UNIVERSITATEA „PETRU MAIOR”
DIN TÂRGU MUREȘ
FACULTATEA DE ȘTIINȚE ȘI LITERE

LUCRARE DE DISERTAȚIE

Coordonator Științific:
Conf. Dr. Ing. Haller Piroska

Masterand:
Potop Radu

- 2011 -

Analiza Algoritmilor de Sortare pe Arhitecturi Paralele

UNIVERSITATEA „PETRU MAIOR” TG. MUREȘ **FACULTATEA DE ȘTIINȚE ȘI LITERE** **Specializarea:** Tehnologia Informației	**LUCRARE DE DISERTAȚIE** **Candidat:** Potop Radu **Anul absolvirii:** 2011
Conducătorul științific: Conf. Dr. Ing. Haller Piroska	**Viza facultății**

a. Tema lucrării de disertație:

Analiza Algoritmilor de Sortare pe Arhitecturi Paralele

b. Problemele principale tratate:

Analiza algoritmilor de sortare, compararea performanțelor pe arhitecturi paralele, algoritmi hibrizi.

c. Bibliografia recomandată:

- Introduction to Algorithms - Thomas H. Cormen et al.
- The Art of Computer Programming, Vol. 3 - Donald E. Knuth
- Algorithms and Data Structures - Robert Sedgewick, Kevin Wayne

d. Termene obligatorii de consultații:

Lunar

e. Locul și durata practicii:

Laborator UPM

Primit tema la data de: 07.02.2011	
Termen de predare: 27.06.2011	
Semnătura șefului de catedră	**Semnătura conducătorului**
Semnătura candidatului	

Cuprins

Indexul Figurilor

Introducere

Lucrarea de față urmărește analiza atât din punct de vedere teoretic, cât și din punct de vedere practic a unei serii de algoritmi de sortare, în situații variate.

Vom începe prin prezentarea algoritmilor de sortare folosiți, a categoriilor din care fac parte aceștia, și a particularităților fiecărei categorii. Vom parcurge noțiuni de complexitatea algoritmilor și noțiuni despre arbori de recurență. Vom studia și implementa algoritmi care rulează atât secvențial cât și în paralel. Aceste implementări le vom rula pe arhitecturi multi-procesor, pentru a observa avantajele și dezavantajele paralelizării. De asemenea, vom construi și o serie de algoritmi hibrizi, care rulează pe arhitecturi paralele.

Toți acești algoritmi vor fi supuși unei serii de teste pentru a evidenția punctele forte sau punctele slabe, în diverse situații, și cu diferite date de intrare. Vom analiza atât o selecție de grafice cât și date brute obținute din urma testelor.

Limbajul de programare folosit pentru implementarea algoritmilor este Python, atât datorită înclinației personale spre acest limbaj dar și pentru că limbajul permite scrierea algoritmilor într-un mod concis, ușor de înțeles, captând astfel esența acestora.

Motivația, importanța și metodologia cercetării

Motivația pentru a realiza o lucrare cu acest subiect provine atât din fascinația personală pentru algoritmi de sortare, cât și din dorința de a implementa acești algoritmi pe arhitecturi paralele, de a-i confrunta între ei și de a analiza rezultatele lor. De asemenea am dorit și realizarea unei serii de algoritmi paraleli hibrizi, de a-i rafina și de a descoperi dacă aceștia oferă sporuri de performanță față de cei paraleli simpli.

Arhitecturile multi-procesor au devenit tot mai răspândite în ultimii ani, găsindu-și locul atât în calculatoarele personale, cât și în echipamente mobile. Din acest motiv, trebuie sa re-gândim atât aplicațiile pe care le folosim zi de zi pentru a beneficia de avantajele oferite de către aceste arhitecturi, dar și algoritmii de bază, cum sunt cei de sortare sau de căutare.

Metodele de cercetare folosite au fost predominant experimentale. Cu toate acestea ele au fost susținute atât de fundamente teoretice cunoscute până în prezent cât și de noțiuni descoperite în cadrul lucrării.

