru calcule nespecializate, pe lângă scopul lor tradițional de calcul pentru grafica computerizată. Încorporarea GPU-urilor în scopuri generale îmbunătățește arhitectura CPU prin accelerarea porțiunilor unei aplicații, în timp ce restul continuă să ruleze pe CPU, oferind astfel un nivel de paralelism de supercalculare, creând în cele din urmă o aplicație generală mai rapidă și de înaltă performanță prin combinarea puterii de procesare a procesorului și a GPU-ului.

Producătorul de tehnologii NVIDIA a revoluționat GPGPU și a accelerat calculul încă din 2007 când a creat o arhitectura software și hardware pentru calculul paralel - Compute Unified Device Architecture (CUDA).

II. Basic MPI

A. Introducere în MPI

MPI (Message Passing Interface) este un „standard prin consens”, conceput inițial în cadrul unui forum deschis care a inclus furnizori de hardware, cercetători, cadre universitare, dezvoltatori de librării software și utilizatori, reprezentând peste 40 de organizații. Această participare largă la dezvoltarea sa a asigurat apariția rapidă a MPI ca standard utilizat pe scară largă pentru scrierea programelor de transmitere a mesajelor. MPI nu este un standard adevărat; adică nu a fost emis de o organizație de standardizare precum ANSI sau ISO.

Specificaţia MPI (Message Passing Interface) cunoaşte diverse implementări (API-uri MPI) sub forma unor librării 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 și/sau distribuite.

MPI (Message Passing Interface) are la baza modelul proceselor comunicate prin mesaje, fiind o specificație de librărie pentru arhitecturi de calcul paralel, care permite comunicarea informațiilor între diferite noduri de procesare dintr-un cluster. În prezent, MPI este cel mai comun protocol utilizat în calculul de înaltă performanță și este caracterizat prin portabilitate, scalabilitate și performanță ridicată, putând fi rulat pe aproape orice arhitectură distribuită, iar fiecare operațiune este optimizată corespunzător hardware-ului specific pe care rulează.

„Standardul” MPI a fost introdus de Forumul MPI în mai 1994 și actualizat în iunie 1995. Documentul care îl definește este intitulat „MPI: A Message-Passing Standard”, publicat de Universitatea din Tennesee și disponibil pe Forumul MPI. MPI-2.0 a fost finalizat în 1997, iar MPI-3.0 a fost finalizat în 2012. MPI-3.1 a fost finalizat în 2015, care oferă mici completări și remedieri la MPI-3.0. Având în vedere baza mare de aplicații paralele care se bazează pe MPI și istoria lungă a interfeței stabile a MPI, este probabil ca MPI să continue să aibă API-ul de bază neschimbat în anii următori. MPI 3 este încă cel mai utilizat standard. În 9 iunie 2021 a apărut MPI-4 ce este o actualizare majoră a standardului MPI.

MPI-2 produce extensii la standardul de transmitere a mesajelor MPI ce permite extinderea MPI în următoarele domenii:

- Comunicare unilaterală (obține, pune)
- Managementul dinamic al proceselor
- I/O paralel
- Comunicare colectivă extinsă
- Legături C++, F90
- Interfețe externe

MPI-3 poate prezenta probleme de compatibilitate inversă în unele cazuri însă majoritatea codului MPI va fi în continuare compatibil cu MPI-3, deoarece este compus în mare parte din extensii pentru MPI-2:

- Comunicare unilaterală: suport îmbunătățit pentru modelele de memorie partajată
- Comunicarea colectivă

- S-au adăugat funcții neblocante
- S-au adăugat neighborhood collectives pentru specificarea topologiei procesului
- A fost adăugat MPI_Count pentru definirea tipurilor de date derivate contigue mari
- Interfața instrumentului MPIT - permite inspecția variabilelor interne MPI
- S-au adăugat legături Fortran 2008
- Legăturile C++ au fost eliminate; au început să fie utilizate în schimb legături C din C++
- Unele funcții avansate care au fost depreciate în MPI-2 au fost eliminate sau înlocuite în MPI-3

Principalul update al versiunii MPI-4 este reprezentat de comunicarea partiționată însă, în prezent, există puține implementări publice disponibile. Totodată, a fost creat prototipul unei librării de extensii MPI portabile (cu numele funcției MPIX indicând funcții care nu sunt încă incluse în standardul MPI), iar această librărie de extensie implementează API-ul de comunicare partiționată peste orice librărie MPI existentă.

