Căutarea paralelă în tablouri masive de numere folosind sisteme distribuite

Ovidiu Gherman

***Abstract*— The searching algorithms are among the most used in data processing. In the case of massive data structures, we can try to use distributed systems (a grid) to increase the performance of the applications. In this regard, we can elaborate a searching application and run it on a grid test bed, so we can compare its runtime with that of a mono-processor sequential application; in this way we can determine if such distributed implementation is adequate to this category of algorithms.**

***Index Terms*— Algorithm, data structures, grid, parallel search**

I. Introducere

Încă dinainte de a apare calculatorul, operaţia de căutare reprezenta un proces esenţial de prelucrare a datelor. Aceasta era (şi încă este) utilizată frecvent atât în viaţa de zi cu zi (căutarea unui număr de telefon într-o agendă, de exemplu) cât şi în procesarea datelor matematice sau ştiinţifice. În cazul unei cantităţi mici de date (sau a tipurilor de date simple), acest lucru se poate face manual. Dar atunci când vorbim de cantităţi mari de date (sau date mai complexe, de genul şirurilor lungi de caractere sau a numerelor mari), mai ales când acestea nu sunt sortate iniţial, o abordare automată a căutării este de dorit [1].

Algoritmii folosiţi pentru procesul de căutare pot varia în funcţie de rapiditatea de execuţie, simplitatea implementării, etc. Dar atunci când avem la dispoziţie un număr mare de date, posibil nesortate, este de dorit ca respectivul algoritm să ofere rezultatele cât mai repede. Abordarea clasică, secvenţială, presupune folosirea unui singur procesor pe care să ruleze programul. Dar, în contextul în care tehnica de calcul evoluează spre folosirea unor sisteme multiprocesor, se pot realiza programe care să beneficieze de această nouă abordare.

II. Calculul paralel pe sisteme grid

Grid computing este noua paradigmă în procesarea datelor. Utilizarea mai multor sisteme de calcul pentru rezolvarea unei probleme complexe poate duce la scurtarea timpului de procesare, care poate fi critic în cazul unor aplicaţii speciale sau foarte mari. Dar în timp ce sistemele multiprocesor (de exemplu clustere de calculatoare) sunt relativ scumpe, a apărut o altă soluţie ce poate fi implementată folosind sisteme de calcul de uz general, eterogene, disponibile în număr mare: platformele de calcul de tip grid.

Există mai multe protocoale pentru implementarea aplicaţiilor de calcul distribuit, dar unul din cele mai folosite este MPI (*Message Passing Interface*). Avantajul acestor sisteme de calcul distribuit comparativ cu sistemele tip cluster îl reprezintă faptul că un sistem grid poate fi creat folosind, de exemplu, PC-uri de uz general, ieftine. Acestea pot fi adăugate în număr foarte mare, folosite cu costuri minime (de exemplu atunci când proprietarul real al echipamentului nu îl foloseşte), prin instalarea unei platforme MPI; aplicaţia distribuită va fi rulată iniţial de pe o staţie de lucru, dar va folosi toate resursele disponibile în grid la momentul rulării [2].

Totuşi, un punct nevralgic în ceea ce priveşte folosirea sistemelor distribuite îl reprezintă comunicaţia între procese. Dat fiind că nu mai beneficiem de magistralele de comunicaţie dedicate (cum ar fi în cazul sistemelor cluster, de exemplu), în cazul sistemelor grid eterogene trebuie să folosim reţele de comunicaţii standard, implementate pentru uzul curent al sistemelor PC, cum ar fi Ethernet. Astfel putem spune că această arhitectură este slab cuplată [3].

Un exemplu de sistem grid este cel pe care au fost rulate aplicaţiile de test realizate pe baza algoritmilor detaliaţi mai jos. Acest sistem constă din 6 unităţi tip PC, cu specificaţii modeste, conectate printr-o reţea Gigabit Ethernet (se încearcă evitarea gâtuirilor la transferul de cantităţi mari de date prin folosirea unor medii de comunicare cât mai rapide, ţinând cont că acesta poate fi un posibil punct sensibil al acestui tip de sistem). Sistemul comunică cu exteriorul (practic, cu terminalul de comandă prin care un utilizator poate da comenzi sistemului) printr-o reţea 10/100 Ethernet. Sistemul de operare folosit este Scientific Linux, cu o implementarea OpenMPI (v 2.0) a protocolului MPI [4].