O metodă folosită de-a lungul lucrării a fost rafinarea și selectarea conceptelor sau a rezultatelor relevante obținute în cadrul fiecărui capitol și aplicarea lor în capitolele următoare. Am obținut astfel o gradație de la algoritmi simpli la algoritmi complecși și am redus treptat volumul mare de rezultate inițiale, la rezultate concludente pentru fiecare subiect tratat.

Cap.1. O prezentare a algoritmilor de sortare

Un algoritm de sortare aranjează elementele unei liste într-o anumită ordine. Acești algoritmi sunt folosiți pentru a sorta liste fie de tip numeric, fie alfabetic. Încă de la începuturile informaticii, algoritmii de sortare au atras un număr mare de studii în domeniu. Deși ideea de bază a lor este aparent simplă și oarecum familiară, complexitatea rezolvării eficiente a sortării este mare.

Existența unor algoritmi de sortare eficienți este esențială pentru utilizarea optimă a altor algoritmi, cum ar fi cei de căutare, sau cei de interclasare.

Pentru a considera un algoritm ca fiind de sortare, datele acestuia de ieșire ale acestuia trebuie să îndeplinească două condiții:

- ordinea elementelor din datele de ieșire trebuie să fie ori crescătoare, ori descrescătoare
- datele de ieșire trebuie să fie o rearanjare / permutare a celor de intrare, nu o alterare a acestora

Deși problema algoritmilor de sortare este considerată rezolvată de către mulți specialiști din domeniu, variații pe marginea acestor algoritmi continuă să apară, mai ales cu apariția noilor arhitecturi paralele, și cu accesul în masă la ele.

Pe lângă variațiile pe marginea algoritmilor existenți, au apărut și algoritmi de sortare noi, cum ar fi:

- Library sort, descris de către Michael A. Bender, Martin Farach-Colton și Miguel Mosteiro în 2006 [1] - este un Insertion sort cu elemente pauză, numit și Gapped Insertion sort.
- Bead sort, Joshua J. Arulanandham, Cristian S. Calude și Michael J. Dinneen, 2002 [2] - este un algoritm de sortare complet original, și face parte din

categoria celor care nu folosesc comparații.

- Spread sort, formulat de către Steven J. Ross în 2002 [3] - împrumută elemente de la Bucket sort, Radix sort și Quick sort.
- Tim sort, formulat de către Tim Peters în 2002 [4] - este un algoritm de sortare hibrid care a cunoscut un mare succes odată ce a devenit metoda standard de sortare atât în limbajul Python cât și în Java.

1.1. Clasificarea algoritmilor de sortare

Putem clasifica algoritmii de sortare după o serie de criterii. Cei mai cunoscuți algoritmi sunt cei prin comparație, iar cel mai comun criteriu de clasificare a acestora este după metoda generală de lucru. Astfel avem algoritmi de sortare prin:

Inserție:

- Insertion sort – un algoritm de sortare eficient pe liste de intrare mici sau aproape sortate
- Shellsort
- Binary tree sort
- Cyclesort – un algoritm cu numărul de scrieri în memorie redus

Selecție:

- Selectionsort – eficient pe liste de intrare mici
- Heapsort – un algoritm de sortare cu timp de execuție constant
- Smoothsort – inspirat de Heapsort, dezvoltat de către Edsger Dijkstra

Algoritmii de sortare prin inserție și selecție sunt în general algoritmi care nu pot fi ușor paralelizați, dar pot fi folosiți împreună cu alți algoritmi pentru a forma algoritmi de sortare hibrizi. Mai avem algoritmi de sortare prin:

Partiționare:

- Quicksort – unul dintre cei mai cunoscuți algoritmi de sortare

- Introsort – un hibrid între Quicksort și Heapsort

Interclasare (merging):

- Mergesort
- Timsort – este un hibrid între Mergesort și Insertion sort

Distribuire:

- Bucketsort

Algoritmii prin partiționare, interclasare și distribuire sunt algoritmii care pot fi paralelizați cel mai ușor datorită naturii acestora.