- Soluție pentru operațiuni „Big Count”.
- Colectivități persistente
- Comunicare partiționată
- Soluții de topologie
- Noi opțiuni de pornire prin sesiuni MPI
- Tratarea simplă a erorilor pentru a permite soluții de toleranță la erori
- Nouă interfață de instrumente pentru evenimente
- Reelaborarea capitolului termeni

B. Concepte de programare paralelă

În prezent există mai multe modele de programare paralelă folosite de către dezvoltatorii de software. Paralelismul este codat explicit de programator acesta fiind responsabil atât pentru analiza algoritmului / aplicației seriale de bază cât și pentru identificarea modalităților prin care programatorul poate descompune calculele și extrage concurența. Codurile sunt scrise folosind paradigmele asincrone sau slab sincrone. În paradigma asincronă toate sarcinile concomitente se execută asincron, astfel de programe pot fi mai greu de motivat și pot avea un comportament nedeterminist. În cadrul programelor sincronizate, sarcinile sau subseturile de sarcini se sincronizează pentru a efectua interacțiuni, iar între aceste interacțiuni, sarcinile se execută complet asincron. Principalele tipuri de arhitectură de aplicații paralele sunt *Single Program Multiple Data (SPMD) si Multiple Programs, Multiple Data (MPMD)*

SPMD este de fapt un model de programare „la nivel înalt” care poate fi construit pe orice combinație a modelelor de programare paralelă. **Single Program** înseamnă ca toate procesoarele rulează simultan o copie a aceluiași program, însă fiecare proces funcționează pe o copie separată a datelor iar **Multiple Data** înseamnă că procesoarele rulează task-uri ce pot folosi date diferite. Programele SPMD au, de obicei, logica necesară programată în ele pentru a permite diferitelor sarcini să se ramifice sau să execute condiționat doar acele părți ale programului pentru care sunt proiectate

să le execute, mai precis, sarcinile nu trebuie neapărat să execute întregul program – pot doar o parte din acesta.

Figura 7. Modelul SPMD

Modelul SPMD, care utilizează transmiterea mesajelor sau programarea hibridă, este probabil cel mai frecvent utilizat model de programare paralelă pentru clustere cu mai multe noduri. De fapt multe aplicații paralele pot fi dezvoltate folosind modelul popular SPMD deoarece acest model prezintă avantaje în proiectarea programelor pentru aplicații mici, inclusiv eficiența în construirea și rularea aplicațiilor și comoditatea în utilizarea tehnicilor de programare secvențială. Cu toate acestea, deoarece o singură sursă de program trebuie să includă toate sarcinile pentru toate procesoarele din aplicație, un program SPMD este greu de modificat. Prin urmare, atunci când aplicația devine mare și complexă, în special cu calcule eterogene care necesită modele de comunicare neregulate sau necunoscute, este preferat modelul MPMD (multiple program multiple data).

MPMD separă aplicația în module funcționale diferite cu surse de cod separate pentru sarcini concurente, promovând astfel reutilizarea codului și capacitatea de a compune programe. Este, de asemenea, potrivit pentru metacalcularea într-un mediu eterogen, la scară largă, deoarece programele MPMD sunt mult mai slab cuplate decât cele scrise folosind modelul SPMD.

Figura 8. Modelul MPMD

C. The Six Necessary MPI Commands

Schița de bază a unui program MPI urmează acești pași generali:

- Inițializare comunicație,
- Comunicare pentru a partaja date între procese,
- Ieșire într-un mod „curat” din sistemul de transmitere a mesajelor când s-a finalizat comunicarea

Din punct de vedere logic un programator debutant în MPI se poate descurca de obicei doar cu aceste șase funcții:

Inițializare comunicații

- `MPI_INIT` - inițializează mediul MPI
- `MPI_COMM_SIZE` - returnează numărul de procese
- `MPI_COMM_RANK` - returnează numărul (clasamentul) proceselor

Comunicație pentru a partaja date între procese

- `MPI_SEND` - trimite un mesaj
- `MPI_RECV` - primește un mesaj

Ieșire din sistemul de transmitere a mesajelor

- `MPI_FINALIZE`

MPI trebuie să fie întotdeauna inițializat și finalizat, aceste două comenzi sunt întotdeauna primul și ultimul apel din program. Comenzile corespunzătoare sunt MPI_Init și MPI_Finalize. MPI_Init ia întotdeauna ia ca referință argumentele liniei de comandă, în timp ce MPI_Finalize nu.