III. Căutarea în tablouri masive de date

Atunci când vorbim de căutarea în structuri de date mari, neordonate, algoritmul de căutare secvenţială este unul din cele mai folosite. Putem presupune că împărţirea problemei în sub-probleme este o modalitate de a creşte performanţa

O. Gherman este doctorand în domeniul Calculatoare şi Tehnologia Informaţiei în cadrul Facultăţii de Inginerie Electrică şi Ştiinţa Calculatoarelor al Universităţii "Ştefan cel Mare" Suceava (e-mail: ovidiuh@stud.usv.ro)

aplicaţiei prin execuţia diferitelor părţi ale programului (numite procese) pe procesoare distincte, dar pentru aceasta trebuie să vedem dacă nu cumva un sistem monoprocesor nu poate realiza acelaşi lucru mai rapid, fără a fi penalizat la viteza execuţiei prin folosirea unor sisteme de comunicaţii lente (comparativ cu viteza de procesare a unităţii centrale) între procesoare - în cazul unui sistem grid, între PC-uri.

În cazul algoritmului de căutare secvenţială, considerăm o valoare de căutat, numită "target" (de exemplu o valoare integer). Datele de intrare se citesc dintr-un fişier formatat corespunzător. Fiecare număr distinct este citit, comparat cu valoarea target; în cazul în care cele două valori sunt identice, incrementăm contorul şi marcăm poziţia în care am descoperit acea valoare. La final, datele de ieşire (numărul de valori găsite, reprezentat prin contor, precum şi poziţiile acestora) sunt afişate pe consolă şi într-un fişier de ieşire.

Există posibilitatea de ordonare a tabloului de elemente, dar în cazul unor tablouri masive (de sute de mii sau chiar milioane de elemente), acest lucru poate deveni un impediment, sporind complexitatea calculului. În acelaşi timp, trebuie să luăm în considerare şi posibilitatea ca alimentarea cu date să se facă în mod continuu, de exemplu de la un proces al sistemului grid (fiind o structura eterogenă, un sistem grid poate include şi alte dispozitive decât PC-uri, de exemplu un aparat de măsură ce poate furniza în mod continuu date pentru procesare într-un buffer).

Paralelizarea acestui algoritm îi creşte complexitatea, pentru că implică descompunerea tabloului de elemente (dată de intrare) în sub-tablouri, care vor fi procesate separat. Rezultatul prelucrării datelor va fi întors către procesul ce a iniţiat algoritmul, pentru a fi afişat. Această tehnică de implementare paralelă este cunoscută sub numele de "replicated workers" sau "master-worker" şi presupune un proces *master* (sau *root*, în mod tradiţional procesul cu rangul 0) care iniţiază calculele, descompune problema şi o dă spre prelucrare unor procese *worker* identice; fiecare worker preia sarcina, o execută şi returnează către master rezultatul[1] [5].

Algoritmul paralel implementat pe sistemul distribuit poate avea structura din Fig. 1:

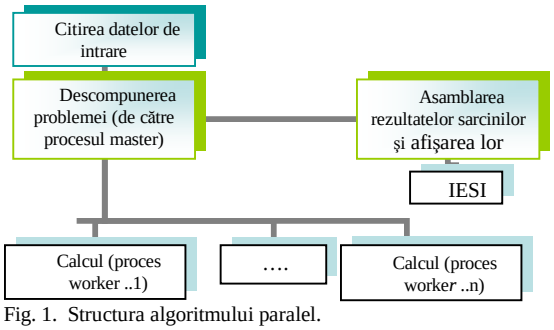

Fig. 1. Structura algoritmului paralel.

