Simularea PARALELISMULUI

Emilian-Marius Bold

În cadrul acestui articol vom prezenta o modalitate de simulare a paralelismului. Abordarea este exemplificată pentru simularea unor algoritmi paraleli simpli de sortare.

De multe ori suntem puși în situația de a realiza un program cât mai simplu, care să "pară" că folosește metode sofisticate cum ar fi paralelismul. Probabil că mulți dintre cei care citesc acest articol au încercat, cel puțin odată în perioada de inițiere în programare, să simuleze diferite tehnici superioare (sisteme de ferestre în mod grafic, efectuarea în paralel a mai multor operații etc). Acest lucru nu este inutil - nu încearcă nimeni să reinventeze roata - deoarece se pot învăța multe și din această tentativă. (Autorul recunoaște că și-a explicat mult mai bine lucrul cu pointeri, liste și alte structuri dinamice când a încercat să construiască un sistem de ferestre în mod grafic cu o simulare de *callback* gen *Windows* și *timer*-e.).

Un astfel de program simplu pe care am vrut să-l fac - a se citi trebuia să-l fac - avea scopul de a arăta cu cât este mai puțin bună metoda ***quicksort*** în comparație cu celelalte metode de sortare. Deci, avem mai mulți algoritmi de sortare care primesc același șir (de fapt mai multe șiruri care inițial conțin aceleași numere) și trebuie ca sortarea să se facă sub ochii utilizatorului.

Rularea în paralel a algoritmilor se putea realiza folosind posibilitățile ***multitasking*** ale sistemului de operare folosit, însă scopul declarat a fost de a scrie un program la nivel de liceu care să poată face aceste lucruri. Vom discuta în continuare doar la nivel de pseudocod, deci limbajul de programare în sine nu are nici o importanță.

Posibilitatea *multitasking* fiind eliminată, am urmat ideea ca fiecare procedură (corespunzătoare câte unui algoritm) să fie apelată de un număr de ori până se ajunge la sortarea șirului; la fiecare apel procedura trebuie să ruleze doar un "pas" din algoritm. Această idee poate fi transpusă astfel:

```
[1]
01: cât timp nu s-au sortat toate șirurile execută
02: apelează QuickSort
03: apelează BubbleSort
...
xx: sfârșit cât timp
```

Din nou subliniez faptul că un apel al unei proceduri nu determină sortarea imediată a șirului. La un apel procedura execută un număr de instrucțiuni, urmând ca la următorul apel să execute instrucțiunile care urmează în mod logic.

Încă de la început apare o idee evidentă: pentru ca aceasta să fie o simulare corectă a paralelismului trebuie ca fiecare procedură să aibă ca număr maxim de instrucțiuni o constantă.

Deoarece fiecare procedură are un timp de rulare $T(1)$ și evident și o complexitate $O(1)$ este ca și când s-ar oferi acesteia o cotă dintr-un procesor virtual.

Singurul lucru care deranjează într-o simulare este posibilitatea ca una dintre proceduri să acapareze firul de execuție. Tocmai de aceea instrucțiunile ciclice reprezintă un "pericol" ridicat. Ceea ce dorim să evităm sunt de fapt acele "salturi înapoi" - înțelegem printr-un salt înapoi un salt la o instrucțiune peste care am trecut deja, mai exact saltul la o instrucțiune care are o adresă mai mică decât adresa instrucțiunii curente (presupunând că adresele cresc) - și ne folosim astfel de operații decizionale (dacă). Evident că toată această transformare a operațiilor ciclice în operații decizionale poate să necesite variabile suplimentare.

În practică, procedurile se vor executa destul de repede astfel încât, pentru a putea observa modificările efectuate de algoritm, ar trebui utilizat câte un apel gen ***delay***.

De la prima implementare se observă o altă necesitate: trebuie să existe pentru fiecare algoritm o procedură de inițializare pe lângă cea de simulare deoarece procedura de simulare nu se poate baza pe datele pe care i le transmitem. Evident, codul de inițializare putea fi inclus în procedura

de simulare, urmând ca noi să transmitem un parametru special care să indice dacă este sau nu primul apel al procedurii.

Să luăm spre exemplu următorul algoritm de sortare:

```
[2]
1: repetă
2:    sch=fals
3:    pentru i=1, n-1 execută
4:       dacă a[i]>a[i+1] atunci
5:          sch=adevărat
6:          apelează Schimbă a[i],a[i+1]
7:       sfârșit dacă
8:    sfârșit pentru
9: până când not sch
```

Avem de-a face cu o variantă a algoritmului *Bubblesort*. Pentru a avea, după cum am mai spus, un număr fix de instrucțiuni executate la un apel al procedurii trebuie să simulăm instrucțiunile ciclice și eventualele apeluri recursive.

