Noțiuni de programare paralelă şi algoritmi paraleli

1. Introducere

Algoritmii în care operațiile se execută una după alta se numesc seriali sau secvențiali. Cei pentru care mai multe operații se pot executa simultan se numesc algoritmi paraleli. Un algoritm paralel pentru un calculator paralel se poate defini drept un set de procese ce se pot executa simultan şi care comunică între ele în vederea rezolvării unei probleme date. Prin proces se înțelege o secțiune de programe rulată pe un procesor.

Proiectarea unui algoritm paralel se bazează pe determinarea eficienței de utilizare a resurselor disponibile. Unul din parametrii care caracterizează eficiența unui algoritm este timpul de rulare (sau de execuție), definit ca timpul scurs între activarea primului procesor (sau a primului set de procesoare) şi terminarea rulării pe ultimul procesor (sau ultimul set de procesoare).

Pentru proiectarea unui algoritm paralel se pot considera mai multe abordări. Una dintre ele constă în paralelizarea unui algoritm serial. Dacă există un anumit algoritm serial pentru rezolvarea unei probleme se cere determinarea paralelismului inerent care apare în mod natural în cadrul algoritmului, nu ca urmare a unui efort special, urmată de implementarea pe o maşină paralelă. Trebuie subliniat că exploatarea paralelismului inerent într-un algoritm secvențial nu conduce întotdeauna la un algoritm eficient. Uneori, o abordare mai bună constă în construirea unui algoritm care rezolvă o problemă dată într-o manieră similară, dar diferită în formă cu cea folosită de algoritmul secvențial. De multe ori este de preferat elaborarea unui algoritm paralel, total diferit de cel secvențial.

În orice caz, proiectarea unui algoritm paralel nu poate ignora câteva considerații importante. Unul dintre acestea este costul comunicației dintre procese. Importanța sa derivă din faptul că pentru un algoritm dat costul comunicațiilor ar putea fi mai mare decât costul prelucrării propriu-zise. Un alt considerent este cel legat de arhitectura calculatorului pe care se execută algoritmul, ceea ce face algoritmul eficient pe o maşină şi ineficient pe o alta.

În acest capitol se consideră două modele folosite pe scară largă în programarea paralelă: modelul bazat pe partajare de memorie comună şi modelul în care comunicația se realizează prin transfer de mesaje.

Modelul cu memorie comună se referă la programarea într-un mediu multiprocesor pentru care comunicația între procese se realizează prin intermediul memoriei comune (sau globale). Modelul cu transfer de mesaje (message passing) este adecvat programării pe multicalculatoare, în cadrul cărora comunicația se realizează prin mesaje.

Comunicația prin zone comune de memorie, în mediul oferit de multiprocesoare, nu este lipsită de probleme. Ele apar daca două procese modifică acelaşi element de date într-o ordine inacceptabilă în mod obişnuit. Multiprocesoarele oferă suport pentru instrucțiuni de sincronizare destinate evitării acestui fel de probleme; aceste instrumente de sincronizare vor fi prezentate în secțiunea ce urmează.
În contrast cu multiprocesoarele, actualizarea datelor într-un mediu multicalculator nu reprezintă o problemă. Memoria unui procesor nu este folosită în comun cu alte procesoare. Principala problemă este generată de transferul de mesaje, ceea ce impune implementarea unor mecanisme de transmitere şi recepție a mesajelor.

În cadrul acestui capitol se vor discuta probleme ale programării paralele si ale elaborării de algoritmi paraleli. Sunt descrise structuri de tip sincron, asincron, pipeline şi se prezintă unele elemente de măsurare a performanțelor. Capitolul se încheie cu câteva exemple ce ilustrează partea teoretică prezentată.

2. Modele de programare

Sunt abordate două tipuri de programare paralelă: cel pe multiprocesoare şi cel pe multicalculatoare.

2.1 Programare paralelă pe multiprocesoare

Scrierea de programe paralele într-un anumit limbaj într-un mediu multiprocesor presupune existenta posibilității de efectuare a unor anumite operații precum crearea de procese, sincronizarea acestora, efectuarea de operații atomice ş.a.m.d. Aceste operații sunt prezentate în cele ce urmează.

Crearea proceselor.
Unul din mecanismele de creare a unui proces este *fork* şi este folosit de limbaje de nivel înalt (Fortran, C/C++) sub sisteme de operare de tip UNIX. Când un proces executa apelul sistem *fork* se creează un nou proces numit *slave* (sau *child*) ce reprezintă o copie a procesului original *master* (sau *parent*). Procesul slave îşi începe execuția din punctul următor apelului fork. Funcția fork returnează procesului master identificatorul UNIX al procesului slave creat. Aceeaşi funcție fork returnează 0 procesului slave.

```
return_code = fork();
if (return_code = 0)
{
      slave(); -- procesul slave începe activitatea
               -- apelarea rutinei slave
      exit(0); -- procesul slave îşi încetează activitatea
}
else
      {
            if (return_code=-1)
                  print("eşuarea creării procesului slave");
            else
                  -- procesul master îşi continuă activitatea
      }
```

În codul de mai sus funcția fork returnează 0 în copia procesului slave a lui return_code şi identificatorul UNIX al procesului slave în copia procesului master a variabilei return_code. Este de notat că ambele procese au câte o copie a return_code în locații diferite de memorie.

Sincronizarea.
De regulă comunicarea în medii multiprocesor se realizează prin secțiuni comune de date. Acestea elimină necesitatea unor copii multiple ale unui element de date. Existența unui element unic de date folosit în comun de mai multe procesoare economiseşte memorie şi evită problemele de actualizare caracteristice unor copii multiple. Pot apare însă probleme dacă două procese efectuează acces simultan la acelaşi element de date pe care îl prelucrează şi îl actualizează folosind rezultatul prelucrării . De aceea accesul la aceste date se bazează pe *excluderea mutuală*.

Pentru exemplificare se consideră execuția unei instrucțiuni care incrementează variabila comuna index:

```
index = index + 1;
```

Instrucțiunea presupune o citire urmată de o înscriere. Între cele două operații are loc incrementarea. Dacă două procese p1 şi p2 trebuie să efectueze incrementarea, valoarea finală a variabilei index ar trebui să fie cu 2 mai mare decât cea inițială, lucru posibil când secvența citire - înscriere efectuată de primul proces nu se intercalează cu operația de citire sau cu operațiile citire – înscriere a celui de-al doilea proces. În cazul în care aceste operații se intercalează valoarea finală a variabilei index va fi cu 1 mai mare decât cea inițială, ceea ce este incorect. Acest exemplu argumentează necesitatea excluderii mutuale în accesarea variabilei comune, mecanism care va permite accesul doar a unui singur proces la un moment de timp dat.
Multiprocesoarele oferă mai multe instrucțiuni simple (numite şi primitive) pentru excluderea mutuală care permit sincronizarea resurselor si a proceselor. Adesea ele sunt implementate printr-o combinație hardware-software. Iată câteva exemple:

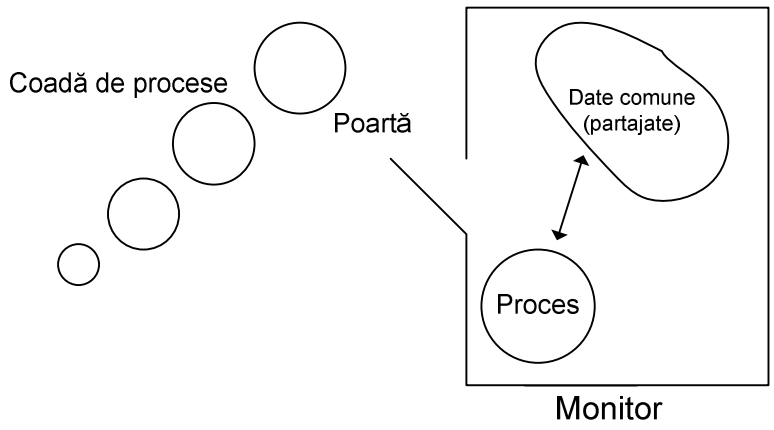
Lock / Unlock.