Printre cei mai lenți algoritmi se numără cei prin inter-schimbare:

- Bubblesort
- Odd-even sort - o variație ușor îmbunătățită a Bubble-sort

Toți algoritmii prezentați anterior fac parte din categoria algoritmilor de sortare prin comparație. Exemple de algoritmi de sortare care nu folosesc comparația sunt:

- Radix sort
- Bead sort

Radix sort spre exemplu datează din 1887, fiind folosit de către Herman Hollerith pe mașini tabelare. Modul de lucru al acestuia nu presupune comparația numerelor din lista de intrare, ci comparația cifrelor numerelor, în funcție de poziția pe care o ocupă acestea.

O altă metodă de clasificare a algoritmilor este în funcție de stabilitatea acestora. Un algoritm de sortare stabil va păstra ordinea elementelor care au chei egale. Astfel dacă avem de sortat un șir de dicționare cum ar fi:

```
{2: 'amet'} {1: 'dolor'} {1: 'sit'} {0: 'lorem'} {0: 'ipsum'}
```

Un algoritm de sortare stabil va produce:

```
{0: 'lorem'} {0: 'ipsum'} {1: 'dolor'} {1: 'sit'} {2: 'amet'}
```

Păstrând astfel ordinea inițială a elementelor care au chei egale.

Dintre algoritmii menționați urmează să prezentăm pe larg câțiva dintre ei, și anume: Quicksort, Mergesort, Heapsort și Selection sort.

1.2. Quicksort

Quicksort este un algoritm de sortare conceput de către C.A.R. Hoare în 1962. În practică este unul dintre cei mai rapizi algoritmi de sortare. Quicksort face în medie $O(nlog(n))$ comparații atunci când mărimea listei de intrare este de *n* elemente. De regulă este mai rapid decât alți algoritmi $O(nlog(n))$. În cel mai rău caz, Quicksort face $O(n^2)$ comparații, deși acest caz este de obicei rar. Nu este un algoritm de sortare stabil.

Quicksort se bazează pe principiul *divide et impera*. Algoritmul împarte lista de intrare în două sub-liste mai mici, pe care le sortează în mod recursiv, reaplicând același algoritm. Pașii efectivi sunt:

- Alege un element, numit *pivot*, din listă.
- Re-aranjează lista în așa fel încât elementele mai mici decât pivotul vor fi în stânga acestuia, iar elementele mai mari vor fi în dreapta. Această operație se numește partiționare.
- Aplică recursiv algoritmul pe sub-lista cu elemente mai mici, și pe sub-lista cu elemente mai mari.
- Atunci când mărimea listei de intrare este 0 sau 1, nu este nevoie să mai re-aplicăm algoritmul [5] [6].

1.3. Mergesort

Mergesort este un algoritm de sortare descoperit de către John von Neumann în 1945. La fel ca și Quicksort, este un algoritm care se bazează pe principiul divide et impera. Acesta face atât în medie cât și în cel mai rău caz $O(nlog(n))$ comparații. În cazul Mergesort, o funcție importantă este cea de interclasare (merge). De reținut că acest algoritm este unul stabil.

Pașii pentru realizarea Mergesort sunt:

- Împarte lista inițială în două sub-liste de mărimi egale. De notat este că nu folosim un pivot.
- Aplică recursiv algoritmul pe fiecare dintre sub-liste.
- Atunci când mărimea listei de intrare este 0 sau 1, lista este considerată sortată.
- Interclasează sub-listele sortate într-o singură listă [5].

Algoritmul se bazează pe două principii simple pentru a câștiga performanță:

- O listă mai mică va avea nevoie de mai puțini pași pentru a fi sortată decât o lista de dimensiuni mari.
- Sunt necesari mai puțini pași pentru a construi o listă sortată din două sub-liste sortate, decât două sub-liste ne-sortate.

1.4. Heapsort

Heapsort este un algoritm care face parte din familia algoritmilor de sortare prin selecție. Deși este mai lent în practică decât Quicksort sau Mergesort, acesta are avantajul de a avea atât numărul mediu de comparații cât și numărul maxim de comparații de $O(nlog(n))$. Heapsort nu este un algoritm stabil.

Algoritmul este mai complex decât celelalte prezentate. Pașii acestuia sunt:

- Creează un arbore binar din elementele listei de intrare cu proprietatea că fiecare nod din arbore va fi mai mic decât oricare dintre copii acestuia. Aceasta se numește și *heap property* [6]. Astfel nodul rădăcină va fi cel mai mic element din arbore.
- Elementul rădăcină va fi scos / șters din arbore și concatenat listei sortate.
- Se reface arborele în așa fel încât cel mai mic element din arbore să devină noua rădăcină. Aceasta se întâmplă prin operația de *sift-up* [5].