Procedura START se deduce din pseudocodul algoritmului, dacă ne amintim că instrucțiunea **pentru** este echivalentă cu:

```
[3]
1: i=1
2: dacă i<=n atunci
3:    execută corpul instrucțiunii "pentru"
4:    i=i+1
5:    salt la linia 2
6: sfârșit dacă
```

Structurile repetitive sunt de fapt înlocuite de structuri alternative, eventual adăugând variabile suplimentare - sună cunoscut?

Astfel, o implementare ar putea fi:

```
[4]
START:
1: sch=fals
2: i=1
BSORT:
01: dacă i<n atunci
02:    dacă a[i]>a[i+1] atunci
03:       sch=adevărat
04:       apelează Schimbă a[i],a[i+1]
05:    sfârșit dacă
06: sfârșit dacă
07: i=i+1
08: dacă i>=n atunci
09:    dacă sch atunci
10:     sch=fals
11:     i=1
12:    altfel
13:     încheiere algoritm - sortat!
14:    sfârșit dacă
15: sfârșit dacă
```

În procedura BSORT presupunem că suntem deja în interiorul ciclului **repetă**. La linia 1 se verifică dacă suntem în interiorul ciclului **pentru**, iar dacă da, urmează instrucțiunile care le observăm și în liniile 4-7 [2].

După ce mărim *i* la linia 7 putem avea două cazuri: *i* este mai mic decât *n* și, deci, încă nu am ieșit din ciclul **pentru** sau este egal cu *n*, ceea ce înseamnă că trebuie să vedem dacă vom ieși și din **repetă**. Cazul în care *i* este mai mare decât *n* apare doar când *n* este mai mic decât 2; este un caz particular de mic interes.

O dată încheiat ciclul **pentru** (linia 8) verificăm dacă au fost efectuate schimbări sau nu. Dacă au fost, ne pregătim pentru un nou ciclu **pentru**; observați asemănarea dintre liniile 10-11 și liniile 1-2 din START. În momentul în care ajungem la linia 13 înseamnă că nu au fost efectuate schimbări, șirul este ordonat și putem declara că algoritmul de sortare s-a încheiat. Această linie nu este echivalentă cu o simplă ieșire din procedură - în practică am folosit o variabilă suplimentară pentru a determina dacă șirul este sortat sau nu; variabila devenea fals în START și adevărat abia în acest moment.

Ar fi un exercițiu interesant pentru cititori realizarea în același mod al algoritmului SelectSort.

Algoritmul prezentat a fost destul de ușor, deoarece nu apare recurență în algoritmul inițial. În continuare vom observa că recurența modifică în mare masură modul de simulare.

Vom încerca algoritmul Quicksort. Varianta sa în pseudocod este:

```
[5]
QUICKSORT(primul, ultimul)
01: i=primul
02: j=ultimul
03: temp=a[(i+1)/2]
04: repetă
05:    cât timp a[i]<temp execută
06:       i=i+1
07:    sfârșit cât timp
08:    cât timp a[i]>temp execută
09:       j=j-1
10:    sfârșit cât timp
11:    dacă i<j atunci
12:       apelează Schimbă a[i],a[j]
13:    sfârșit dacă
14:    dacă i<=j atunci
15:       j=j-1
16:       i=i+1
17:    sfârșit dacă
18: până când i>=j
19: dacă primul<j atunci
20:    apelează QUICKSORT primul,j
21: sfârșit dacă
22: dacă i<ultimul atunci
23:    apelează QUICKSORT i,ultimul
24: sfârșit dacă
```

În continuare va trebui să simulăm stiva, iar procedura va lucra mereu cu elementele din vârful acesteia. Altfel spus, variabilele `i`, `j` etc. vor fi variabile din vârful stivei simulate. Structura stivei va include `primul`, `ultimul`, `i`, `j`, `temp`, `pas2`. Procedura de inițializare devine:

```
[6]
START
0: A=vidă
1: A<=(1,n,0)
```

Prin `A<=(X,Y,Z)` înțelegem aici introducerea în stiva `A` a unui bloc care are `primul=X`, `ultimul=Y` și `pas2=Z` iar celelalte variabile sunt nedefinite (deocamdată). În același mod se definește `x<=A` ca reprezentând scoaterea unui nod din `A` și depunerea în `x`.