Un proces care încearcă efectuarea accesului la date comune partajate prin "poarta lock" aşteaptă până când poarta devine "unlocked" după care repune poarta în starea "lock" pentru a împiedica accesul altor procese la aceleaşi date. La terminarea folosirii datelor din zona blocată procesul deblochează poarta pentru a permite accesul altor procese la datele comune. Caracteristica importantă a operațiilor lock şi unlock este execuția lor neîntreruptibilă (operație atomică). Odată inițiate operațiile nu mai sunt întrerupte până la terminare.

```
Lock(L)
      {
         while (L==1) NOP;
         L=1;
      }
Unlock(L)
      {
          L=0;
      }
```

L reprezintă starea porții de protecție. În condițiile existenței instrucțiunilor Lock şi Unlock problema actualizării indexului se tratează precum urmează:

```
Lock(L);
index=index+1;
Unlock(L);
```



Aşa cum se ştie, o secvență neîntreruptibilă „protejează" o *secțiune critică* (sau *regiune critică*). În general o secțiune critică poate fi executată simultan de un număr de procese ce nu depăşeşte o valoare dată. În cazul excluderii mutuale această valoare este 1. O structură care permite accesul exclusiv la o secțiune critică se mai numeşte şi monitor. Monitoarele au fost introduse încă din 1974 în teoria programării paralele (de către Hoare) şi au devenit un mecanism important în cadrul acesteia. Monitorul poate fi conceput ca un fel de „gard" în jurul datelor comune cu o poartă prin care poate pătrunde un singur proces la un moment de

timp dat. Odată aflat în interiorul „curții” procesul poate avea acces la datele comune. În mod general utilizarea mecanismului este următoarea:

```
Lock(L);
      <regiune critică>
Unlock(L);
```

Implementare mecanismului Lock se poate face prin busy / waiting (formă ineficientă) sau suspendare într-o listă de aşteptare. Existența instrucțiunii atomice “test and set” simplifică implementarea.
Instrucțiunea Lock este implementată adesea în sisteme multiprocesor utilizând o instrucțiune specială de tip Test&Set. Această instrucțiune este o operație atomică ce returnează valoarea curentă a locației de memorie şi înscrie în aceasta o anumită valoare. Ambele faze ale instrucțiunii (test şi set) sunt implementate ca o operație neîntreruptibilă la nivel hardware. În momentul în care un procesor execută o astfel de instrucțiune, nici un alt procesor nu poate interveni în execuția acestuia. În general operația Test&Set se poate defini (presupunând că valoarea de înscris stabilită a fi este 1, valoarea opusă 0):

```
Test&Set(L)
      {
      temp=L;
      L=1;
      return temp;
      }
Reset(L)
      {
      L=0;
      }
```

Implementarea instrucțiunilor Lock şi Unlock utilizând instrucțiunea Test&Set arată în felul următor:

```
Lock(L)
      {
      while (Test&Set(L)==1) NOP;
      }
Unlock(L)
      {
      Reset(L)
      }
```

În practică, multe dintre sistemele multiprocesor se bazează pe microprocesoare comerciale disponibile. Adesea, aceste microprocesoare suportă instrucțiuni de bază similare cu instrucțiunea Test&Set. De exemplu, seria de microprocesoare Motorola 88000 dispune de o instrucțiune, Xmem – ce interschimbă o locație de memorie cu un registru, pentru implementarea sincronizării în sisteme multiprocesor. În mod similar, procesoarele Intel Pentium dispun de instrucțiunea Xchg, similară cu cea de la procesoarele Motorola. O descompunere a uneia dintre aceste instrucțiuni ar arăta în felul următor:

```
Xchg(L,R)
      {
      temp=L;
      L=R;
      R=temp;
      }
```

În general instrucțiunile de interschimbare necesită o secvență de operații de citire, modificare şi scriere. Pe parcursul acestei secvențe procesorului trebuie să dețină exclusivitate asupra locației de memorie la care se face referire. Pentru a asigura acces exclusiv operațiilor atomice executate de procesor, în sistemele multiprocesor, este necesară existența unui semnal ce indică celorlalte componente din sistem derularea unei succesiuni citire – modificare - scriere atomice (neîntreruptibilă). Procesoarele Intel Pentium dispun de un semnal extern, denumit lock#, pentru a efectua operații atomice de acces la memorie. Când acest semnal este activ se indică faptul că microprocesorul (sau componenta) care a inițiat ciclul magistrală curent nu trebuie întrerupt. Acest mecanism hardware este utilizat la implementarea instrucțiunilor de tip Xchg (respectiv Lock / Unlock).

Wait and Signal (sau Increment şi Decrement) sunt operațiile clasice P şi V asupra semafoarelor despre care s-a discutat în precedentul an universitar. Reprezintă o alternativă la instrucțiunile Lock şi Unlock. Decrementează sau incrementează o locație de memorie fără ca operația să poată fi întreruptă (operație atomică). În unele situații, când mai mult de unui proces i se permite intrarea într-o secțiune critică, implementarea protecției este mai eficientă (necesită mai puține instrucțiuni) utilizând instrucțiunile Wait & Signal decât utilizând instrucțiunile Lock & Unlock deoarece cele din urmă manipulează doar valori binare (0/1) pe când Wait & Signal operează pe o variabilă de tip semafor care poate lua valori într-o plajă mult mai largă.
Operațiile atomice Wait & Signal pot fi detaliate în felul următor:

```
Wait(S)
      {
      Wait (S<=0) NOP;
      S=S-1;
      }
Signal(S)
      {
      S=S+1;
      }
```

Utilizarea acestora pentru protecția accesului la o secțiune critică:

```
Wait(semafor)
      <secțiune critică>
Signal(semafor)
```

Fetch & Add. Sincronizarea prin metodele precedente este dezavantajoasă când se încearcă execuția unui număr mare de instrucțiuni lock/unlock respectiv wait/signal. Dacă n procese încearcă simultan accesul la o zonă de memorie comună va fi necesară execuția succesivă a n instrucțiuni lock sau wait chiar dacă un singur procesor va reuşi, în mod firesc, obținerea accesului (şi acest lucru doar la prima tentativă de acces!!!). Această acțiune de acces simultan la memorie produce strangulări în execuție când numărul de procese ajunge de ordinul sutelor.
O modalitate de soluționare a problemei constă în folosirea instrucțiunii Fetch&Add(x,c) al cărei efect constă în creşterea cu valoarea constanta c a conținutului locației comune de memorie x, returnând valoarea lui x înainte de efectuarea adunării (succesiune de operații privite ca o singură operație atomică):

```
Fetch&Add(x,c)
      {
      temp = x;
      x = temp + c;
      return temp;
      }
```

În acest caz, în loc de a concura doar pentru obținerea accesului la x, procesoarele primesc pe lângă accesul la x şi valoarea actualizată (de procesele anterioare) a acesteia.

Luând ca exemplu adunarea paralela a doi vectori :

```
for(i=1;i<m;i++)
      z[i]= x[i] + y[i];
```

Cele n procesoare execută instrucțiunea Fetch&Add(next_index,1). Valoarea next_index este inițializată cu 1. Fiecare proces primeşte valoarea actualizată a variabilei comune next_index şi la rândul său actualizează next_index prin intermediul aceleiaşi instrucțiuni (prin care se face şi accesul). Fiecare procesor va executa următoarea secvență:

```
int i;
i = Fetch&Add(next_index,i);
while (i<m)
      {
      z[i] = x[i] + y[i];
      i = Fetch&Add(next_index,i);
      }
```

Barrier. Bariera este un punct din program la care un număr stabilit de procese trebuie să ajungă înainte ca execuția acestora să poată continua.

Exemplu:

```
sum = 0;
for(i=1;i<=10;i++)
      sum = sum + a[i];
for(i=1;i<=10;i++)
      b[i] = b[i] / sum;
```

Fie un sistem cu două procesoare în care se pot genera două procese notate 0 şi 1. În prima fază a calculului cele doua procese efectuează simultan evaluarea sumei. Doar când ambele au terminat evaluarea se poate trece la faza următoare, ponderarea elementelor lui b. Procesorul 0 lucrează asupra elementelor impare , procesorul 1 asupra celor pare. Variabila id indică numărul procesorului (0 sau 1).

```
int i, partial_sum;
partial_sum = 0;
for(i=1+id;i<10;i=i+2)
      partial_sum = partial_sum + a[i];
Lock(L);
      sum = sum + partial_sum;
Unlock(L);
BARRIER(2); // nici unul din procese nu poate trece de barieră
            // până când ambele au ajuns la acest punct
for(i=1+id;i<10;i=i+2)
      b[i]= b[i] / sum;
```

Implementarea funcției BARRIER se poate face ca mai jos:

```
BARRIER(n)
      {
      Lock (barrier_lock);
      if(number_of_blocked_processes<n-1)
            BLOCK
      WAKE_UP
      }
```