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

Numărul dintr-un comunicator se numește dimensiunea comunicatorului (**size).** În același timp, fiecare proces din interiorul unui comunicator are un număr unic pentru a-l identifica. Acest număr se numește rangul procesului (**rank).** Modul de definire a Size și Rank se realizează prin următoarele apeluri:

```
int size, rank;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Funcțiile de comunicare MPI_Send() și MPI_Recv() sunt exemple de operații de comunicare point-to-point. Pentru a transmite un mesaj, rangul i (nodul sursă) va apela funcția MPI_Send(), în timp ce rangul j (nodul destinație) va prelua datele apelând funcția MPI_Recv().

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

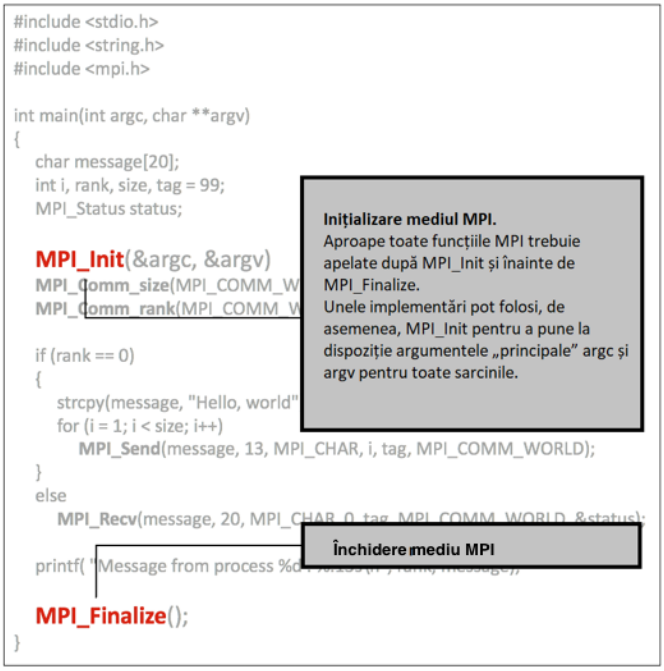
- *smessage*: specifică un buffer care conține datele care urmează să fie transmise;
- *count*: numărul de elemente care urmează sa fie transmise;
- *datatype*: specifică tipul de date din buffer; toate datele au același tip;
- *dest*: specifică rangul procesului care urmează să primească datele;
- *tag*: o eticheta care permite destinatarului să distingă între diferitele mesaje de la aceeași sursă;
- *comm*: specifică comunicatorul folosit în comunicare;

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

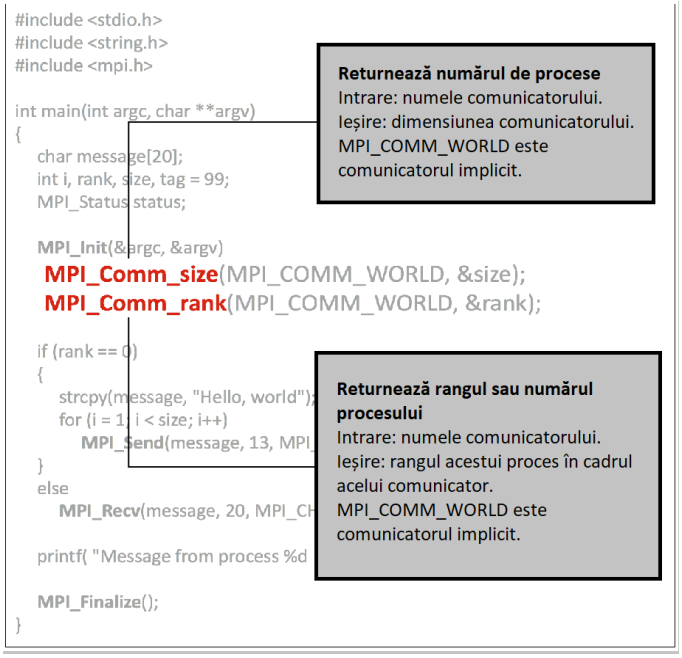
- *rmessage*: specifica bufferul în care se primesc datele;
- *count*: numărul maxim de elemente care urmează sa fie primite;
- *datatype*: tipul datelor care vor fi primite;
- *source*: specifică rangul procesului care trimite mesajul;
- *tag*: eticheta pe care trebuie s-o aibă mesajul pentru a fi primit;
- *comm*: specifică comunicator-ul folosit în comunicare;
- *status*: este o structură care specifică informații despre mesaj, după încheierea operației de comunicare (id_procesor send, cod de eroare);

```
#include <stdio.h>
#include <string.h>
#include <mpi.h>

int main(int argc, char **argv)
{
  char message[20];
  int  i, rank, size, tag = 99;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0)
  {
    strcpy(message, "Hello, world");
    for (i = 1; i < size; i++)
      MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
  }
  else
    MPI_Recv(message, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);

  printf( "Message from process %d : %.13s\n", rank, message);