Elementele nedefinite ca urmare a procedurii de inițializare vor fi definite de către procedura de simulare în sine. Elementul `pas2` este o variabilă suplimentară care devine 1 doar atunci când s-a realizat definirea elementelor despre care vorbim. Variabila `temp` nu putea fi inițializată în `START`, deoarece se presupune acces la șirul `a` și credem că este mai firesc ca procedura de simulare să fie singura cu acces la acest șir. Această procedură devine:

```
[7]
QSORT
01: dacă pas2=0 atunci
02:    i=primul
03:    j=ultimul
04:    temp=a[(i+j)/2]
05:    pas2=1
06: sfârșit dacă
07: dacă a[i]<temp atunci
08:    i=i+1
09: sfârșit dacă
10: dacă a[i]>temp  atunci
11:    j=j-1
12: sfârșit dacă
13: dacă (a[i]<temp) sau (a[i]>temp) atunci
14:    ieșire QSORT
15: sfârșit dacă
16: dacă i<j atunci
17:    apelează schimbă a[i], a[j]
18: sfârșit dacă
19: dacă i<=j atunci
20:    j=j-1
21:    i=i+1
22: sfârșit dacă
23: dacă i>=j atunci
24:    k=0
25:    dacă primul<j atunci
26:       A<=(primul,j,0)
27:       primul=j;
28:       k=1
29:    sfârșit dacă
30:    dacă i<ultimul atunci
31:       A<=(i,ultimul,0)
32:       ultimul=i;
33:       k=1
34:    sfârșit dacă
35:    dacă k=0 atunci
36:       x <= A
37:       dacă A=vidă atunci
38:          încheiere algoritm - sortat!
39:       sfârșit dacă
40:    sfârșit dacă
41: sfârșit dacă
```

Este foarte important faptul că toate variabilele din algoritmul de mai sus, cu excepția lui `x` și `k`, sunt de fapt cele din vârful stivei. Mai mult, `x` nici nu trebuie să fie o variabilă propriu-zisă - este suficientă ștergerea nodului din vârful stivei (cu actualizările de rigoare).

La linia 1 se testează dacă au fost inițializate toate variabilele.

Liniile 7-9, 10-12 corespund secvențelor **`cât timp`** din algoritmul original, doar că se execută câte un singur pas.

Liniile 13-15 verifică dacă putem trece la instrucțiunea de la linia 11 [5]. Pentru a respecta complexitatea $O(1)$ a versiunii pseudo-paralele, ciclurile **`cât timp`** nu se pot reproduce exact și de aceea apelăm la aceasta metodă. Observați, totuși, că la un apel al procedurii se pot executa atât linia 8 cât și 11 - dorim ca la un apel să se poată executa cât mai multe instrucțiuni.

Liniile 16-21 sunt aceleași cu 11-17 din `[5]`, iar linia 22 verifică dacă am ajuns la sfârșitul ciclului **`repetă`** din `[5]`.

Liniile 25-29, 30-34 corespund liniilor 19-21, 22-24 din `[5]`. Deoarece simulăm stiva, **`apelează`** `QSORT x, y` devine `A<=(x,y,0)`. Liniile 28 și 33 au un rol aparte: `k=1` marchează faptul că s-au introdus elemente în stivă, iar prin operațiile `primul=j` și `ultimul=i` ne asigurăm că la o viitoare apelare a procedurii pentru acest nod al stivei nu vom mai întroduce din nou alte noduri.

Linia 35 testează dacă `k=0`. Aceasta înseamnă că nu s-au introdus noduri, deci putem șterge din stivă fără prea mari probleme. Sortarea se încheie când `A` este vidă.

Se poate observa că în cazul unui bloc din stivă avem două cazuri:

- dacă acesta nu determină crearea altor noduri, el se va șterge în același moment în care vom ajunge la pasul 16;
- dacă acesta determină crearea altor blocuri în stivă el se va șterge în urma unui apel ulterior al procedurii, după ce toate blocurile care se vor crea după ce a fost creat el se vor șterge. Acest mod de a șterge blocul în urma unei apelări practic vide a procedurii (căci ultima apelare nu face altceva decat să șteargă nodul) poate fi înlocuit cu o altă metodă de a crea noi blocuri: unul din cele două posibile noi blocuri ar putea înlocui conținutul blocului curent. Se poate ajunge astfel și la o utilizare redusă a stivei. (Din păcate în program nu am folosit această metodă deși pare foarte bună!).

Emilian-Marius Bold este student la Universitatea de Vest din Timișoara. Poate fi contactat prin e-mail la fierarul@info.uvt.ro .