Algoritmul se reia până când toate elementele au fost concatenate listei sortate, iar arborele nu va mai conține nici un element.

1.5. Selection sort

Selection sort, după cum îi spune și numele, este un algoritm de sortare prin selecție. Este un algoritm simplu, care face $O(n^2)$ comparații, atât în medie cât și în cel mai rău caz. Este considerat ineficient pentru liste de intrare mari, dar poate fi potrivit pentru liste de intrare de dimensiuni mici. Algoritmul nu este stabil.

Pașii acestuia sunt:

- Găsește cel mai mic element din lista de intrare.
- Pune acest element pe prima poziție din lista de intrare.
- Repetă pașii anteriori pentru elementele care au rămas, așezând minimurile din fiecare parcurgere pe pozițiile următoare.

Cap.2. Complexitatea algoritmilor

Am comparat algoritmii de sortare prezentați până acum folosind numărul de comparații făcute de fiecare dintre aceştia. După cum am văzut, un număr de comparații de $O(nlog(n))$ este favorabil, pe când $O(n^2)$ comparații sunt considerate nefavorabile.

Această notație, O mare sau O micron, este o reprezentare relativă a complexității unui algoritm. O micron în cea mai simplă formă a sa reduce comparația dintre algoritmi la o singură variabilă. Această variabilă o putem alege observând algoritmii în cauză. Algoritmii de sortare prezentați se bazează pe comparația dintre două valori pentru a determina ordinea lor. Astfel operația de bază în cazul acestora va fi cea de comparație. Nu vom putea compara un algoritm care face multiplicare aritmetică cu un algoritm care sortează o listă de numere pentru că nu ar avea sens. Dar compararea a doi algoritmi de acelaşi tip ne va da un indiciu despre complexitatea acestora.

Complexitatea se referă atât la durata în timp necesară pentru a executa un anumit algoritm (în cazul de față direct proporțională cu numărul de comparații făcute) cât şi la spațiul necesar de memorie folosit de acel algoritm. Până acum am insistat doar asupra complexității în timp, deoarece acesta este cel mai grăitor indiciu pentru a evalua performanțele unui algoritm de sortare. Cu toate acestea vom observa în continuare că şi spațiul în memorie folosit de un algoritm este un criteriu de măsurare a sa. Atât complexitatea în timp cât şi cea în spațiu sunt exprimate în funcție de *n*, unde *n* este mărimea listei de intrare.

Urmează să facem o analiză succintă a doi dintre algoritmii prezentați, şi anume Quicksort şi Mergesort.

2.1. Quicksort

Dacă luăm exemplul Quicksort, nu este imediat evident faptul că acesta are o complexitate de $O(nlog(n))$. La o analiză a acestuia, îl putem descompune la fiecare pas în operațiile executate și stabili astfel complexitatea sa.

Operația de partiționare, care iterează o dată peste elementele listei de intrare are o complexitate de $O(n)$. În cazul favorabil, dacă pivotul selectat împarte lista în două sub-liste de dimensiuni aproximativ egale, înseamnă că fiecare apel recursiv va procesa jumătate din datele de intrare inițiale. În consecință vom face $O(logn)$ apeluri până când vom ajunge la o listă de dimensiunea 1, pe care, evident, nu mai este nevoie să o sortăm. Înseamnă că înălțimea arborelui format de către apelurile recursive va fi $O(logn)$. Dar apelurile situate la același nivel al arborelui nu vor procesa aceeași parte a listei inițiale, deci fiecare nivel va avea $O(n)$ comparații. Astfel algoritmul final folosește $O(nlog(n))$ comparații.

O altă metodă de analiză este să folosim o relație de recurență, unde $T(n)$ reprezintă timpul necesar pentru a sorta o listă de mărimea *n*. Pentru că un singur apel al Quicksort înseamnă $O(n)$ plus încă două apeluri pe liste de dimensiunea $n/2$, în cazul cel mai favorabil, relația va fi: $T(n)=O(n)+2T(n/2)$, rezultând în $T(n)=O(nlog(n))$, conform *Master Theorem* [6].