Prin number_of_blocked_processes s-a notat o variabilă comună care reprezintă numărul de procesoare intrate în zona barierei până la momentul curent. Valoarea sa inițială este 0. Un proces intrat în zona barierei când numărul de procesoare blocate (aşteptând posibilitatea de trecere a barierei) este mai mic decât n-1 execută funcția BLOCK care îl suspendă şi îl ataşează unei liste de aşteptare. În acelaşi timp execuția lui BLOCK eliberează accesul în zona barierei (care este o zonă critica) prin execuția Unlock(barrier_lock). Dacă numărul de procesoare blocate este n-1, al n-lea proces care intră în zona barierei execută WAKE-UP si părăseşte zona barierei fără a mai debloca bariera - Unlock(barrier_lock). WAKE-UP are ca efect eliberarea unuia din procesele blocate (daca un astfel de proces există). Procesul eliberat îşi continuă execuția de la punctul de la care aceasta a fost suspendată, adică execută WAKE-UP şi părăseşte bariera. Lucrurile continuă în acest mod pana când toate procesele

trec de barieră, fiecare eliberând următorul proces. Ultimul proces, chiar înainte de a părăsi bariera o deblochează executând Unlock(barrier_lock).

```
BARRIER(n)
      {
      Lock(barrier_lock)
      if(nr_of_blocked_processes<n-1)
            {
            nr_of_blocked_processes=nr_of_blocked_processes+1;
            Unlock(barrier_lock);
            Lock(block_lock);
            }
      if(nr_of_blocked_processes>0)
            {
            nr_of_blocked_processes=nr_of_blocked_processes+1;
            Unlock(block_lock);
            }
      else
            Unlock(barrier_lock);
      }
```

Codul de mai sus funcționează corect cu condiția ca variabila de blocare barrier_lock sa fie inițializată cu 0 si block_lock să fie inițializată cu 1. Primele n-1 procese care intră în zona barierei incrementează variabila nr_of_blocked_processes şi se suspendă prin Lock(block_lock). Ultimul proces care intră în zona barierei decrementează nr_of_blocked_processes şi execută Unlock(block_lock) pentru a elibera unul din procesele blocate.

Programarea paralelă poate conduce la situații de blocare reciprocă a proceselor – situații de **Deadlock**. Acestea apar în momentul în care două sau mai multe procese solicită exclusivitatea pentru o anume resursă şi în acelaşi timp dețin exclusivitatea pentru o alta în mod circular față de celelalte procese implicate. De exemplu:

Perioadă de timp	**Proces 1**	**Proces 2**
t_0	Lock(A)	
t_1		Lock(B)
t_2	Lock(B)	
t_3		Lock(A)

Având implicate două procese (1 şi 2) şi două variabile partajate (A şi B) situația de blocare (deadlock) apare ca urmare a blocării succesive a celor două variabile în vederea utilizării exclusive, fiecare proces deținând accesul exclusiv la una din cele două variabile, ulterior fiecare proces dorind blocarea şi celei de a doua variabile. Procesul 1 aşteaptă deblocarea variabilei B, procesul 2 deblocarea variabilei A. Ambele procese rămân blocare la infinit aşteptând deblocarea variabilei deținute de celălalt proces şi nedeblocând niciodată variabila proprie. Astfel de situații trebuiesc evitate în momentul scrierii programelor neexistând mecanisme care să deblocheze procesele fără întreruperea execuției lor.

2.2 Programare paralelă pe multicalculatoare

Spre deosebire de multiprocesoare, multicalculatoarele constau din procesoare (noduri) care nu se pot coordona prin memorie comună ci prin mesaje. Similar mediului multiprocesor, crearea de către procesul master a unui proces slave produce un nou identificator de proces (process id) care este făcut cunoscut ambelor procese. Aceste id-uri sunt folosite pentru transmiterea de mesaje între procese.
În plus față de id, procesul master specifică numele (adresa) nodului care execută procesul creat. Daca există cel puțin două procese poate avea loc schimbul de mesaje. Un mesaj poate fi un mesaj de comandă sau un mesaj de date. Mesajele sunt dirijate pe diverse trasee in funcție de lungime şi/sau adresa de destinație.

Aceste diferențe sunt invizibile programatorului. În general mesajele pot aştepta într-un şir, fie la nodul de transmisie, fie la cel (sau cele) de recepție, fie în noduri intermediare. În consecință, apar întârzieri arbitrare între transmisie şi recepție, dar, de regulă, ordinea mesajelor se conservă în rețea.
Fiecare mesaj conține informații suplimentare cum ar fi identificatorul procesului de destinație şi lungimea mesajului. În unele implementări se menționează şi identificatorul procesului sursă. Când un proces transmite un mesaj începe prin a-i aloca un tampon. Comanda de transmisie propriu-zisă este emisă după construirea în tampon a mesajului:

```
p =  mem_allocate(message_length);
            // p = pointer către tamponul alocat
send(p, message_length, source_id, destination_id);
```

Un proces poate primi un mesaj prin:

```
p = receive_b();
```

unde b de la sfârşitul lui receive_b arată că o astfel de comandă este o funcție cu blocare, adică una care nu cedează controlul decât după sosirea mesajului solicitat. Funcția alocă un tampon cu dimensiunea mesajului recepționat şi returnează p, pointerul către tamponul ce conține mesajul recepționat.
Spre deosebire de multiprocesoare, comunicarea procesor-memorie nu reprezintă o problemă în sisteme multicalculator. Transferul de mesaje între procese este mai puțin frecvent decât accesul la memorie. Chiar dacă, în comunicația între procese, apar întârzieri, realizarea unui sistem cu mii de noduri nu reprezintă o dificultate majoră.
Unele din caracteristicile multicalculatoarelor pot fi implementate în sisteme multiprocesor prin stabilirea unei zone comune de memorie (însoțită de pointeri adecvați) pentru comunicarea prin mesaje. De asemenea se poate folosi mecanismul de mailbox cu asigurarea accesului prin excludere mutuală la acesta.

3. Noțiuni de calcul paralel

Pentru ca un program să se poată executa pe un calculator paralel trebuie descompus într-o serie de procese. Descompunerea presupune partiționare (partitioning) şi alocare (assignment). Partiționarea constă în specificarea setului de taskuri (sau sarcini) care implementează în maniera cea mai eficientă algoritmul pe o maşină paralelă. Alocarea reprezintă procesul de distribuire a taskurilor (sau sarcinilor) procesoarelor.

3.1 Partiționarea

Performanța unui algoritm paralel depinde de **granularitatea** programului. Granularitatea se referă la dimensiunea taskului în comparație cu consumul suplimentar de resurse necesare implementării (sincronizare, regiuni critice, comunicare). La taskurile de mari dimensiuni volumul de calcule executate este mult mai mare decât cel necesar implementării. Deci, o soluție pentru calculul paralel ar fi partiționarea în taskuri de mari dimensiuni, ceea ce conduce la granularitate grosieră (coarse granularity parallelism). Pe de altă parte, taskurile de mari dimensiuni rezultă în reducerea numărului de procese, deci a gradului de paralelism, ceea ce face ca dezirabilă partiționarea problemei într-un număr mare de taskuri de mici dimensiuni (fine granularity parallelism).

Cu alte cuvinte, îmbunătățirea performanțelor algoritmilor de calcul paralel se face prin găsirea unui compromis între dimensiunea taskurilor şi consumul suplimentar de resurse necesare implementării. O soluție constă în aşa numita „clustering”, adică gruparea taskurilor astfel încât consumul suplimentar de resurse în interiorul grupului să fie mai mare decât cel între grupuri.
În practică numărul de procesoare se corelează de regulă cu dimensiunea problemei pentru ca durata rulării să fie menținută sub o limită impusă. Pentru atingerea acestui obiectiv este important ca algoritmul să folosească toate procesoarele disponibile, precum şi menținerea la minimum necesar a consumului suplimentar de resurse.

Există două metode de partiționare: statică şi dinamică. Metoda statică partiționează taskurile înainte de execuție. Ea are avantajul de a conduce la un volum redus de comunicații între procese şi la diminuarea concurenței între acestea. Dezavantajul metodei este dat de posibilitatea ca datele de intrare să dicteze gradul de paralelism şi, în consecință, ca unele procesoare să nu fie utilizate în cadrul procesului de calcul.

Metoda de partiționare dinamică generează taskurile în timpul execuției. Avantajul partiționării dinamice este tendința de a menține procesoarele ocupate, dar aceasta are loc cu prețul creşterii volumului de comunicații între procese.

Procesele pot fi create astfel încât toate procesele efectuează aceeaşi funcție pe diferite secțiuni de date sau astfel încât fiecare din ele efectuează o funcție distinctă de ale celorlalte procese. Prima abordare se numeşte data partitioning (sau data parallelism), iar cea de a doua function partitioning (control parallelism sau functional parallelism). Întrucât data partitioning implică generarea de procese identice ea se mai numeşte homogenous multitasking. Din motive similare, cealaltă abordare se numeşte şi heterogeneous multitasking.

