Parallel Sorting Algorithms

Asist. Felician ALECU
Catedra de Informatică Economică, A.S.E. Bucureşti

One of the fundamental problems of computer science is ordering a list of items. There are a lot of solutions for this problem, known as sorting algorithms. Some sorting algorithms are simple and intuitive, such as the bubble sort, but others, like quick sort, are extremely complicated but produce lightening-fast results.
Keywords: *parallel processing, performance analysis, parallel efficiency, sorting algorithm, parallelizing.*

Sortarea, ce se constituie ca una dintre cele mai importante operații efectuate în cadrul aplicațiilor informatice, fie ele ştiinţifice sau comerciale, reprezintă o metoda prin care elementele unei mulțimi sunt aranjate într-o anumită ordine, numerică sau lexicografică, pe baza valorilor unor chei. Sortarea este una dintre cele mai răspândite operații care se execută pe calculatoarele de astăzi iar de eficiența ei depind performanțele altor algoritmi care operează asupra unor liste ce trebuie să fie în prealabil ordonate, cum ar fi cei de căutare şi cei de interclasare.

Cu toate că există numeroşi algoritmi de sortare, nu se poate afirma că unul dintre ei ar fi cel mai bun. Alegerea algoritmului potrivit pentru o aplicație dată trebuie să țină seama de numărul de elemente ce trebuie sortate, de complexitatea operațiilor de interschimbarea a valorilor a două înregistrări şi de cât de neordonate sunt elementele listei.

Dacă presupunem că fiecare algoritm va opera asupra unui vector de întregi ce se doreşte a fi ordonat crescător în funcție de valorile elementelor atunci operația de interschimbarea a două elemente ale vectorului va avea o complexitate de ordinul *O(1)*. Pentru exprimarea indicatorilor de performanță ne vom raporta la timpul de execuție al celui mai rapid algoritm secvențial de sortare care este de ordinul *O(n log n)*.

În continuare vor fi trecuți în revistă cei mai cunoscuți algoritmi de sortare împreună cu indicatorii de eficiență asociați.

Sortarea prin metoda bulelor (Bubble Sort)

Cu toate că sortarea prin metoda bulelor reprezintă una dintre cele mai cunoscute şi mai utilizate metode de ordonare, eficiența acesteia este una foarte scăzută. Astfel, algoritmul secvențial de sortare prin metoda bulelor are o complexitate de ordinul $O(n^2)$ datorită faptului că sunt comparate perechi adiacente ce sunt interschimbate în cazul în care criteriul de ordonare nu este îndeplinit. După o parcurgere integrală a elementelor vectorului, procesul se reia începând din nou cu primul element. Algoritmul mai este cunoscut sub denumirea de metoda bulelor datorită faptului că, la fiecare pas, elementele sunt deplasate către poziția corectă în funcție de condiția de sortare. După fiecare iterație, o parte dintre elementele vectorului (începând de la cel de-al doilea din ultima pereche modificată) sunt deja ordonate. Algoritmul îşi încheie execuția după un număr de maxim *n–1* iterații, atunci când, la o nouă parcurgere, nu s-a mai produs nici o interschimbare.

Datorită faptului că sunt comparate în ordine perechi de elemente adiacente, algoritmul de sortare prin metoda bulelor nu este foarte uşor de paralelizat. Pentru obținerea variantei paralele se foloseşte metoda numită *transpoziție par-impar*. Astfel, fiecare iterație este realizată în două faze. În prima dintre ele, denumită *fază impară*, elementele cu indici impari sunt comparate cu vecinii din dreapta şi se efectuează interschimbarea acestora în cazul în care nu este îndeplinită relația de ordine. În cea de-a doua fază, numită *faza pară*, elementele cu indici pari se compară cu vecinii din dreapta, efectuându-se interschimbarea dacă este nevoie. Ordonarea vectorului este completă după un număr de maxim *n* fa-

ze (*n/2* iteraţii), unde *n* reprezintă numărul de elemente ale vectorului. Iteraţiile buclelor interioare vor fi executate simultan (în paralel) pe procesoarele disponibile în cadrul sistemului.