  MPI_Finalize();
}
```

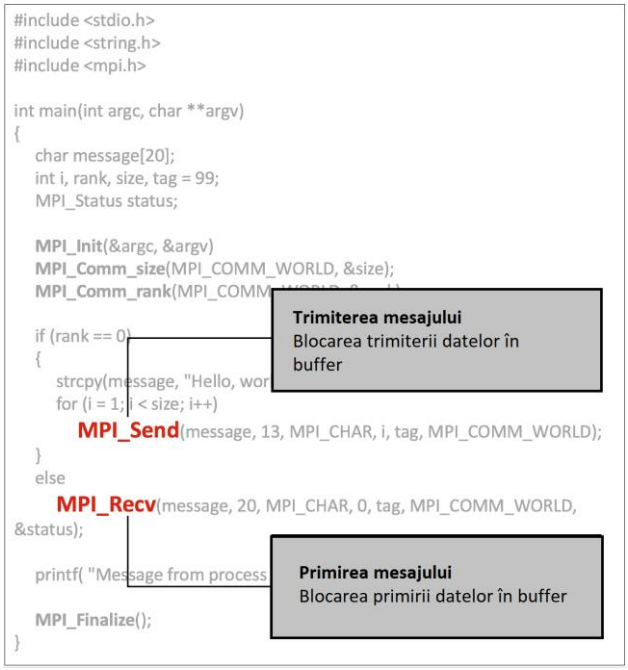
Exemplu pentru inițializarea și oprirea mediul MPI



Exemplu interogarea dimensiunii grupului de sarcini comunicator și găsirea numărul de rang al acestui proces



Exemplu - trimiterea si primirea mesajelor



III. Comunicația colectivă (MPI Collectives)

MPI Collectives respectiv comunicația colectivă se poate defini ca o comunicație ce implică unul sau mai multe grupuri de procese. Cu alte cuvinte, comunicarea de date folosind toate procesele dintr-un comunicator dat (comunicatorul implicit fiind MPI_COMM_WORLD).

Este necesar să evidențiem câteva aspecte importante atunci când folosim comunicația colectivă:

- Un apel colectiv este executat de către toate procesele din comunicator;
- Înlocuiește o secvență mai complexă de apeluri punct la punct;
- Poate folosi sau nu comunicația sincronizată;
- Comunicația colectivă nu interferează cu comunicația punct-la-punct și nici invers;
- Comunicația colectivă nu utilizează tag-uri;
- Buffer-urile de trimitere și primire, când se folosesc call-uri de comunicație colectivă, trebuie să se potrivească pentru a permite call-ului să funcționeze.

Operațiile colective fundamentale în MPI Collectives sunt de următoarele tipuri:

- Sincronizarea barierei: blochează apelul până când toate procesele sunt sincronizate într-un anumit punct (barieră);
- Mutarea datelor sau comunicația globală: Difuzare (Broadcast), împrăștiere (Scatter), adunarea (Gather), transmisia datelor în format All-to-All;

- Reducere globală: Un proces dintr-un comunicator colectează date din fiecare proces și execută operații speciale de reducere cu acele date în vederea obținerii unui rezultat.

A. Funcția de sincronizare (Barrier)

În aplicațiile paralele într-un mediu de memorie distribuită, uneori este necesară sincronizarea explicită sau implicită. Ca și alte librării de transmitere a mesajelor, MPI oferă o rutină MPI_BARRIER(), pentru sincronizarea tuturor proceselor dintr-un comunicator

Funcția MPI_BARRIER() blochează apelantul până când toți membri grupului îl apelează. Apelul se returnează la oricare dintre procese doar după ce toți membri grupului inițiază apelul.

```
int MPI_Barrier(MPI_Comm comm)
```

- *comm*: Comunicatorul – In general acesta este MPI_COMM_WORLD;

În exemplul următor, considerăm fișierul hellompi.cpp cu următorul conținut:

```
#include <stdio.h>
#include <string.h>
#include <mpi.h>

int main(int argc, char **argv)

{
    int rank, nprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Barrier(MPI_COMM_WORLD);
    printf("Hello, world.  I am %d of %d\n", rank, nprocs);fflush(stdout);
    MPI_Finalize();
    return 0;
}
```

Pentru a rula acest program folosind 8 procese, se execută comanda *mpirun -np 8 hellompi*. Rezultatul rulării programului este:

Compiling
Compilation is OK
Execution ...
Hello, world. I am 0 of 8

Hello, world. I am 1 of 8
Hello, world. I am 2 of 8
Hello, world. I am 3 of 8
Hello, world. I am 4 of 8
Hello, world. I am 5 of 8
Hello, world. I am 6 of 8
Hello, world. I am 7 of 8
Done.

B. Funcția Broadcast

Există multe situații unde un proces care trebuie să trimită/difuzeze date către toate procesele dintr-un grup (inclusiv el însuși). MPI furnizează primitivul de difuzare MPI_BCAST() pentru a îndeplini această sarcină.

Funcția MPI_BCAST() transmite un mesaj de la procesul cu rank-ul root către toate procesele din grup (inclusiv el însuși). Este apelat de către toți membri grupului folosind aceleași argumente pentru comunicator și pentru root. Funcția copiază buffer-ul din root în toate celelalte procese.

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int
root,MPI_Comm comm)
```