Programul aplicaţiei va fi scris identic pentru fiecare nod (platforma MPI replicându-l corespunzător pe toate nodurile), de aceea tratarea sarcinilor se va face în funcţie de rangul procesului care rulează programul. Dacă rangul procesului curent este 0, îl vom considera ca fiind procesul master (şi va executa sarcinile atribuite lui), iar în cazul oricărui alt rang, vom considera că procesul este un worker, şi va îndeplini sarcina computaţională (toate procesele worker au acelaşi statut, iar încărcarea proceselor trebuie să fie cât mai egală, pentru a se folosi în mod optim resursele („load-balanced"). Astfel se execută fie o ramură a programului, fie alta, dar niciodată ambele pe un procesor dat.

Implementarea acestui program se poate face conform structurii de mai jos:

- declararea variabilelor;
- iniţierea mediului MPI;
- extragerea rangului procesului (pentru a realiza discriminarea master-worker) şi a nr. de procese;
- dacă (master) atunci
  - se citesc datele de intrare (valoare target, lungimea tabloului);
  - calcularea *pasului* cu care se va împărţi tabloul masiv în sub-tablouri;
  - calcularea cu o formulă generală a *limitelor* sub-tabloului cu referire la tabloul masiv;
  - difuzarea datelor către procesele worker (valoarea target, lungimea sub-tabloului);
  - se aşteaptă ca rezultatele de la procesele worker (numărul de hit-uri şi tabloul parţial cu poziţia acestora) să sosească;
  - după primirea tuturor rezultatelor, se recompune tabloul poziţiilor şi se calculează numărul total de hit-uri;
  - afişare rezultat (consolă, fişier text);
- altfel
  - procesul worker curent primeşte sarcina;
  - se caută numărul de coincidenţe între valoarea target şi elementul curent din şir;
  - se trimit rezultatele către procesul master;
- se calculează timpul de execuţie;
- se părăseşte mediul MPI.

În cadrul algoritmului de mai sus, putem defini următoarele valori:

lungime: numărul de elemente din tabloul de intrare;

pas: pasul cu care va fi împărţit tabloul de intrare în sarcini alocabile proceselor worker;

proc: numărul de procesoare alocat aplicaţiei;

k: rangul procesului curent (doar pentru procese tip worker);

x1, x2: limitele inferioare şi superioare ale sub-tablourilor procesate de worker.

În aceste condiţii putem defini valorile folosite în algoritm:

[1] Putem aloca în mediul OpenMPI un număr de procese egal sau mai mare cu cel fizic disponibil, caz în care fiecare worker va primi, pe rând, mai multe sarcini spre rezolvat, până la epuizarea acestora.

$$pas = lungime/(proc-1)+1 \quad (1)$$

şi

$$x1 = (k-1)* pas + 1 \quad (2)$$

$$x2 = k * pas \quad (3a)$$

cu excepţia cazului $x2 > lungime$, în care:

$$x2 = lungime \quad (3b)$$

Se încearcă realizarea unor sub-tablouri relativ egale ca lungime, astfel încât să nu existe unele procese care să lucreze excesiv, în timp ce altele sunt in stare de repaus (au terminat deja procesarea). Se poate observa că sarcinile sunt cu atât mai echilibrate cu cât valorile *lungime* şi *proc* sunt mai mari (în cazul în care cele două valori sunt extreme şi opuse, se ajunge la un dezechilibru a sarcinilor alocate proceselor worker) [6].

Putem observa că această implementare are o *granularitate* mare, cu comunicări rare (fiecare proces worker primeşte şi trimite doar câte un set de date).

IV. TESTE COMPARATIVE

În continuare sunt prezentate rezultatele unor teste comparative între versiunea monoprocesor (serială) a algoritmului şi cea distribuită (în al doilea caz se poate verifica eficienţa algoritmului în funcţie de numărul de procesoare disponibile 2, 3, 4, …):

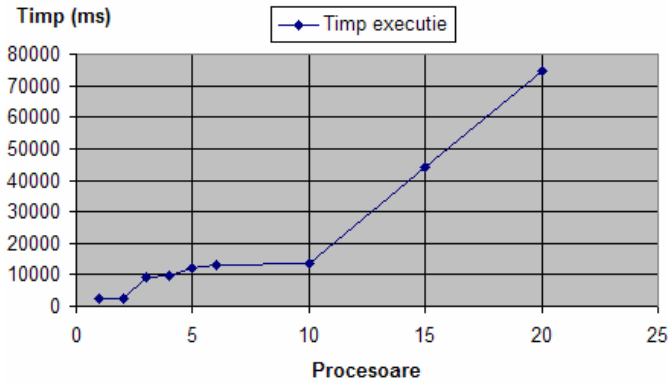

Fig. 2. Durata de execuţie a programului pe grid (pentru tablouri de 1.000 de elemente).