Data partitioning extrage paralelismul din organizarea datelor problemei. Structura de date este divizată în secțiuni, fiecare secțiune fiind prelucrată în paralel cu celelalte. Partiționarea datelor este recomandată în rezolvarea de probleme numerice ce conțin matrici sau vectori de mari dimensiuni. De asemenea ea este utilizată cu succes în probleme numerice de tipul sortare sau căutare. Metoda este potrivită pentru multicalculatoare, ca urmare a necesităților reduse de comunicare între procesoare.

Pentru exemplificare se consideră:

$$Z[i] = (A[i]*B[i]) + (C[i]/D[i]) \quad \text{pentru } i = 1 - 10$$

În cazul partiționării datelor se creează 10 procese identice, fiecare executând calculele pentru o valoare dată a indicelui i.

În cazul partiționării funcțiilor se creează două procese diferite P1 şi P2.
P1 efectuează x=A[i]*B[i] după care transmite x lui P2;
P2 efectuează y=C[i]/D[i] şi după punerea lui x de la P1, calculează z=x+y.

Pe ansamblu, partiționarea datelor este mai avantajoasă, deoarece:
- oferă un grad mai mare de paralelism;
- asigură o încărcare echilibrată a procesoarelor;
- este mai uşor de implementat.

3.2. Alocarea (planificarea)

Alocarea (sau planificarea) constă în distribuirea de taskuri (sarcini de lucru) procesoarelor. Ea poate fi statică sau dinamică.

În cazul alocării statice setul de sarcini şi ordinea de execuție sunt cunoscute înainte de execuție. Algoritmii de planificare statică necesită un volum mic de comunicație între procese şi sunt potrivite pentru situațiile în care costul comunicațiilor este mare. De asemenea „costurile” planificării sunt suportate o singură dată, chiar dacă acelaşi program este rulat de un număr mare de ori cu date diferite.

În cazul planificării dinamice alocarea sarcinilor de lucru se face în timpul rulării. Această tehnică permite o utilizare echilibrată a procesoarele şi oferă flexibilitatea în utilizarea unui număr variabil de procesoare (lucru util mai ales când numărul de procese depinde de dimensiunea datelor de intrare).
Ca dezavantaje se menționează:
- structura programelor devine dificil de înțeles;
- detectarea blocărilor (deadlock) se face cu dificultate;
- analiza performanțelor devine uneori imposibilă, ca urmare a alocării taskurilor la momentul rulării;
- volumul de comunicații şi competiția între procese sunt mari.

4. Structuri de algoritmi

O posibilă definiție a unui algoritm de calcul paralel pentru un sistem cu procesoare multiple poate fi: „colecție de procese concurente care se execută simultan pentru a rezolva o problemă dată”. Algoritmii de calcul paralel se pot clasifica în algoritmi sincroni, asincroni şi cu structură pipeline.

4.1. Structuri sincrone

Algoritmii din această categorie se caracterizează de faptul că două sau mai multe procese sunt legate printr-un punct comun de execuție, folosit pentru sincronizare. Un proces ajunge la un punct al execuției sale la care trebuie să aştepte ca unul sau mai multe procese să atingă un anumit punct. După ce procesele au ajuns la punctul de sincronizare ele îşi pot continua execuția. Ca efect, procesele ce trebuie să se sincronizeze la un anumit punct de execuție îl aşteaptă pe cel mai lent, fapt ce constituie principalul dezavantaj al algoritmilor sincroni. Ei se mai numesc şi „partitioning algorithms”.

Problemele numerice de mari dimensiuni (cum sunt cele ce necesită rezolvarea unor sisteme mari de ecuații) se pretează utilizării algoritmilor sincroni. Adesea, tehnicile utilizate presupun efectuarea unui număr de iterații folosind matrici „voluminoase”. Fiecare iterație foloseşte rezultatele parțiale de la pasul precedent. Efectuarea fiecărui pas se poate paraleliza prin folosirea unui număr de procesoare la prelucrarea a câte unei secțiuni a matricei. După fiecare pas procesele trebuie să se sincronizeze înainte de a începe pasul următor.

Algoritmii paraleli sincroni se pot implementa atât pe multiprocesoare cât şi pe multicalculatoare. În implementarea pe structuri de tip message-passing comunicația între procese se face în mod explicit, folosind mecanismele oferite de structură. În sistemele cu memorie comună comunicația poate fi explicită sau implicită (prin referirea de zone comune de memorie).

Pentru exemplificare se foloseşte următorul calcul asupra a patru vectori A, B, C, D, în condițiile existenței a două procesoare:

```
for (i=1; i <10; i++)
      {
      Z[i] = (A[i]*B[i]) + (C[i]/D[i]);
      }
```

Paralelizarea este evidentă. Se creează două procese P1 şi P2.
Pentru toate valorile lui i P1 efectuează x=A[i]*B[i],
iar P2 calculează y=C[i]/D[i] şi Z[i]=x+y.

În cazul comunicării explicite dintre procese P1 transmite lui P2, pentru fiecare valoare a lui i, valoarea x pe care a evaluat-o. La rândul său P2 evaluează y şi, la primirea mesajului, evaluează Z[i].
Când procesele comunică implicit, P2 evaluează y şi verifică dacă P1 a evaluat x. Dacă da, preia x din memorie şi evaluează Z[i]. În caz contrar aşteaptă până când P1 evaluează x.

Principalii paşi ai algoritmului implicit de mai sus sunt descrişi în codul:

```
struct global_memory //se creează structura de memorie partajată
      {
      shared int next_index;
      shared int A[10], B[10], C[10], D[10], Z[10];
      shared int x;
      shared char turn[6];
      }
main()
      {
      int y;
      next_index = 1;
      turn = ,slave’;
      // se creează un proces slave,
      // care constă în execuţia procedurii slave
      CREATE(slave)
      while(next_index <= 10)
            {
            y = C[next_index] / D[next_index];
            while (turn == ,slave’) NOP;
            Z[next_index] = x+y;
            next_index = next_index+1;
            turn=’slave’;
            }
      PRINT_RESULT
      }
slave()
      {
      while(next_index <= 10)
            {while (turn == ,master’) NOP;
            x = A[next_index]*B[next_index];
            turn = ,master’;}
      }
```

Vectorii A, B, C, D, Z se află în memoria globală comună. De îndată ce procesul principal, numit master, a alocat memoria, execută instrucțiunea CREATE, care conduce la crearea unui alt proces, numit slave. Procesul slave îşi începe execuția cu procedura slave, specificată drept argument al lui CREATE.

4.2 Structura asincronă

Algoritmii paraleli asincroni sunt caracterizați de faptul că permit funcționarea proceselor cu cele mai recente valori ale datelor. Ei se pot implementa atât pe multiprocesoare cât şi pe multicalculatoare. În cazul modelului cu memorie comună există un set de variabile globale accesibile tuturor proceselor. La terminarea de către un proces a unei etape a programului el citeşte anumite variabile globale. Pe baza valorilor acestora şi a rezultatelor obținute în ultima etapă de prelucrare procesul activează etapa următoare şi actualizează anumite variabile globale.

În cazul implementării pe o structura de tip message passing un proces completează o etapă a programului după care citeşte anumite mesaje de intrare. Pe baza acestor mesaje şi a rezultatelor obținute anterior procesul activează următoarea etapă şi transmite mesaje spre alte procese.

În concluzie, un algoritm asincron continuă sau termină un proces în funcție de valori ale unor variabile globale (sau mesaje primite) şi nu aşteaptă un set de intrări, ca în cazul algoritmului sincron. Întrucât algoritmii asincroni nu presupun sincronizarea pentru certitudinea existenței datelor de intrare, ei se mai numesc şi relaxed algorithms.

Pentru exemplificare se consideră aceeaşi problemă ca în cazul algoritmilor sincroni.

$$Z[i]=(A[i]*B[i])+(C[i]/D[i]) \qquad i = 1..10$$

Fiecare proces solicită o valoare a lui i, după care evaluează Z[i] pentru valoarea obținută i, după care solicită o nouă valoare a lui i şi începe etapa următoare. Procedura continuă până la epuizarea valorilor i.

```
struct global_memory
      {
      shared int next_index;
      shared int A[10], B[10], C[10], D[10], Z[10];
      }
main()
      {
      // se creează un proces slave,
      // care constă în execuția procedurii slave
      CREATE(slave);
```

```
        task();
        WAIT_FOR_END; //aşteaptă terminarea procesului slave
        PRINT_RESULT;
        }
slave()
        {
        task();
        }
task()
        {
        int i;
        GET_NEXT_INDEX(i);
        while(i > 0)
              {
              Z[i]=(A[i]*B[i])+(C[i]/D[i]);
              GET_NEXT_INDEX(i);
              }
        }
```

În general algoritmii asincroni sunt mai eficienți decât cei sincroni din următoarele motive:

1. procesele nu aşteaptă date de intrare de la alte procese;
2. rezultatele proceselor care rulează rapid pot fi folosite pentru dezactivarea proceselor mai lente, căci condițiile o permit;
3. algoritmii asincroni sunt mai fiabili;
4. algoritmii asincroni prezintă probleme mai puțin complexe de concurență pentru memorie, mai ales când algoritmul este bazat pe partiționarea datelor.