- *buffer*: Specifică bufferul unde sunt stocate datele pentru transmisie;
- *count*: Numărul de elemente din buffer
- *datatype*: Tipul de date din buffer;
- *root*: Rank-ul;
- *comm*: Comunicatorul;

În următorul exemplu se folosește funcția MPI_Bcast() pentru transmiterea numărului 1234 folosind mai multe procese. Considerăm în continuare că acest script este conținutul fișierului hellompi.cpp.

```
#include <stdio.h>
#include <string.h>
#include <mpi.h>

int main(int argc, char **argv)

{
    MPI_Init(&argc, &argv);
    // Get my rank in the communicator
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    // Determine the rank of the broadcast emitter process
    int broadcast_root = 0;
    int buffer;
```

```
    if(my_rank == broadcast_root)
    {
        buffer = 12345;
        printf("[MPI process %d] I am the broadcast root, and send value %d.\n", my_rank,
buffer);
    }
    MPI_Bcast(&buffer, 1, MPI_INT, broadcast_root, MPI_COMM_WORLD);
    if(my_rank != broadcast_root)
    {
        printf("[MPI process %d] I am a broadcast receiver, and obtained value %d.\n", my_rank,
buffer);
    }
    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

După executarea comenzii mpirun -np 8 hellompi rezultatul este:

Compiling
Compilation is OK
Execution ...
[MPI process 0] I am the broadcast root, and send value 12345.
[MPI process 1] I am a broadcast receiver, and obtained value 12345.
[MPI process 2] I am a broadcast receiver, and obtained value 12345.
[MPI process 3] I am a broadcast receiver, and obtained value 12345.
[MPI process 4] I am a broadcast receiver, and obtained value 12345.
[MPI process 5] I am a broadcast receiver, and obtained value 12345.
[MPI process 6] I am a broadcast receiver, and obtained value 12345.
[MPI process 7] I am a broadcast receiver, and obtained value 12345.
Done.

C. Funcția Gather

Dacă o structură de date este împrăștiată în toate procesele dintr-un grup și cineva dorește să colecteze fiecare element din structură într-o structură de date specificată pe un singur proces, apelul ce se utilizează este MPI_GATHER().

În funcția MPI_GATHER(), fiecare proces (inclusiv root) trimite conținutul bufferului către procesul root. Procesul root primește mesajele și le înregistrează în ordinea rankului.

Sunt permise tipuri de date derivate pentru sendtype și recvtype iar tipul de semnături ale sendcount și sendtype trebuie să fie egale cu tipul de semnături recvcount și recvtype ceea ce înseamnă că cantitatea de date trimisă trebuie să fie egală cu cantitatea de date primită.

```
int MPI_Gather(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm
comm)
```

- *sendbuf*: Bufferul ce conține datele de trimis;

- *sendcount*: Numărul de elemente din buffer
- sendtype: Tipul de date din bufferul de trimis;
- *recvbuf*: Bufferul unde se stochează datele strânse din procesul root;
- *recvcount*: Numărul de elemente dintr-un mesaj primit;
- recvtype: Tipul unui element din bufferul primit;
- *root*: Rank-ul procesului root;
- *comm*: Comunicatorul.

În următorul exemplu, se utilizează 3 procese la rularea comenzii *mpirun -np 3 hellompi*

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

int main(int argc, char **argv)

{
    int isend, irecv[3];
    int rank, size;

    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );

    isend = rank + 1;

    MPI_Gather(&isend, 1, MPI_INT, &irecv, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if(rank == 0)
       printf("irecv = %d %d %d\n", irecv[0], irecv[1], irecv[2]);

    MPI_Finalize();
    return 0;
}
```

Rezultatul rulării este:

Compiling
Compilation is OK
Execution ...
irecv = 1 2 3
Done.