Complexitatea secvenţială a algoritmului de sortare prin metoda bulelor este de ordinul $O(n^2)$, iar cea paralelă se reduce la $O(n)$.

Sortarea prin metoda selecţiei (Selection Sort)

Sortarea prin metoda selecţiei presupune determinarea elementului minim dintre cele nesortate şi aducerea lui pe poziţia corectă. Iniţial, algoritmul va determina elementul minim din vector care va fi adus pe prima poziţie, după care se determină minimul din vectorul rămas care va fi adus pe a doua poziţie, şamd.

Complexitatea secvenţială a algoritmului este tot de ordinul $O(n^2)$ însă performanţele metodei sunt cu 60% mai bune faţă de sortarea prin metoda bulelor. Indiferent de ordinea iniţială a elementelor vectorului, algoritmul va efectua un număr de *n(n-1)/2* comparaţii şi *n-1* interschimbări iar din acest motiv ordinul de complexitate al variantei secvenţiale este egal cu $O(n^2)$.

Paralelizarea algoritmului se realizează prin distribuirea iteraţiilor buclei interioare pe procesoarele disponibile în cadrul sistemului. Datorită faptului că iteraţiile interioare vor fi executate simultan, ordinul de complexitate al variantei paralele a algoritmului devine $O(n)$.

Sortarea prin metoda inserţiei (Insertion Sort)

Aşa cum îi spune şi numele, sortarea prin metoda inserţiei nu face altceva decât să insereze fiecare element al vectorului în poziţia în care valoarea acestuia se încadrează conform criteriului de ordonare. Cea mai simplă implementare a algoritmului necesită existenţa a două liste – vectorul iniţial şi cel în care sunt scrise elementele ordonate conform criteriului de sortare. Pentru a economisi memorie, cele mai multe implementări, după ce au inserat un element, glisează toate valorile ce urmează cu o poziţie la dreapta. Se reduce astfel numărul de interschimbări deoarece dacă un element este mai mare ca precedentul, atunci acesta se consideră a fi deja ordonat.

Deşi ordinul de complexitate al variantei secvenţiale a algoritmului este tot $O(n^2)$, sortarea prin inserţie este de peste 2 ori mai eficientă decât algoritmul de sortare prin metoda bulelor.

Pentru obţinerea variantei paralele a algoritmului, bucla interioară va fi modificată astfel încât să realizeze doar calculul poziţiei la care trebuie să se facă inserarea, fără deplasarea corespunzătoare a elementelor vectorului. Această din urmă operaţie va apărea individualizată în cadrul unei noi bucle interioare care va fi executată în paralel.

Datorită faptului că mai multe procesoare vor dori să modifice simultan valorile variabilei care reţine poziţia la care se va efectua inserarea elementului curent şi a celei ce semnalizează faptul că poziţia la care se va insera elementul curent a fost deja calculată, actualizarea acestora trebuie inclusă în cadrul unor secţiuni critice care realizează serializarea cererilor de acces concurent cu ajutorului mecanismului de excludere mutuală.

Ordinul de complexitate al algoritmului paralel de sortare prin metoda inserţiei este egal cu $O(n)$, net superior faţă de cel al celui mai rapid algoritm de sortare secvenţial care este $O(n \log n)$.

Shell Sort

Creat de Donald Shell în anul 1959, Shell Sort reprezintă unul dintre cei mai vechi algoritmi de sortare. Cunoscut şi sub numele de *Comb Sort*, este cel mai complex şi cel mai eficient algoritm din clasa celor de ordinul $O(n^2)$.

Deşi este un algoritm rapid, uşor de înţeles şi de interpretat, analiza complexităţii acestuia este o sarcină foarte complicată deoarece se poate deduce în mod intuitiv modul în care algoritmul funcţionează însă analiza formală a timpului de execuţie este o problemă încă nerezolvată.