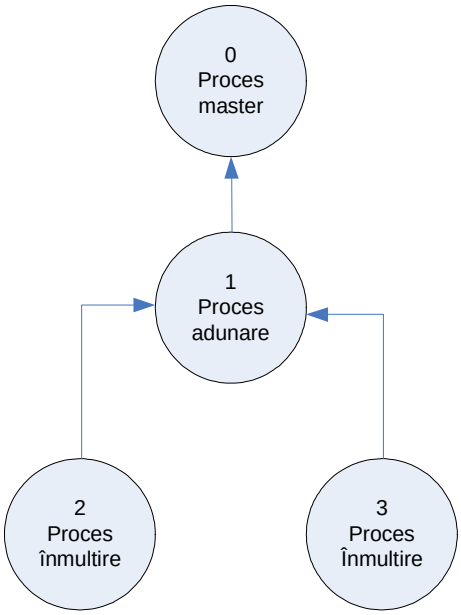
Din dezavantajele algoritmilor asincroni se menționează analiza mai delicată a acestora. Ca urmare a modului dinamic în care au loc comunicațiile, uneori analiza este imposibilă.

4.3. Structuri pipeline

În cazul algoritmilor pipeline, procesele sunt structurate şi datele sunt transmise de la un proces la altul ca într-o structură pipeline. Calculul progresează în etape. Acest tip de prelucrare se numeşte şi macro pipelining. Aceşti algoritmi sunt utili în cazul problemelor ce se pot descompune într-o mulțime finită de procese secvențiale.

Comunicațiile de date între procesele unei structuri pipeline pot fi sincrone sau asincrone. În cazul sincron se foloseşte un mecanism global de sincronizare, similar cu un ceas. Structura asincronă presupune sincronizarea unui proces cu vecinii săi prin intermediul unui mecanism local, de exemplu prin transfer de mesaje.

Fie exemplul Z[i]=(A[i]*B[i])+(C[i]/D[i]), i = 1..10, şi patru procese, din care unul master.



Structura de comunicație între procesoarele care cooperează este cea din figură. Procesul master (0) creează procesele slave (1,2,3), le transmite mesaje de inițializare şi aşteaptă să-şi îndeplinească sarcinile de prelucrare şi să returneze rezultatul.

Descrierea funcțională a acestor procese este:

Pentru procesul master (0):

```
master:     process
                  {
                  inițializarea mediului;
                  citirea datelor de intrare;
                  crearea proceselor de adunare şi înmulțire;
                  pentru fiecare proces de înmulțire
                        {
                        transmiterea către proces a unei perechi de
                        vectorii ce trebuie înmulțiți împreună cu id-ul
                        procesului de adunare ce primeşte ieşirea;
                        }
                  trimite către procesul de adunare un mesaj conținând
                  id-urile proceselor de înmulțire şi id-ul procesului
                  master;
                  atâta timp cât nu s-au primit toate elementele Z
                        {
                        primeşte mesaj de la procesul de adunare şi
                        mută valoarea primită în vectorul Z;
                        }
                  tipăreşte Z;
                  aşteaptă dezactivarea proceselor de adunare şi
                  înmulțire;
                  }
```

Pentru procesele de înmulțire (2,3):

```
multiply:   process
            {
            primeşte vectorii de înmulțit şi id-ul procesului de
            adunare;
            mută vectorii primiți în x şi y;
            for (i=0; i<=10; i++){
```

```
                {
                RESULT = x[i]*y[i];
                Transmite RESULT procesului de adunare;
                }
          transmite mesaj END procesului de adunare;
          }
```

Pentru procesul de adunare (1):

```
add:  process
      {
      recepţionează mesaj cu id-urile proceselor master şi a celui de
      înmulţire;
      primeşte mesaj de la procesul din stânga;
      înscrie rezultatul în LEFT_RESULT;
      primeşte mesaj de la procesul din dreapta;
      înscrie rezultatul în RIGHT_RESULT;
      while (nu s-a primit mesaj END)
            {
            RESULT=LEFT_RESULT + RIGHT_RESULT;
            transmite RESULT procesului master;
            primeşte mesaj de la procesul din stânga;
            înscrie rezultatul în LEFT_RESULT;
            primeşte mesaj de la procesul din dreapta;
            înscrie rezultatul în RIGHT_RESULT;
            }
      transmite mesaj END procesului master;
      }
```

5. Analiza algoritmilor paraleli

5.1 Parametrii de performanță

Există trei factori importanți în analiza algoritmilor paraleli:

- câştigul de timp în execuție (factor de accelerare);
- eficiența de utilizare a proceselor;
- costul de execuție paralelă (în raport cu execuția secvențială).

Se definesc următorii parametrii:

1. **Lucrul** (**W** – work) ca fiind numărul total de operații care se execută în cadrul unui algoritm paralel; este dependent de dimensiunea setului de date - n, *W(n)*, de intrare şi de numărul de procesoare - p, *W(p)*.

2. **Factorul de accelerare** – factor extrem de important în analiza unui algoritm de calcul paralel.

$F_P = \frac{T_S}{T_P}$, unde T_S este timpul de execuție al algoritmului secvențial şi T_P timpul de execuție al algoritmului paralel.

$F_P = p$(nr. de procesoare) în cazul ideal, factor de accelerare ideal.

$F_P = c*p$, c>0; factor de accelerare linear (ideal dacă c = 1).

$F_P = \frac{\Delta s + \Delta p}{\Delta s + \frac{\Delta p}{p}} = \frac{1}{\Delta s + \frac{\Delta p}{p}} < \frac{1}{\Delta s}$, factor de accelerare fix; normalizând prima fracție ($\Delta s+\Delta p=1$) se obține că este totdeauna mai mic decât 1/Δs (pentru $p \to \infty$); Δs reprezintă fracțiunea din timpul de execuție a algoritmului pentru porțiunea care nu poate fi paralelizată (strict serială), Δp timpul de execuție a algoritmului pentru porțiunea care poate fi paralelizată şi p numărul de procesoare.

$F_P = \frac{\Delta s + \Delta p'*n}{\Delta s + \Delta p'} = \Delta s + \Delta p'*p = p - \Delta s*(p-1)$, factor de accelerare scalabil; se exprimă în funcție de Δp' – timpul de execuție a porțiunii din algoritm ce poate fi paralelizată după paralelizare. Se normalizează considerând $\Delta s + \Delta p' = 1$.

În practică, se poate obține o accelerare mai mare decât cea ideală (egală cu numărul de procesoare) datorită specificului arhitectural al maşinii hardware care pune algoritmul secvențial într-o situație defavorabilă de execuție (de exemplu un algoritm ce utilizează un volum foarte mare de date ce nu pot fi stocate în memoria principală ci doar în cea secundară, cu un timp de acces mai mare – algoritmul secvențial va avea penalizări de acces la memoria secundară pe când algoritmul paralel va putea distribui datele în mai multe memorii principale ale diverselor procesoare).

3. **Eficiența paralelă** (**E_P**) – se defineşte ca raportul dintre accelerarea paralelă şi numărul de procesoare.

$$E_P = \frac{F_P}{p} = \frac{T_S}{p*T_P}$$

4. **Costul paralel** (**C_P**) – se defineşte ca suma timpilor consumați de toate procesoarele pentru rezolvarea problemei.

$C_P = p * T_P$

În cazul unui algoritm secvențial (p=1) $C_S=T_S$ şi eficiența se mai poate exprima ca raportul între costul secvențial şi cel paralel:

$E_P = \frac{C_S}{C_P}$.

Un algoritm este optimal din punct de vedere al costului dacă costul acestuia este proporțional cu timpul de execuție al algoritmului secvențial:

$p * T_P = k * T_S$

adică

$E_P = \frac{F_P}{p} = \frac{T_S}{p * T_P} = \frac{1}{k}$, a se citi: eficiența unui algoritm paralel cost-optimal este constată în raport cu dimensiunea problemei şi cu numărul de procesoare utilizate.