Un alt exemplu mai practic este înmulțirea unei matrici *A* cu un vector coloană *B*. Următorul program calculează operația $\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \end{pmatrix} \cdot \begin{pmatrix} 3 \\ 3 \\ 3 \\ 3 \\ 3 \end{pmatrix}$. Rutina MPI_GATHER() preia cpart din fiecare proces și îl înregistrează în ctotal

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

int main(int argc, char **argv)

{
    int i,k;
    int a[2][5] = {{1,1}, {2,2}, {3,3}, {4,4}, {5,5}};
    int b[5] = {3,3,3,3,3};
    int cpart[2] = {0,0}, ctotal[5];
    int root = 0;
    MPI_Init( &argc, &argv );
    for(i=0; i<2; i++)
    {
       for(k=0; k<5; k++)
    {
        cpart[i] = cpart[i]+ a[i][k]*b[k];
    }
    }
    MPI_Gather(cpart, 2, MPI_INT, ctotal, 2, MPI_INT, root, MPI_COMM_WORLD);
    MPI_Finalize();
    for(i=0; i<2; i++){
        printf("%d \n", ctotal[i]);
    }
    return 0;
}
```

Rezultatul rulării:

Compiling
Compilation is OK
Execution ...
15
30
Done.

Funcția MPI_GATHERV() extinde funcționalitatea funcției MPI_GATHER() prin faptul că permite primirea unei cantități de date ce variază de la fiecare proces datorită argumentului recvcounts ce primește un vector de elemente. De asemenea, MPI_GATHERV permite o flexibilitate mult mai mare cu privire la locația unde datele sunt plasate datorită argumentului displs.

```
int MPI_Gatherv(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, const int recvcounts[], const int displs[], MPI_Datatype
recvtype, int root, MPI_Comm comm)
```

- *sendbuf*: Bufferul ce conține datele de trimis;
- *sendcount*: Numărul de elemente din buffer
- sendtype: Tipul de date din bufferul de trimis;
- *recvbuf*: Bufferul unde se stochează datele strânse din procesul root;
- *recvcount*: Numărul de elemente dintr-un mesaj primit;
- displs: intrarea specifică deplasarea relativă la recvbuf unde se plasează datele primite din proces
- *recvtype:* Tipul unui element din bufferul primit;
- *root*: Rank-ul procesului root;
- *comm*: Comunicatorul.

Exemplul de mai jos ilustrează aplearea funcției MPI_GATHERV() folosind 4 procese la rularea comenzii mpirun -np 4 hellompi.

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

int main(int argc, char **argv)

{
    int buffer[6];
    int rank, size, i;
    int receive_counts[4] = { 0, 1, 2, 3 };
    int receive_displacements[4] = { 0, 0, 1, 3 };

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    for (i=0; i<rank; i++)
    {
        buffer[i] = rank;
    }
    MPI_Gatherv(buffer, rank, MPI_INT, buffer, receive_counts, receive_displacements, MPI_INT,
0, MPI_COMM_WORLD);
    if (rank == 0)
    {
        for (i=0; i<6; i++)
        {
            printf("[%d]", buffer[i]);
        }
        printf("\n");
```

```
        fflush(stdout);
    }
    MPI_Finalize();
    return 0;
}
```

Rezultatul rulării:

Compiling
Compilation is OK
Execution ...
[1][2][2][3][3][3]
Done.

D. Funcția Scatter

Funcția MPI_SCATTER() primește un vector de elemente și distribuie aceste elementele în ordinea rankului de proces. Mesajul este împărțit în n părți egale $0 \dots i \dots n-1$ iar partea i este trimis la procesul i din grup și fiecare proces primește acest mesaj.

```
int MPI_Scatter(const void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm
comm)
```

- *sendbuf*: Bufferul ce conține datele de trimis;
- *sendcount*: Numărul de elemente din buffer
- sendtype: Tipul de date din bufferul de trimis;
- *recvbuf*: Bufferul unde se stochează datele strânse din procesul root;
- *recvcount*: Numărul de elemente dintr-un mesaj primit;
- *recvtype:* Tipul unui element din bufferul primit;
- *root*: Rank-ul procesului root;
- *comm*: Comunicatorul.

Următorul exemplu folosește funcția MPI_SCATTER() cu 3 procese:

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

int main(int argc, char **argv)

{
    int isend[3], irecv;
    int rank, size, i;

    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
```

```
    if(rank == 0){
      for(i=0;i<size;i++) isend[i] = i+1;
    }

    MPI_Scatter(&isend, 1, MPI_INT, &irecv, 1, MPI_INT, 0, MPI_COMM_WORLD);

    printf("rank = %d: irecv = %d\n", rank,irecv);

    MPI_Finalize();
    return 0;
}
```

Rezultatul rulării programului folosind comanda mpirun -np 3 hellompi:

Compiling
Compilation is OK
Execution ...
rank = 0: irecv = 1
rank = 1: irecv = 2
rank = 2: irecv = 3
Done.

MPI_SCATTERV() este funcția inversă a MPI_GATHERV(). MPI_SCATTERV extinde funcția MPI_SCATTER(), permițând trimiterea unui număr variabil de date către fiecare proces deoarece argumentul sendcounts este un vector de elemente. De asemenea, primul item din fiecare bloc nu trebuie poziționat la o anumită distanță față de blocul precedent. Mai mult, blocurile nu necesită să fie stocate în ordinea corectă.

Funcția MPI_SCATTER necesită ca datele trimise să fie stocate în adrese de memorie adiacente de mărimi uniforme. Prin urmare, primele n date sunt trimise către primul proces din comunicator iar următoarele n către următorul proces și asa mai departe. În acest context, acest lucru poate fi prea restrictiv pentru unele aplicații. Este posibil ca unele procese să aibă nevoie de mai puțin de n date iar acest lucru poate duce la trimiterea de buffere cu date inutile (memorie tampon). Pentru astfel de situații se folosește funcția MPI_SCATTERV().

