LECȚIA 1. SISTEME ȘI MODELE DE CALCUL ȘI PROGRAMARE PARALELĂ

Obiective

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

1.1 Clasificarea sistemelor de calcul paralel

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

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

Problema de calcul trebuie să etaleze uzual caracteristici de genul:

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

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

Clasificarea clasică după Flynn

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

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

Facem o succintă caracteristică a fiecărei dintre aceste clase.

Clasa SISD

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

Clasa SIMD

Pentru acest model, calculatorul paralel constă din *N* procesoare identice. Fiecare procesor are propria sa memorie locală în care poate stoca programe sau date. Toate procesoarele lucrează sub controlul unui singur șir de instrucțiuni emise de o singură unitate centrală de control. Altfel spus, putem presupune că toate procesoarele păstrează o copie a programului de executat (unic) în memoria locală. Există *N* șiruri de date, câte unul pentru fiecare procesor. Toate procesoarele operează sincronizat: la fiecare pas, toate execută aceeași instrucțiune, fiecare folosind alte date de intrare (recepționate din memoria externă). O instrucțiune poate fi una simplă (de exemplu, o operație logică) sau una compusă (de exemplu, interclasarea a două liste). Analog, datele pot fi simple sau complexe. Uneori este necesar ca numai o parte din procesoare să execute instrucțiunea primită. Acest lucru poate fi codificat în instrucțiune, spunând procesorului dacă la pasul respectiv va fi activ (și execută instrucțiunea) sau inactiv (și așteptă următoarea instrucțiune). Există un mecanism de genul unui „ceas global", care asigură sincronizarea execuției instrucțiunilor. Intervalul dintre două instrucțiuni succesive poate fi fix sau să depindă de instrucțiunea ce urmează să se execute. În majoritatea problemelor ce se rezolvă pe acest model este necesar ca procesoarele să poată comunica între ele (pe parcursul derulării algoritmului) pentru a transmite rezultate intermediare sau date. Acest lucru se poate face în două moduri: o memorie comună (Shared Memory) sau folosind o rețea de conexiuni (Interconnection Network).

Shared Memory SIMD

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

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

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

Clasa MISD

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

Clasa MIMD

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

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

1.2 Definiția algoritmilor paraleli

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

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

Această definiție necesită următoarele explicații:

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

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

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

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

Aceste elemente trebuie să verifice următoarele condiții:

i. *Dacă* $v \in V_0$, *rezultă că* $\tau(v) = 0$.
ii. *Dacă arcul* $(v, v') \in E$, *rezultă că* $\tau(v') \geq \tau(v) + 1$.
iii. *Pentru orie* $v, v' \in V \backslash V_0$ *și* $v \neq v', \tau(v) = \tau(v')$, *rezultă că* $\pi(v) \neq \pi(v')$.

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

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

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

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

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

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

MODELE DE PROGRAMARE PARALELĂ

Obiective

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

2.1 Sisteme de calcul cu memorie distribuită și partajată

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

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

Vom descrie mai jos particularitățile de bază ale acestor modele.

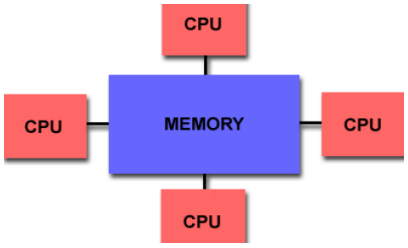
Calcul paralel este numită execuția în paralel pe mai multe procesoare a acelorași instrucțiuni sau și a unor instrucțiuni diferite, cu scopul rezolvării mai rapide a unei probleme, de obicei special adaptată sau subdivizată. Ideea de bază este aceea că problemele de rezolvat pot fi uneori împărțite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

Calculul distribuit (din engleză Distributed computing) este un domeniu al informaticii care se ocupă de sisteme distribuite. Un sistem distribuit este format din mai multe

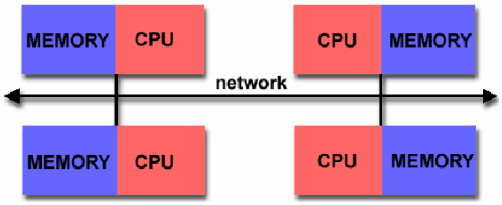
calculatoare autonome care comunică printr-o rețea. Calculatoarele interacționează între ele pentru atingerea unui scop comun. Un program care rulează într-un sistem distribuit se numește program distribuit, iar procesul de scriere a astfel de programe se numește programare distribuită.

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

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



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



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

Regiuni critice

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

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

Un exemplu de regiune critică este secvența de pseudocod:

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

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

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

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

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

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

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

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

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

Excluderea reciprocă

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

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

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

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

1) **excluderea reciprocă propriu-zisă**, care constă în faptul că la fiecare moment cel mult un proces se află în secțiunea sa critică;

2) **competiția constructivă, neantagonistă**, care se exprimă astfel: dacă niciun proces nu este în secțiunea critică și dacă există procese care doresc să intre în secțiunile lor critice, atunci unul dintre acestea va intra efectiv;
3) **conexiunea liberă între procese**, care se exprimă astfel: dacă un proces „întârzie" în secțiunea sa necritică, atunci această situație nu trebuie să împiedice alt proces să intre în secțiunea sa critică (dacă dorește acest lucru).

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

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

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