Algoritmul realizează parcurgeri multiple ale vectorului iar la fiecare iteraţie ordonează seturi de date de dimensiuni egale folosind sortarea prin metoda inserţiei.

Dimensiunea seturilor de date creşte de la o iteraţie la alta până în momentul în care de-

vine egală cu numărul de elemente ale vectorului de sortat. Elementele conținute în fiecare set nu sunt contigue. Astfel, dacă la pasul curent sunt prelucrate *i* seturi, atunci setul j va fi format din elementele aflate în vector pe pozițiile *j*, *j+i*, *j+2i*, şamd. Dacă *i* este egal cu 3, setul al 2-lea, de exemplu, va fi format din elementele aflate în vector la pozițiile 2, 5, 8, etc. Acest comportament este echivalent cu dispunerea elementelor într-un vector bidimensional urmată de ordonarea acestuia pe coloane. La fiecare pas, numărul de coloane scade astfel încât în final se va obține o singură coloană care conține elementele vectorului aproape ordonate.

Shell Sort reprezintă de fapt o extensie a algoritmului de sortare prin metoda inserției. Astfel, *Insertion Sort* este un algoritm foarte eficient în condițiile în care datele de intrare sunt aproape sortate însă, per ansamblu, eficiența acestuia scade datorită faptului că elementele sunt deplasate doar cu câte o poziție. *Shell Sort* este un algoritm foarte similar cu *Insertion Sort* dar care foloseşte paşi mai mari ca unitatea pentru rearanjarea valorilor vectorului. Dimensiunea pasului scade de la o iterație la alta astfel încât în final algoritmul va efectua o sortare clasică prin inserție dar aplicată asupra unor valori ce sunt aproape ordonate. Astfel, dacă o valoare mică este poziționată către sfârşitul vectorului, algoritmul de sortare prin inserție va avea nevoie de aproximativ *n* comparații şi interschimbări pentru a muta elementul respectiv pe poziția corectă. Folosind însă *Shell Sort*, datorită faptului că algoritmul foloseşte paşi de dimensiune mai mare, valoarea respectivă va fi adusă pe poziția corespunzătoare într-un număr mult mai mic de paşi.

Corectitudinea algoritmului este garantată de faptul că la ultimul pas se realizează o sortare clasică a întregului vector prin metoda inserției. Dar datorită faptului că datele sunt deja parțial ordonate, sortarea prin metoda inserției va avea nevoie de doar câteva iterații pentru a finaliza ordonarea elementelor vectorului. Numărul acestor iterații depinde de modul în care se aleg dimensiunile paşilor de calcul. Dimensiunea seturilor de date folosite la fiecare pas al algoritmului are un impact major asupra eficienței cu care se realizează operația de ordonare. În cazul cel mai defavorabil, algoritmul necesită $O(n^2)$ comparații şi interschimbări.

Pentru obținerea varintei paralele vor trebui parcurse aceleaşi etape ca şi în cazul sortării prin inserție, obținându-se un algoritm ce are ordinul de complexitate egal cu $O(n)$.

Algoritmul *Shell Sort* este de departe cel mai rapid algoritm din clasa celor de complexitate secvențială egală cu $O(n^2)$. Astfel, viteza algoritmului este de aproximativ cinci ori mai mare față de sortarea prin metoda bulelor şi de peste două ori mai mare în comparație cu metoda de sortare prin inserție. Cu toate că algoritmul este mai lent în comparație cu alte metode de sortare (cum ar fi, de exemplu, *Quick Sort*, *Merge Sort* sau *Heap Sort*), simplitatea acestuia îl face să fie o foarte bună alegere pentru cazurile în care se doreşte sortarea repetată a unor liste de dimensiune moderată.