```
int MPI_Scatterv(const void* sendbuf, const int sendcounts[], const int
displs[], MPI_Datatype sendtype, void* recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```

- *sendbuf*: Bufferul ce conține datele de trimis;
- *sendcount*: Numărul de elemente din buffer
- *displ:* Intrarea specifică deplasarea relativă la recvbuf unde se plasează datele primite din proces
- sendtype: Tipul de date din bufferul de trimis;
- *recvbuf*: Bufferul unde se stochează datele strânse din procesul root;
- *recvcount*: Numărul de elemente dintr-un mesaj primit;
- *recvtype:* Tipul unui element din bufferul primit;
- *root*: Rank-ul procesului root;
- *comm*: Comunicatorul.

O alternativă mai sigură și mai performantă, în funcție de problemă, ar putea fi ca un al doilea scatterv să trimită regiuni suprapuse către un al doilea buffer de recepție. Sau în alte cazuri, citirile redundante ar putea fi gestionate prin comunicare punct la punct. Fiecare problemă este diferită iar obținerea soluției optime nu este întotdeauna ușoară și poate depinde chiar de mediul hardware pe care îl folosește.

În general, rutinele MPI sunt bine optimizate iar încercarea de a utiliza mai multe apeluri MPI în locul unui singur apel MPI trebuie analizat cu mare atenție.

E. Funcția Reduce

Reducția globală este una dintre cele mai folositoare operații colective. Rezultatul aplicării unei funcții peste toate procesele dintr-un grup este colectat într-un anumit proces specificat sau în mai multe procese. Operațiile de reducție ale acestor funcții pot fi operații predefinite (min, max, sum, etc) sau pot fi definite de utilizator. Există mai multe funcții de reducție ce permit returnarea rezultatelor într-un singur nod, în mai multe noduri sau funcții ce oferă o combinație de capabilități ale funcțiilor de reducție și împrăștiere.

Operațiile de reducție predefinite sunt operații logice, operații pe biți sau operații matematice:

MPI_Op	Descriere	Tipuri de date permise
MPI_MAX	Maximum	Int, float
MPI_MIN	Minimum	Int, float
MPI_SUM	Sumă	Int, float, complex
MPI_PROD	Produs	Int, float, complex
MPI_LAND	ȘI logic	Int, bool
MPI_BAND	ȘI binar	Int, bool
MPI_LOR	SAU logic	Int, bool
MPI_BOR	SAU binar	Int, bool
MPI_LXOR	SAU EXCLUSIV logic	Int, bool
MPI_BXOR	SAU EXCLUSIV binar	Int, bool
MPI_MAXLOC	Valoarea maximă și locația	*
MPI_MINLOC	Valoarea minimă și locația	*

Tabel 1. Operații Reduce

Operatorul MPI_MAXLOC este folosit pentru a calcula un maxim global și un index ce este atașat acestei valori. În mod similar, MPI_MINLOC calculează un minim global și un index. În

general se folosește pentru a calcula maximum/minimum global și rank-ul procesului ce conține această valoare.

MPI_REDUCE() combină elementele din bufferul de intrare al fiecărui proces din grup folosind o operație de reducție și returnează rezultatul în bufferul de ieșire al procesului cu rank-ul root.

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

- *sendbuf*: Adresa bufferului trimis;
- *recvbuf*: Adresa bufferului primit;
- *count*: Numărul de elemente dintr-un mesaj primit;
- *datatype:* Tipul de date a elementelor din bufferul trimis;
- *op:* Operația de reducție;
- *root*: Rank-ul procesului root;
- *comm*: Comunicatorul.

F. Funcția All-Gather

Funcția MPI_ALLGATHER() este similară cu MPI_GATHER(), cu excepția faptului că toate procesele primesc rezultatul (reamintim că în cazul funcției MPI_GATHER(), doar procesul root primește rezultatul). Drept urmare, parametrul root nu mai este prezent în MPI_ALLGATHER().

```
int MPI_Allgather(const void* sendbuf, int sendcount, MPI_Datatype
sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm
comm)
```

- *sendbuf*: Bufferul ce conține datele de trimis;
- *sendcount*: Numărul de elemente din buffer
- sendtype: Tipul de date din bufferul de trimis;
- *recvbuf*: Bufferul unde se stochează datele;
- *recvcount*: Numărul de elemente dintr-un mesaj primit;
- *recvtype:* Tipul unui element din bufferul primit;
- *comm*: Comunicatorul.