În cazul nefavorabil, când cele două sub-liste au dimensiunea *1* și *n-1* arborele construit de către apelurile recursive devine liniar. Astfel relația de recurență va fi: $T(n)=O(n)+T(0)+T(n-1)$ care va rezulta în $T(n)=O(n^2)$ [7] [8].

De menționat că spațiul în memorie folosit de către Quicksort este $O(logn)$, chiar și în cazul nefavorabil.

2.2. Mergesort

Atât în cazul favorabil cât și în cel nefavorabil, Mergesort are un timp de execuție de $O(nlog(n))$. Relația de recurență a algoritmului va fi:

$$T(n)=\begin{cases} 0, & n=1 \\ T(n/2)+T(n/2)+n, & n>1 \end{cases}$$

Deoarece avem nevoie de $T(n/2)$ timp pentru a sorta fiecare sub-listă și de *n* timp pentru operația de interclasare (merge). Mai jos avem o reprezentare grafică a arborelui de recurență pentru Mergesort. Menționăm că spațiul în memorie folosit de către Mergesort este $O(n)$.

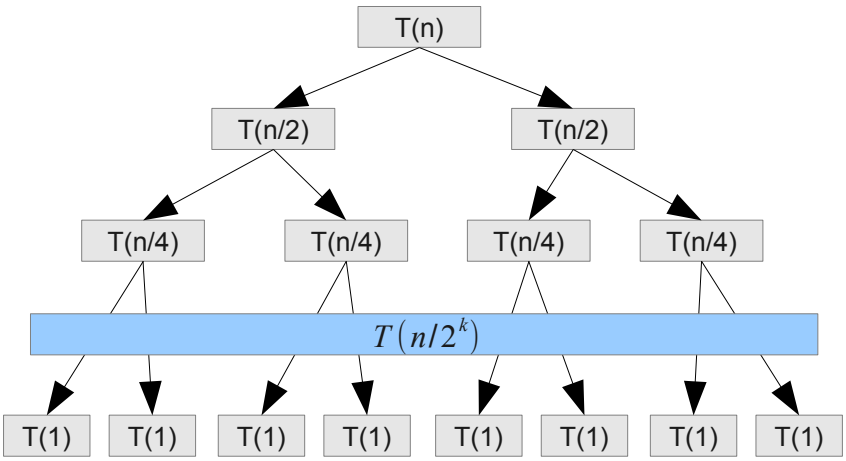

Fig. 1: Arborele de recurență pentru Mergesort

[9]

Cap.3. Algoritmi secvențiali și paraleli

Nu toți algoritmii pot fi paralelizați. Iar dintre algoritmii care pot fi paralelizați, nu în toate cazurile se poate face cu aceeași ușurință. Dintre algoritmii prezentați, cei prin inserție și selecție nu pot fi paralelizați datorită naturii secvențiale a acestora.

Algoritmii care pot fi paralelizați cel mai ușor sunt cei care folosesc metoda divide et impera, datorită modului lor de lucru asupra datelor de intrare. De exemplu, atât Quicksort cât și Mergesort nu au nevoie de o coordonare a datelor de intrare pentru a le sorta, fiecare nivel de iterație lucrând cu o altă listă de intrare.

Un algoritm cum este Selection sort ar fi dificil de paralelizat, deoarece la fiecare iterație trebuie să parcurgem lista inițială de intrare. La fel și în cazul Heapsort, ar fi dificil, dacă nu imposibil, să construim arbori binari din sub-liste de intrare.

Atât în cazul Quicksort, cât și în cazul Mergesort, dacă avem o lista cu *n* elemente și avem la dispoziție o arhitectură cu *p* procesoare, putem împărți lista în *p* sub-liste. Și vom putea sorta fiecare dintre aceste liste într-un timp de:

$$O(\frac{n}{p}\log(\frac{n}{p}))$$

În ambele cazuri vom avea în plus $O(n)$ timp de pre-procesare (Quicksort) sau interclasare (Mergesort). Avantajul celor doi algoritmi este că procesoarele nu trebuie să comunice între ele și nu este nevoie de o sincronizare. Când o sub-listă este disponibilă pentru sortare, procesul poate începe, iar când toate procesele s-au încheiat, lista de intrare este sortată.