Sortarea prin enumerare (Rank Sort)

Algoritmul de sortare *Rank Sort*, cunoscut şi sub denumirea de sortare prin enumerare, are la baza ideea de a determina rangul fiecărui element al vectorului, rang care va fi apoi utilizat pentru punerea elementului pe poziția corectă. Prin rangul unui element vom înțelege numărul total de elemente din listă care sunt mai mici decât elementul respectiv. Pentru a calcula rangul unui element este necesar ca acesta să fie comparat cu fiecare element al listei. În cazul unei liste ordonate ascendent, rangul unui element va fi de fapt poziția sa actuală. Dacă lista nu este deja ordonată, rangul elementelor ne va indica poziția finală a acestora în lista sortată. Metoda de sortare prin enumerare calculează rangul fiecărui element din listă după care foloseşte aceste valori pentru a plasa elementele în poziția corectă.

Indiferent de relația de ordine existentă inițial între elementele vectorului, algoritmul secvențial va efectua un număr de n^2 iterații şi comparații, datorită existenței a două bucle imbricate, fiecare dintre ele având un număr de *n* iterații. Din acest motiv putem afirma faptul că ordinul de complexitate al algoritmului secvențial este egal cu $O(n^2)$. Deoarece

poziția finală a unui element este determinată pe baza rangului, nu este necesar ca algoritmul să efectueze nici un fel de interschimbări.

Paralelizarea algoritmului se poate realiza prin execuția concurentă, pe procesoarele disponibile în sistem, a iterațiilor buclei exterioare. În felul acesta rangul unui element va fi calculat în mod independent pe unul dintre procesoarele sistemului paralel. Datorită faptului că iterațiile buclei exterioare sunt executate în paralel (complexitate egală cu *O(1)*), ordinul de complexitate al algoritmului paralel de sortare prin enumerare va fi egal cu *O(n)*. Acest rezultat, care este net superior în comparație cu performanțele celui mai rapid algoritm secvențial de sortare, se obține indiferent de modul în care sunt dispuse inițial elementele vectorului.

Deşi face parte din categoria algoritmilor de sortare cu complexitate de ordinul $O(n^2)$, *Rank Sort* este un algoritm foarte simplu care poate fi folosit cu succes pentru ordonarea unor seturi de date de dimensiune moderată

Sortarea prin interclasare (Merge Sort)

Sortarea prin metoda *Merge Sort* are la bază algoritmul de interclasare ce divizează lista de sortat în două subliste care sunt apoi recursiv ordonate după care se execută o operație de interclasare pentru obținerea listei finale. Implementările clasice folosesc trei vectori, doi pentru subliste şi unul pentru obținerea rezultatului final însă algoritmul poate fi modificat astfel încât să lucreze cu doar două masive: unul pentru datele de intrare şi unul în care se vor memora elementele ordonate. Complexitatea algoritmului secvențial este egală cu *O(n log n)*.

Paralelizarea algoritmului se va face prin distribuirea apelurilor către procesoarele sistemului. În felul acesta, fiecare procesor va opera asupra unuia sau a mai multor subvectori obținuți pe baza vectorului inițial. Datorită distribuirii apelurilor recursive, ordinul de complexitate al variantei paralele a algoritmului este egal cu *O(n)*. Acest ordin de complexitate este net superior față de cel al celui mai rapid algoritm secvențial de sortare.

Deşi sortarea prin interclasare este un algoritm stabil ce necesită cu 30% mai puține comparații decât *Quick Sort*, metoda este mai lentă decât *Quick Sort* datorită faptului că efectuează mai multe interschimbări.

Sortarea rapidă (Quick Sort)

Datorită performanțelor sale deosebite, algoritmul *Quick Sort* este unul dintre cei mai utilizați algoritmi de sortare. *Quick Sort* este un algoritm ce poate fi descris foarte simplu la modul teoretic însă scrierea acestuia sub forma de cod sursă a pus întotdeauna probleme. Cercetătorii din domeniul informaticii au avut nevoie de ani de zile pentru a obține o implementare practică a acestui algoritm.