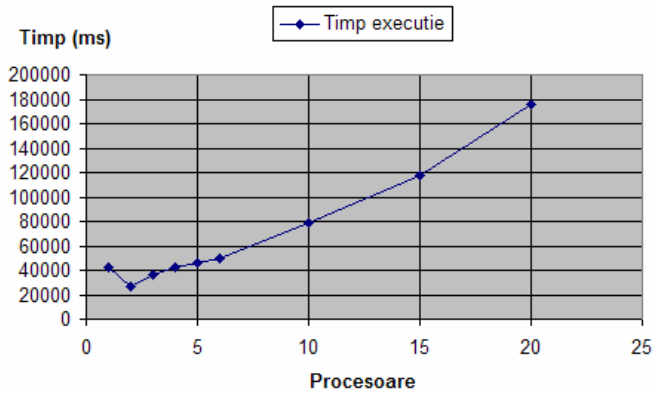

Fig. 3. Durata de execuţie a programului pe grid (tablou de 100.000 elemente).

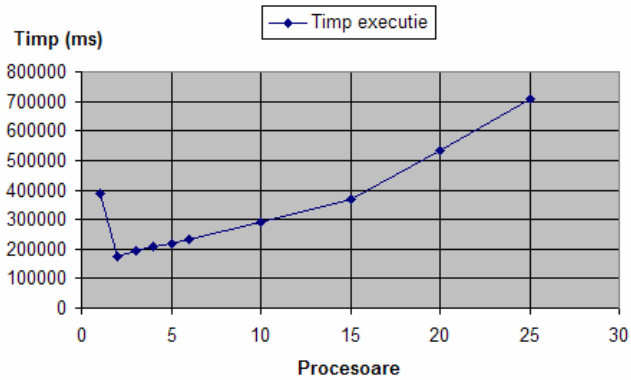

Fig. 4. Durata de execuţie a programului pe grid (tablou de 990.000 elemente).

În cazul în care alocăm mai multe procesoare decât cele disponibile fizic, procesele worker vor primi mai multe seturi de date pe care le vor prelucra în ordinea primirii; chiar dacă aceste seturi vor fi mai mici ca şi dimensiuni (datorită segmentării mai puternice produse de numărul mai mare de procese alocate), se va pierde mai mult timp cu iniţiere transmisiei şi cu recepţia datelor (sa nu uităm că vitezele de transmisie sunt mai mari decât cele de lucru ale procesorului). Astfel, se va constata o creştere a timpului total de lucru, ca în Fig. 5:

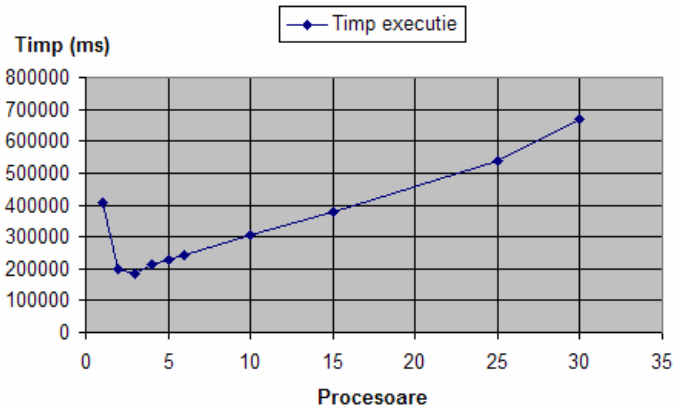

Fig. 5. Durata de execuţie a programului (pe un tablou de 1.035.000 elemente) creşte la alocarea unui număr de procese mai mare decât numărul disponibil fizic (gridul de test conţine 6 unităţi de procesare independente).

