SPM. 16

Noțiuni de calcul paralel

Partiționare — granularitate < grosieră / medie / fină

→ statică < se face o sg. dată
nu pot ține de dtte. pe care le prelucrează alg.

→ dinamică < la runtime
pot beneficia de un || mai mare de dtte. mai permit

Partiționarea < după dtte. / control (fcțională)

$Z[i] = A[i] * B[i] + C[i] * D[i]$

Pp. că avem 2 procese P1 & P2 → 2 fire ||

P1: $A[i] * B[i]$

P2: $C[i] * D[i]$

Partiționarea de dtte. → dtte. pare le dăm lui P1, & cele impare lui P2.

Partiționarea fcțională: P1: $A[i] * B[i]$, +
P2: $C[i] * D[i]$ ← atît merge la o partiționare statică

De preferat partiționarea de dtte. < oferă n seturi de dtte.
mărcarea lips. e echilibrată
cel mai simplu de implementat

↓ nu e mereu posibil de implementat (nu merem înmulțiri cu vectori & matrice.) !!!

Alocarea → gestionarea firelor de execuție

→ statică → facută înainte lansării în exe.; se face o sg. dată; nu introduce timpi suplimentari de alocare în timpul de execuție, dar nu poate fi modificat

→ dinamică → în mom. exe. se scalează în fcție. de resursele de calcul din mom. rulării → e f. greu de urmărit & introduce un timp suplim. de alocare în mom. rulării

Structuri de algoritmi:

Alg. <
- sincron: toate firele de exe. se sincronizează după fiecare pas;
- asincron: fiecare fir se exe. fără a se sincroniza cu celelalte;
- ppl.: ∃ un fir de exe. care face o prelucrare, dă rez. firului urm., ↳ execuția e spartă pe etape, iar firele de exe. își furnizează rez. intermediare;

ex:
$Z[i] = A[i] * B[i] + C[i] * D[i]$
P1: $A[i] * B[i]$, +
P2: $C[i] * D[i]$

I. ~~alg.~~ alg. sincron:

```
Struct global_memory {
        shared int index;
        shared int A[10], B[10], C[10], D[10], Z[10];
        shared int x;
        shared char turn[6]; // var. de semaforizare
        };

main ()
{
        int y;
        index 1;
        turn = 'slave';  // slave = procesul nou creat
        CREATE (slave); // proc. de inițializare a proc. slave
        while (index <= 10)
            { y = C[index] * D[index]; // procesul părinte
              while (turn = 'slave')  // cât timp = procesul copil nu
                    NOP;              // a terminat, aștept un
              Z[index] = x+y; // în x depune procesul copil A[i]*B[i]
              index = index + 1;
              turn = 'slave';  // aștept evaluarea urm. produs A[i]*B[i]
            }
}
```

```
slave ( )
{   while (index <= 10)
        { while(turn = 'slave')
          {x = A[index] * B[index];
           turn = 'master'; }
        }
}
```

→ Hb. să aștepte ca procesul părinte să îi redea controlul, ptr. că altfel ar fi tot calculat x pt. același index până când master incrementa indexul.

E sincronizat, ptr. că la fiecare iterație se sincronizează master & slave. Produsele se execută în || (A*B cu C*D).

II. Alg. asincron:

Dc. ∃ cel puțin un pct. de sincronizare per întreaga execuție, alg. este sincron !!! ⇒ alg. sincroni sunt ≠ buni dc. mașina conține resurse de calcul aproximativ identice ⇒ încărcarea procesoarelor e.a.m. (vtza. de lucru e cam aceeași).
Alg. asincron ⇒ nu se sincronizează de loc.

```
struct global_memory {
        share int index;
        share int A[10], B[10], C[10], D[10], Z[10];
                };

main ( )
{   CREATE (slave);
    task ( ); // procedură care evaluează Z[i]
    WAIT-END; // procedură ce așteaptă terminarea procesului copil
}

slave ( )
{ task ( );}

task ( )
{   int i;
    NEXT-INDEX (i); // proc. ce face teoretic decrementarea de la 10 la 0.
    while (i>=0)
      { Z[i] = A[i] * B[i] + C[i]*D[i] (?)
        NEXT-INDEX (i);
      }
}
```

→ punem evidență faptul că aceasta este o operație atomică

P1 P2

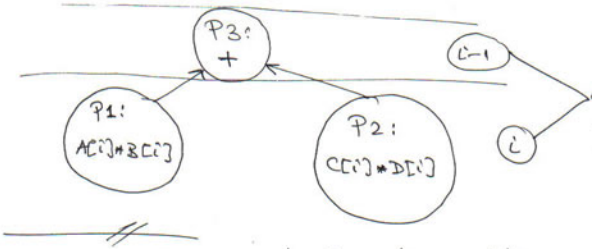
→ SLAVE

→ task → task // intră simultan în task
i=10 i=9

Dc. P2 merge f. încet, el va calcula doar pt. câțiva indecși, restul prel. fiecăruia P1.

Proc. wait_end e cea care asigură încheierea copilului înainte de a se încheia el însuși (ca să nu am procese zombie prin memorie)

III. Alg. Ppl.:



~~Exemple~~ Analiza alg. paraleli

Criterii — costul de timp (-fact. de accelerare)
— eficiența de utilizare a resurselor de calcul
— costul de execuție

(1) Lucru (w = work): nr. op. efectuate în cadrul alg. de calcul ||
→ dependent de nr. de fire și de setul de date de intrare

(2) Factorul de accelerare ⇒ $F = \frac{T_{seral}}{T_{paralel}} = \frac{T_s}{T_p}$

→ utilizarea mașinilor de calcul || pt. taskuri negândite || duce la exe. îngreunată ⇒ dc. progr. secvențial e gândit pt. o mașină secvențială, se poate ca $F > n$ (partea secvențială încetinește mult mai prost decât partea ||) ⇒ în realitate ne interesează ca alg. || să încetinească mai bine pe mașina || decât încetinește alg. secvențial pe mașina secvențială!

3) Eficienta II: $E_p = \frac{T}{p}$, unde p = nr. de procesoare

→ ideal $E_p = 1$ (100%)

→ comparăm factorul de acc. real (T) cu cel ideal (p)

4) Costul: $C_p = p * T_p$

→ măsurare a timpilor de execuție

→ dc. alg. e neechilibrat (timpi dif. de exe. ptr. fiecare μP), $C_p = T_{p1} + T_{p2}$

5) Redundanța: $R_p = \frac{W(p)}{W(1)}$

→ raportul dintre lucrul paralel ($W(p)$ = nr. de op. pe p procesoare) și lucrul secvențial ($W(1)$ = nr. op. pe 1 sg. μP);

→ câte op. inutile fac; (spune)

6) Utilizarea: $U_p = \frac{W(p)}{C_p}$

→ lucru / cost ⇒ dă gradul de încărcare a mașinii (nr. de op. pe timpul total măsurat în respectiva implementare);

7) Calitatea: $Q_p = \frac{T_s^3}{C_p * W(p)} = \frac{T_s^3}{p * T_p * W(p)}$

→ param. global ⇒ corelează câștigul de timp (factorul de acc.) cu resursele ⇒ e bună investiția?

→ a doua fun. e ptr. timp de execuție egal pe toate μP.

Arbore binar:

→ nodul răd. ⇒ dr. < stga. 001

→ mec. de dirijare: ⇒ o să pozitez pe biți ///