Quick Sort este cel mai cunoscut algoritm de sortare din clasa celor de complexitate medie *O(n log n)*. Algoritmii din această clasă nu sunt la fel de intuitivi însă sunt mult mai performanți decât cei de complexitate $O(n^2)$.

Aşa cum se poate observa cu uşurință, *Quick Sort* reprezintă o variantă mai rapidă a algoritmului *Merge Sort*. Varianta recursivă a algoritmului se compune din patru paşi care vor fi descrişi în continuare:

1. Dacă toate elementele vectorului au fost deja ordonate atunci execuția algoritmului se încheie imediat
2. Se alege unul din elementele vectorului pe post de pivot
3. Se împarte vectorul în doi subvectori (două partiții) care vor conține doar elemente mai mici, respectiv mai mari decât pivotul
4. Algoritmul se repetă recursiv pentru ambele partiții obținute anterior

Cu toate că este un algoritm de sortare foarte rapid, utilizarea *Quick Sort* poate conduce la umplerea stivei atunci când se utilizează seturi largi de date. În plus, datorită complexității, algoritmul nu este o alegere practică atunci când se doreşte ordonarea unor seturi de date de dimensiune redusă.

Modul în care este ales elementul care joacă rol de pivot are un impact major asupra performanțelor algoritmului. În cazul cel mai defavorabil, ordinul de complexitate al algoritmului este egal cu $O(n^2)$ şi se obține în situația în care elementele vectorului sunt deja ordonate iar pe post de pivot se alege elementul cel mai din stânga. Dacă alegerea pivotului

se face însă aleatoriu, atunci complexitatea algoritmului devine din nou egală cu *O(n log n)*.

În general, alegerea pivotului şi partiţionarea vectorului se execută în cadrul unei subrutine separate după care vectorul se parcurge pornind din extremitatea stângă până se găseşte un element mai mare ca pivotul. Algoritmul continuă cu căutarea, pornind din extrema dreaptă, a unui element care este mai mic ca pivotul. Cele două elemente găsite sunt interschimbate iar căutarea continuă până în momentul în care indecşii curenţi pentru subvectorii din stânga şi din dreapta pivotului se intersectează.

Ordinul mediu de complexitate pentru varianta secvenţială a algoritmului este egal cu *O(n log n)*. Paralelizarea algoritmului se face prin distribuirea apelurilor succesive către procesoare diferite ale sistemului. În felul acesta, fiecare procesor va opera asupra uneia sau a mai multor partiţii obţinute pe baza vectorului iniţial. În plus, cele două bucle repetitive din cadrul procedurii de partiţionare care determină indicii elementelor din subvectori care nu respectă relaţia de ordine pot fi executate în paralel iar corectitudinea rezultatelor obţinute este garantată de existenţa unui punct de sincronizare.

Ordinul de complexitate al variantei paralele a algoritmului este dat de complexitatea procedurii de partiţionare care este egală cu *O(n)*. Acest ordin de complexitate este net superior faţă de *O(n log n)*.

Quick Sort reprezintă unul dintre cei mai rapizi algoritmi de sortare. Datorită acestui lucru, algoritmul reprezintă cea mai bună alegere în toate situaţiile în care viteza este un factor foarte important, indiferent de dimensiunea seturilor de date ce se doresc a fi sortate. Cu toate acestea nu trebuie scăpat din vedere faptul că algoritmul nu este stabil iar performanţele acestuia sunt modeste atunci când operează asupra unor liste aproape ordonate.

Bibliografie

[Lad04] S. Ladd, *Guide to Parallel Programming*, Springer-Verlag, 2004

[Wyr04] R. Wyrzykowski, *Parallel Processing And Applied Mathematics*, Springer, 2004

[Dod02] Gh. Dodescu, B. Oancea, M. Raceanu, *Procesare paralelă*, Editura Economica, Bucureşti, 2002

[Gro02] W. Gropp et al, *The Sourcebook of Parallel Computing*, Morgan Kaufmann, 2002

[Gol99] E. Golub et al, *Empirical studies in parallel sorting*, University of Maryland, 1999