Rezultatul funcției MPI_ALLGATHER() este același ca atunci când s-ar apela funcția MPI_GATHER() de către toate procesele.

Funcția MPI_ALLGATHERV() este similară cu MPI_ALLGATHER(), diferența dintre ele este că în funcția MPI_ALLGATHERV(), fiecare proces poate trimite cantități diferite de date.

```
int MPI_Allgatherv(const void* sendbuf, int sendcount, MPI_Datatype
sendtype, void* recvbuf, const int recvcounts[], const int displs[],
MPI_Datatype recvtype, MPI_Comm comm)
```

- *sendbuf*: Bufferul ce conține datele de trimis;
- *sendcount*: Numărul de elemente din buffer
- sendtype: Tipul de date din bufferul de trimis;
- *recvbuf*: Bufferul unde se stochează datele;
- *recvcount*: Numărul de elemente dintr-un mesaj primit;
- *displs:* intrarea specifică deplasarea relativă la recvbuf unde se plasează datele primite din proces
- *recvtype:* Tipul unui element din bufferul primit;
- *comm*: Comunicatorul.

G. Funcția All-To-All

MPI_ALLTOALL() este o extensie a funcției MPI_ALLGATHER() unde fiecare proces trimite aceleași cantități de date între ele și primește aceleași cantități de date de la fiecare în parte. Operațiile acestei rutine inițiază $2 \cdot n$ comunicații punct-la-punct independente

```
int MPI_AllToAll(const void* sendbuf, int sendcount, MPI_Datatype
sendtype, void* recvbuf, const int recvcounts, MPI_Datatype recvtype,
MPI_Comm comm)
```

- *sendbuf*: Bufferul ce conține datele de trimis;
- *sendcount*: Numărul de elemente din buffer
- sendtype: Tipul de date din bufferul de trimis;
- *recvbuf*: Bufferul unde se stochează datele;
- *recvcount*: Numărul de elemente dintr-un mesaj primit;
- *recvtype:* Tipul unui element din bufferul primit;
- *comm*: Comunicatorul.

Parametru	**Descriere**
sendbuf	Bufferul ce conține datele de trimis
sendcount	Numărul de elemente din buffer
sendtype	Tipul de date din bufferul de trimis
recvbuf	Bufferul unde se stochează datele
recvcount	Numărul de elemente dintr-un mesaj primit
recvtype	Tipul unui element din bufferul primit
comm	comunicatorul

Similar, funcția MPI_ALLTOALLV() permite mai multă flexibilitate: datele de intrare și ieșire sunt definite ca vectori.

```
int MPI_AllToAllv(const void* sendbuf, const int sendcount[], const int
sdispls[], MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm)
```

- *sendbuf*: Bufferul ce conține datele de trimis;
- *sendcount*: Numărul de elemente din buffer
- *sdispls*: Intrarea i specifică deplasarea de la care se trimit datele către rank-ul i
- sendtype: Tipul de date din bufferul de trimis;
- *recvbuf*: Bufferul unde se stochează datele;
- *recvcount*: Numărul de elemente dintr-un mesaj primit;
- *rdispls*: Intrarea i specifică deplasarea la care ar trebui sa fie scrise datele din rank-ul i
- *recvtype:* Tipul unui element din bufferul primit;
- *comm*: Comunicatorul.

H. Funcția All-Reduce

MPI_ALLREDUCE() este similară cu funcția MPI_REDUCE(), cu excepția că rezultatul apare în bufferul de primire al tuturor membrilor grupului. Parametrul root ce este prezent în funcția MPI_REDUCE() nu se regăsește în MPI_ALLREDUCE().

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

- *sendbuf*: Bufferul ce conține datele de trimis;
- *recvbuf*: Bufferul unde se stochează datele;
- *count*: Numărul de elemente dintr-un mesaj primit;
- *datatype*: Tipul de date a elementelor din bufferul trimis;
- *op:* Operația de reducție;
- *comm*: Comunicatorul.

Bibliografie

https://www.open-mpi.org/doc/current/man1/mpirun.1.php

http://www.rc.usf.edu/tutorials/classes/tutorial/mpi/chapter8.html

https://cvw.cac.cornell.edu/mpicc/scatterv

https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

https://jenkov.com/tutorials/java-concurrency/concurrency-vs-parallelism.html