5. **Redundanța** se defineşte ca raportul dintre lucrul total al procesoarelor ce lucrează în paralel şi lucrul total al unui calculator secvențial (în ambele cazuri lucru necesar rezolvării unei probleme date).

$$R_P = \frac{W(p)}{W(1)}$$

6. **Utilizarea** este raportul dintre lucrul total al proceselor paralele şi costul paralel.

$$U_P = \frac{W(p)}{C_P} = \frac{W(p)}{p * T_P}$$

7. **Calitatea** se defineşte ca:

$$Q_P = \frac{T_S^3}{C_P * W(p)} = \frac{T_S^3}{p * T_P * W(p)}$$

Între parametrii de performanță descrişi mai sus se pot demonstra uşor relațiile:

$U_P = R_P * E_P$

$Q_P = E_P * \frac{F_P}{R_P}$

$\frac{1}{p} \le E_P \le U_P \le 1$

$1 \le R_P \le \frac{1}{E_P} \le p$

$Q_P \le F_P \le p$

5.2 Factori ce afectează performanța unui algoritm paralel

Încărcarea neechilibrată a procesoarelor provine din:

- Imposibilitatea împărțirii în task-uri perfect egale;
- Necesitatea sincronizării proceselor ce dă naştere unor timpi suplimentari de aşteptare;
- Variația gradului de paralelism în cadrul algoritmului.

Calculele suplimentare ce apar în cazul în care cel mai rapid algoritm secvențial nu poate fi paralelizat şi se alege un algoritm mai greoi (lent) dar paralelizabil.

Comunicația între procese necesară transferări de operanzi, sincronizări etc.

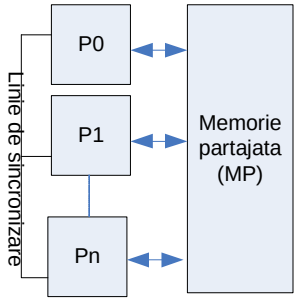
Concurența la setul de date partajat (Lock/Unlock).

5.3 Modele teoretice utilizate în analiza eficienței algoritmilor paraleli

Analiza complexității algoritmilor de calcul paralel presupun folosirea unor modele formale cu ajutorul cărora să se permită evaluarea parametrilor de performanță (prezentați anterior).

Modelele formale sunt dependente de caracteristicile maşinilor paralele care le modelează; folosirea unui model particular care să surprindă cât mai fidel caracteristicile fiecărei categorie de maşini este nepractică datorită numărului mare de modele necesare; folosirea unor modele abstracte cu un grad de generalitate mai mare permite dezvoltarea unor algoritmi de calcul paralel eficienți care pot fi traduşi cu uşurință în funcție de arhitecturi particulare.

Modelul PRAM (Parallel Random Access Machine) consideră că timpul de acces la memorie şi costul de sincronizare sunt nule. Este utilizat pentru analiza timpului de execuție, accelerare, eficiență şi scalabilitate.



Memoria partajată poate fi centralizată sau distribuită. Procesoarele operează sincron operații de citire din memorie, calcule, scrieri în memorie. Model ideal, nu încearcă modelarea unei maşini reale ci doar permite urmărirea aspectelor de concurență.

Modelul PRAM are următoarele opțiuni de acces concurent la memoria partajată:

- ER (Exclusive Read) – câte o citire per ciclu
- EW (Exclusive Write) – câte o scriere per ciclu
- CR (Concurent Read) – mai multe procesoare pot citi simultan o locație de memorie
- CW (Concurent Write) – mai multe procesoare pot înscrie simultan o locație de memorie

Din aceste opțiuni de acces rezultă următoarele variante ale modelului:

- EREW-PRAM – cea mai restrictivă variantă, nu se pot executa simultan nici citiri, nici înscrieri
- CREW-PRAM – variantă bazată pe excludere mutuală
- ERCW-PRAM – variantă rezultată pur combinațional, nu este utilizată
- CRCW-PRAM – cea mai interesantă variantă, în funcție de cum se rezolvă scrierile concurente putem avea:
    - Common PRAM – se permit înscrieri concurente doar pentru valori identice
    - Arbitrary PRAM – o singură valoare, arbitrar aleasă, este înscrisă, restul se ignoră
    - Priority PRAM – în funcție de o listă de priorități se alege valoarea care se înscrie, restul se ignoră
    - Function PRAM – se înscrie o valoare care rezultă din prelucrarea tuturor valorilor ce se încearcă a fi înscrise (suma lor de exemplu).

Modelul bazat pe arbori de lucru în adâncime (work-depth). Este orientat spre algoritm. Nu ține seama de particularitățile sistemului de calcul paralel. Algoritmii sunt reprezentați prin grafuri aciclice orientate (sau arbori binari algebrici) respectând următoarele reguli:

- Fiecare intrare este reprezentată printr-un nod terminal (frunză).
- Fiecare operație este reprezentată ca un nod al arborelui (nu frunze), operanzii fiind reprezentați ca intrări în noduri.
- Ieşirea este reprezentată de nodul rădăcină.

Modelul este independent de arhitectură şi de numărul de procesoare. Pune în evidență două aspecte: lucrul (W) efectuat de algoritm (numărul total de operații) şi adâncimea algoritmului – cel mai lung lanț de dependențe secvențiale din algoritm (drum frunză – rădăcină).

Modelul poate fi analizat, indiferent de arhitectură, considerând numărul de unități de timp necesare execuției algoritmului; în fiecare unitate de timp executându-se orice număr de operații în paralel.

6. Exemple

6.1 Algoritmi asincroni destinați sistemelor multiprocesor

Înmulțirea a două matrice. Se consideră problema înmulțirii a două matrice (M x M) A şi B (C = A * B) utilizând N procese distincte. O problemă importantă ce trebuie luată în considerare în dezvoltarea unui astfel de algoritm este independența algoritmului față de numărul de procesoare disponibile din sistem. Adică, algoritmul trebuie să dea acelaşi rezultat indiferent dacă rulează pe un sistem cu un singur procesor sau pe un sistem cu mai multe procesoare. Pentru a atinge acest scop, algoritmul gândit pentru un număr N de procese distincte va da naştere la o listă de task-uri ce vor fi alocate dinamic procesoarelor din sistem astfel încât să se obțină o încărcare optimă a acestora (fiecare procesor să fie ținut ocupat pe cât posibil). Task-urile trebuie să fie independente în sensul că odată alocat unui procesor, un task nu va trebui să comunice cu un alt task.

În funcție de N şi M putem considera că fiecare task va calcula un element al matricei rezultat C ($N \sim M^2$) sau o coloană a matricei rezultat ($N << M^2$ – task-uri de dimensiune mare). Cu alte cuvinte cu cât N este mai mare cu atât este mai bine să avem task-uri de dimensiune mai mică (mai multe) pentru a putea asigura scalabilitatea alocării dinamice. Pe altă parte un număr mare de task-uri introduce un timp suplimentar datorat concurenței accesului la variabila index comună. Secvența următoare de pseudo-cod implementează un exemplu de task de calcul la nivel de coloană a matricei rezultat (task-ul de calcul la nivel de element a matricei rezultat este similar).

```
struct global_memory
{
      shared float A(M,M), B(M,M), C(M,M);
}
task ()
{
      int k;
      GET_NEXT_INDEX (k);
      while (k>0) {
            for (i=1; i<=M; i++) {
                  C[i,k]=0;
                  for (j=1; j<=M; j++)
                        C[i,k]=C[i,k]+A[i,j]*B[j,k];
            }
            GET_NEXT_INDEX (k);
      }
}
```

Funcția GET_NEXT_INDEX implementează operația atomică de citire şi incrementare a indexului de coloană a matricei rezultat de calculat. Returnează o

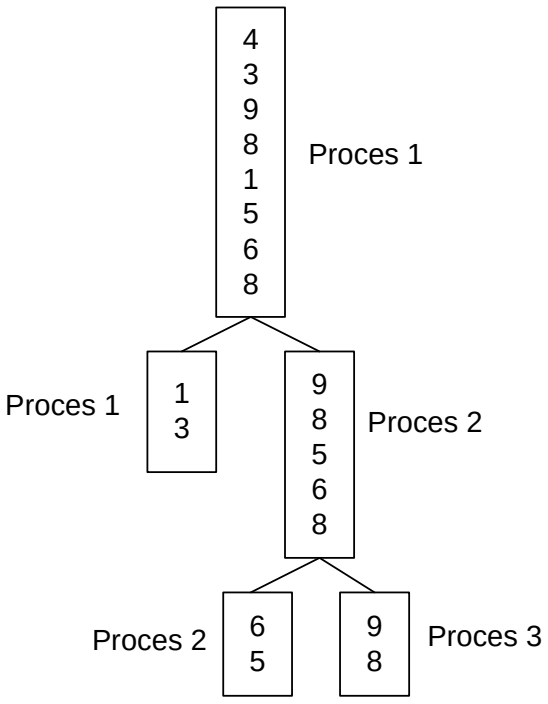
valoare în plaja de valori 1,M atâta timp cât există o următoare coloană de calculat sau -1 în cazul în care nu mai există nici o nouă coloană de calculat.