V. CONCLUZII

Putem calcula accelerarea reală (Sp), definită ca fiind raportul între timpul de execuţie al celui mai bun program serial pe un procesor al sistemului paralel şi timpul de execuţie al programului paralel pentru p procesoare [3]. În cazul ideal (în care presupunem că nu există timpi de comunicare şi sincronizare), Sp ar trebui să fie egal cu p. Vom obţine următoarele valori ale accelerării reale (în funcţie de numărul de elemente al tabloului):

TABEL I

Valorile S_p calculate pentru diferite configuraţii.

	1000	**990000**	**1035000**
S_2	1,0048	2,2085	2,0442
S_3	0,2651	2,0344	2,1925

S_4	0,2546	1,8887	1,9301
S_5	0,2056	1,7821	1,7918
S_6	0,1893	1,6602	1,6949
S_{10}	0,1831	1,3342	1,3433
S_{15}	0,0567	1,0592	1,0759
S_{25}	-	0,5500	0,7571
S_{30}	-	-	0,6090

După cum se observă din graficul de mai jos, pentru un anumit număr de elemente există un număr optim de procese worker care să prelucreze sub-tablourile (mai mare înseamnă mai bun):

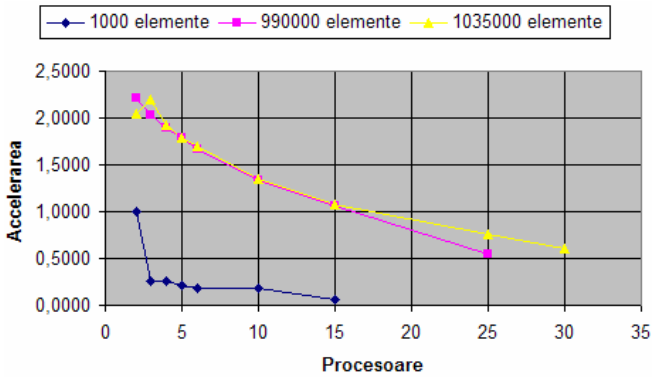

Fig. 6. Diferența dintre viteza algoritmului serial şi cea a algoritmului paralel se manifestă mai ales în cazul tablourilor masive de date (pentru valori mici, algoritmul serial este mai simplu de implementat).

Un număr prea ridicat de procese duce la creşterea timpului de execuţie a aplicaţiei deoarece o mare parte a timpului este pierdută cu transferul datelor între procese şi sincronizări (de ex. fenomenul de overhead) şi nu cu partea computaţională efectivă [6]. Oricum, este de dorit o valoare cât mai mare a accelerării (maximul graficului va determina numărul optim de procese). Trebuie să ne ferim de cazul în care aceasta devine subunitară, deoarece în acest caz algoritmul serial este mai eficient. După cum se constată din Fig. 6, putem spori performanţa sistemului distribuit în cazul folosirii unor cantităţi de date mari, cât mai localizate la nivelul proceselor worker, astfel încât să se exploateze în primul rând resursele de calcul ale nodurilor, şi să se minimizeze cantitatea de date transferate. Dacă este posibil, se pot folosi tehnici de optimizare pentru căutare (de exemplu realizarea unor tabele hash la achiziţia datelor, sau indexarea acestora), ceea ce poate duce la o scădere suplimentară a timpului de execuţie pe sistemul grid.

REFERINȚE

[1] B. Jacob, M. Brown, K. Fukui, N. Trivedi, "Introduction to Grid", publicaţie electronică Redbooks IBM, decembrie 2005.

[2] I. Foster, C. Kesselman, "Computational Grids", capitol al "The Grid: Blueprint for a New Computing Infrastructure", 1999.

[3] D. Grigoras, „Calcul paralel – de la sisteme la programarea aplicatiilor", editura Computer Libris Agora, Cluj, 2000.

[4] Open MPI: Open Source High Performance Computing (disponibil online noiembrie 2008 la http://www.open-mpi.org/projects/user-docs/).

[5] V. Cristea, „Algoritmi paraleli – note de curs" (versiune electronica disponibila octombrie 2008).

[6] „Introduction to MPI „ (curs CI-Tutore a NCSA Illinois, SUA, disponibil online noiembrie 2008 la http://ci-tutor.ncsa.uiuc.edu).