Quicksort. Cunoscut şi sub denumirea de „partition-exchange sort”, algoritmul quicksort este o tehnică de sortare foarte potrivită pentru procesarea paralelă datorită paralelismului implicit al metodei de sortare. Algoritmul presupune, în mod corect, că o listă de dimensiune 1 este deja sortată. Plecând de la această presupunere se împarte în mod repetat setul de date până când se ajunge la un număr de liste cu un singur element ce se recombină ținând seama de ordinea de împărțire.

Fiind dată o listă de numere, se alege un element a acesteia ca element de partiționare (oricare element, poate fi de exemplu primul element al listei). Restul elementelor rămase se împart în două sub-liste: elemente mai mari sau egale ca elementul de partiționare şi elemente mai mici. Procesul de partiționare se repetă în mod identic pentru cele două sub-liste rezultate (se alege un element de partiționare, se împarte lista în două sub-liste – numere mai mari decât elementul de partiționare şi numere mai mici) şi pentru următoarele sub-liste rezultate până în momentul în care se ajunge ca listele să aibă un singur element. În acel moment lista este sortată în urma recombinării ordonate a sub-listelor de dimensiune 1.

4	**4**	**4**	**4**
3	3	3	3
9	9	9	9
8	8	8	8
1	1	1	*1*
5	5	*5*	5
6	*6*	6	6
8	8	8	8
Pas 1	Pas 2	Pas 3	Pas 4 – exchange
1	1	1	1
3	3	3	3
9	*9*	**4**	**4**
8	8	*8*	8
4	**4**	9	9
5	5	5	5
6	6	6	6
8	8	8	8
Pas 5	Pas 6 – exchange	Pas 7	Pas 8
1	1		1
3	3		3
4	**4**		4
8	**8**		5
9	9		6
5	5		8
6	*6*		8
8	8		9
Pas 9	Pas 10 – exchange	….	Listă sortată

În pasul 1 se alege elementul de partiționare şi se începe compararea cu restul elementelor din listă (parcurgerea începe cu ultimul element al listei). Primele comparări (pas 1 – 4<8 – pas 2 – 4<6 – pas 3 – 4<5) nu conduc la nici o interschimbare a elementelor (elementele parcurse vor aparține sub-listei cu elemente mai mari ca elementul de partiționare). În pasul 4 se găseşte primul element mai mic ca elementul de partiționare şi se realizează prima interschimbare. Procesul de comparare se reia în sens invers (sensul de parcurgere este dat de poziția elementelor cu care se face compararea – dacă elementele se găsesc deasupra elementului de partiționat se parcurge de sus în jos – dacă elementele se găsesc sub elementul de partiționat parcurgerea se face de jos în sus). În pasul 8 prima fază de partiționare este completă având trei sub-liste rezultate: sub-lista cu elemente mai mici ca elementul ales de partiționare, sub-lista care conține elementul de partiționare (de dimensiune 1, sortată) şi sub-lista care conține elementele mai mari ca elementul de partiționare. Se repetă procesul de partiționare pe cele două sub-liste (cu elemente mai mici respectiv mai mari ca elementul de partiționare) până la obținerea de sub-liste de dimensiune 1, implicit sortate. Procesul de recompunere este observabil în pasul 10 pentru lista cu elemente mai mici ca elementul de partiționare inițial (în pasul 9 se compară noul element de partiționare – 1 – cu restul elementelor din sub-listă – elementul 3 – iau naştere două sub-liste: una cu elementul de partiționare si una cu elemente mai mari; ambele sub-liste sunt de dimensiune 1, procesul de partiționare se încheie şi se recompun cele două sub-liste plus sub-lista formată din elementul inițial de partiționare).



O metodă de implementare a algoritmului descris pe un sistem multiprocesor presupune crearea unei liste de sarcini de partiționare care să fie alocate proceselor disponibile. Diagrama alăturată indică un mod posibil de alocare a task-urilor de partiționare.

Inițial nu există ocupat decât un proces inițial care prelucrează lista inițială. Ulterior, pe măsură ce apar noi sub-liste de prelucrat sarcinile sunt distribuite unor alte procese astfel încât să existe o încărcare uniformă.

Eliminare gaussiană. Eliminarea gaussiană este o metodă de rezolvare a sistemelor de ecuații liniare. O exemplificare a metodei pe un sistem cu trei ecuații şi trei necunoscute (x_1, x_2, x_3) poate fi văzută în figura de mai jos.

Sistem de ecuații

$$x_1 + 2x_2 - x_3 = -7$$
$$2x_1 - x_1 + x_3 = 7$$
$$-x_1 + 2x_2 + x_3 = -1$$

Reprezentare sistemului de ecuații sub forma unei matrice

$$\begin{bmatrix} a(1,1) & a(1,2) & a(1,3) & a(1,4) \\ a(2,1) & a(2,2) & a(2,3) & a(2,4) \\ a(3,1) & a(2,3) & a(3,3) & a(3,4) \end{bmatrix} = \begin{bmatrix} 1 & 2 & -1 & -7 \\ 2 & -1 & 1 & 7 \\ -1 & 2 & 3 & -1 \end{bmatrix}$$

Eliminare

$$\begin{bmatrix} 1 & 2 & -1 & -7 \\ 0 & -5 & 3 & 21 \\ 0 & 4 & 2 & -8 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & -1 & -7 \\ 0 & -5 & 3 & 21 \\ 0 & 0 & 4.4 & 8.8 \end{bmatrix}$$

Substituție

$4.4x_3 = 8.8 \Rightarrow x_3 = 2$
$-5x_2 + 3 * 2 = 21 \Rightarrow x_2 = -3$
$x_1 + 2 * (-3) - 1 * 2 = -7 \Rightarrow x_1 = 1$

Un sistem de ecuații poate fi reprezentat ca o matrice formată din coeficienții variabilelor împreună cu termenii liberi. Metoda de rezolvare constă în două etape: eliminare şi substituție.

Etapa de eliminare îşi propune aducerea matricei la formă superior triunghiulară prin selectarea a câte unei linii (linie pivot) şi prin operații succesive de adunare a acesteia cu liniile de dedesubtul ei – operații menite să anuleze pe rând elementele de sub diagonala principală. Adunarea liniei pivot la liniile de dedesubtul acesteia este precedată de multiplicarea cu un coeficient (distinct pentru fiecare operație) care să permită anularea elementelor prin adunare.

Substituția permite determinarea efectivă a variabilelor – pornind de la variabila care în urma triangularizării poate fi calculată imediat şi apoi, prin înlocuirea variabilelor cu valorile calculate în următoarele ecuații, şi a celorlalte variabile.

În cadrul algoritmului de calcul paralel destinat unui sistem multiprocesor fiecare procesor se va ocupa cu adunarea liniei pivot (stabilită înaintea începerii fiecărei iterații) la câte o linie de dedesubtul acesteia (stabilind bineînțeles şi factorul de multiplicare). Procesul inițial se ocupă cu incrementarea liniei pivot şi stabilirea primei linii disponibile de prelucrat (care este de fapt linia imediat următoare liniei pivot). Fiecare iterație a algoritmului constă în stabilirea liniei pivot şi a primei linii de prelucrat (de către procesorul 1), sincronizarea tuturor proceselor (se aşteaptă ca procesele să-şi finalizeze prelucrarea anterioară pentru a nu exista pericolul de a se lucra cu date incorecte) şi în prelucrarea efectivă de eliminare (triangularizare a matricei sistem). În procesul de eliminare fiecare procesor determină linia următoare de prelucrat (prin intermediul instrucțiunii atomice Fetch&Add) şi efectuează calculele necesare pentru anularea elementului din

linia respectivă. Prin intermediul mecanismului de concurență în alegerea următoarei linii de prelucrat se asigură încărcarea optimă a tuturor procesoarelor (procesoarele sunt ocupate pe cât se poate). Exemplul în pseudo-cod de mai jos implementează doar partea de eliminare din algoritmul de eliminarea gaussiană; partea de substituție este pur secvențială.

```
#define NUMPROCS // numărul de procesoare
struct global_memory
{
      shared float a(n,n+1); // matricea sistemului
      shared int p=0, next_row; // p – linia pivot curentă
              // next_row – următoarea linie disponibilă de prelucrat
}
task()
{
      int k; // linia de prelucrat

      while (p < n-1)  {
            if (proc_id==1) { // procesorul 1
                  p = p + 1; // este cel care incrementează linia pivot
                  next_row = p + 1;
            }

            BARRIER(NUMPROCS); // se așteaptă ca toate procesele să-și
                               // încheie prelucrarea

            k = Fetch&Add(next_row, 1);

            while (k <= n) {

                  mult = - a(k,p)/a(p,p);
                  for (i=p; i<=n+1; i++)
                        a(k,i) = a(k,i) + mult * a(p,i);
                  k = Fetch&Add(next_row,1);
                  }
      }
}
```

6.2 Algoritmi sincroni destinați sistemelor multicalculator

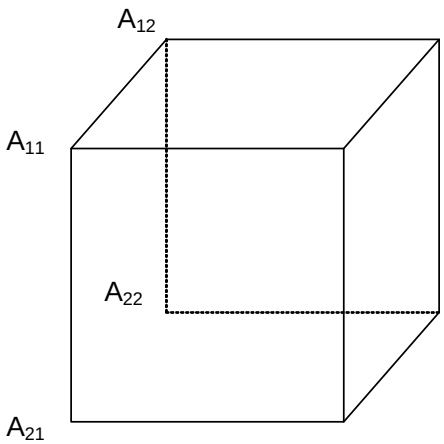
Spre deosebire de sistemele multiprocesor, în cazul sistemelor multicalculator se preferă distribuția datelor în memoriile nodurilor de prelucrare (memoriile locale ale sistemelor). Având în vedere acest lucru în conceperea unui algoritm destinat sistemelor multicalculator se încearcă evitarea referințelor la date care nu se află în memoria proprie a nodului de prelucrare deoarece o astfel de referință ar încetini execuția secvenței de cod prin latența mare de acces la o locație externă.

Înmulțirea a două matrice. În cazul sistemelor multicalculator se preferă o abordare ce implică partiționarea setului de date. O modalitate eficientă de partiționare în cazul înmulțirii a două matrice este spargerea matricelor în sub-blocuri. Fie matricele, de dimensiune MxM, A, B,C unde C = A * B; spargerea identică a celor două matrice A şi B în patru sub-blocuri egale (de dimensiune M/2xM/2) conduce la posibilitatea de a calcula matricea C ca produs a sub-blocurilor matricelor A şi B.

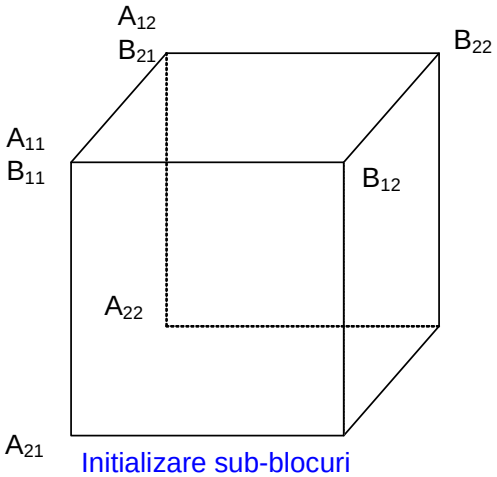
$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11} * B_{11} + A_{12} * B_{21} & A_{11} * B_{12} + A_{12} * B_{22} \\ A_{21} * B_{11} + A_{22} * B_{21} & A_{21} * B_{12} + A_{22} * B_{22} \end{bmatrix}$$

Prin intermediul acestui mecanism de paralelizare putem distribui sub-blocurile celor două matrice în memoriile locale ale nodurilor de prelucrare. Nu mai este necesar ca toate nodurile de prelucrare să aibă acces la matricele A şi B în întregime ci doar la anumite sub-blocuri din acestea (sub-blocurile necesare calculului unui sub-bloc al matricei rezultat – C). Dacă memoriile locale ale nodurilor sunt capabile să stocheze informația necesară calculului unui bloc al matricei rezultat atunci necesarul de comunicație între nodurile de prelucrare în timpul execuției algoritmului este minim.

Vom exemplifica algoritmul pe o structura multicalculator cu o arhitectura de interconectare de tip cub. Într-o primă fază sub-blocurile matricelor A şi B sunt încărcate în memoriilor unor noduri din sistem (se presupune că fiecare nod are capacitatea de a stoca cel puțin două astfel de sub-blocuri).

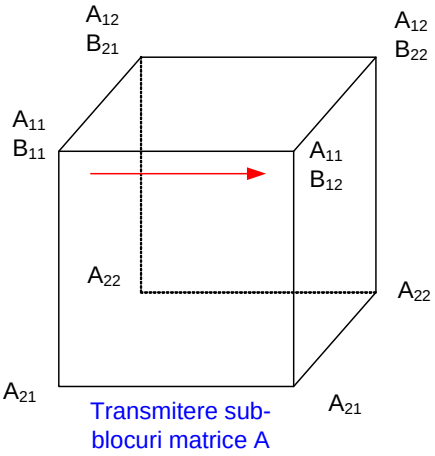


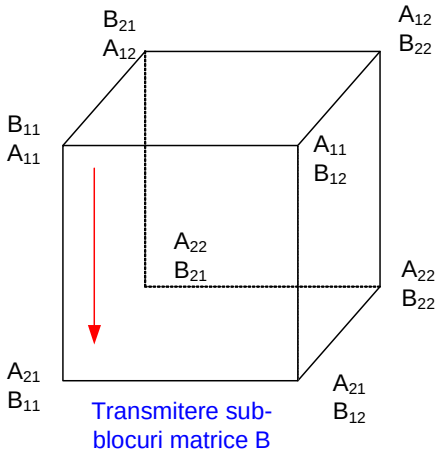
Initializare sub-blocuri matrice A

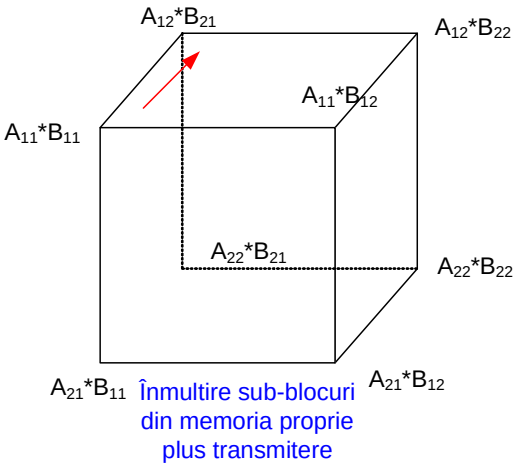


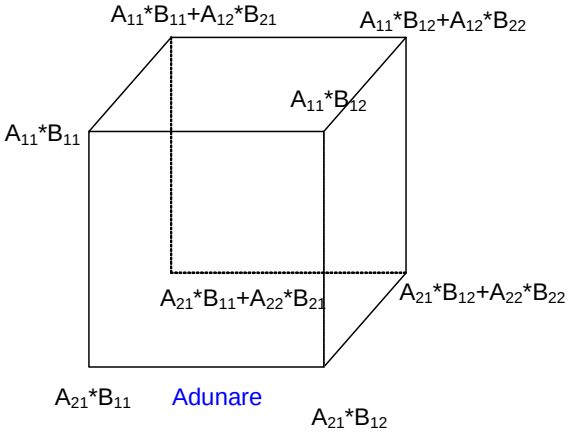
Initializare sub-blocuri matrice B

Faza a doua presupune distribuția acestor sub-blocuri în restul nodurilor din sistem (transmiterea blocurilor matricei A se va face pe orizontală iar a matricei B pe verticală). Având în vedere utilizarea de conexiuni distincte distribuția poate fi făcută simultan.









Partea efectivă de calcul presupune (după finalizarea transmiterii sub-blocurilor celor două matrice) înmulțirea celor două sub-blocuri aflate în memoria proprie (pentru toate nodurile), transmiterea rezultatului (pentru nodurile de pe fața apropiată a cubului) şi adunarea rezultatului propriu la cel primit (pentru nodurile de fața depărtată a cubului). Se observă ca nodurile de pe fața depărtată a cubului (din planul secund) conțin sub-blocurile componente ale matricei rezultat – C.

Exemplul anterior arată o modalitate eficace de implementare a unui algoritm sincron pe un sistem multicalculator. Eficacitatea algoritmului constă în volumul redus de date vehiculat prin intermediul rețelei de interconectare, utilizarea simultană a legăturilor acesteia şi încărcarea uniformă a nodurilor de prelucrare (se observă că operația de înmulțire a sub-blocurilor celor două matrice este distribuită tuturor nodurilor). Exemplul poate fi generalizat pentru o partiționare într-un număr mai mare de sub-blocuri pe o arhitectura de interconectare n-cub.