Abordarea pe care o vom folosi nu va împărți lista de intrare în mod obligatoriu la cele *p* procesoare disponibile pe hardware, ci va face o împărțire în funcție de nivelul de recurență al algoritmului. De asemenea, datorită limitărilor interpretorului Python, firele de execuție paralele vor fi sub forma unor procese independente.

În cazul Quicksort am ales mai multe tipuri de pivot pentru a observa efectele acestora asupra performanțelor algoritmului. Am ales pe rând ca pivot: primul element din listă, elementul din mijlocul listei, media aritmetică a trei elemente din listă, media între minimul și maximul din listă. Fiecare dintre acești pivoți are avantaje și dezavantaje.

3.1. Paralelizarea algoritmilor

Dacă în mod secvențial nu trebuie să ținem cont în mod deosebit de nivelul de recurență al algoritmului, atunci când vrem sa-l paralelizăm, acest nivel de recurență va juca un rol esențial. Luând Quicksort ca exemplu, va trebui să-l distribuim în mai multe procese, care vor rula în mod paralel. Această distribuire o vom face atunci când lista de intrare este împărțită în două sub-liste de către elementul pivot. Astfel vom putea lansa câte un proces independent pentru fiecare dintre cele două sub-liste.

Mergând în jos pe arborele de recursivitate, cele două procese vor lansa la rândul lor câte alte două sub-procese, obținând astfel un arbore de procese care se suprapune arborelui de recursivitate al algoritmului. Cu toate acestea nu vom continua să lansăm procese până la ultimul nivel al arborelui. Ne vom limita la a lansa procese doar pentru primele câteva niveluri de recurență.

Vom nota aceste niveluri astfel:

- N0 – este nivelul rădăcină și rezultă din apelul inițial al funcției, pe lista inițială de intrare.
- N1 – este rezultatul a 2^1 apeluri ale funcției, fiecare pe câte o sub-listă.
- N2 – este rezultatul a 2^2 apeluri, fiecare pe câte o sub-listă a nivelului anterior.

Și așa mai departe pentru următoarele niveluri.

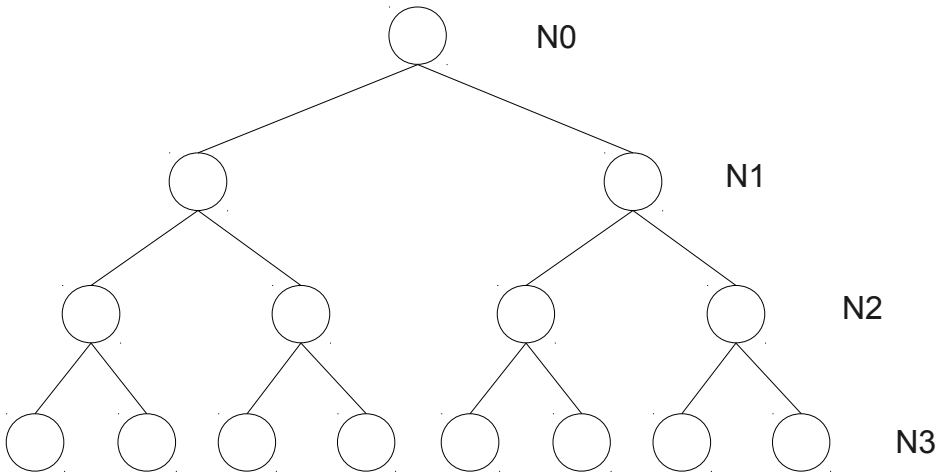

Fig. 2: Nivelurile de recurență

În implementarea de Quicksort și Mergesort dezvoltată, avem posibilitatea să lansăm în execuție 2^i procese separate la fiecare nivel, acolo unde *i* este indicele nivelului. Astfel, vom putea lansa un Quicksort distribuit pe oricâte procese permite arborele de recurență, într-un caz extrem, putem chiar să lansăm câte un proces separat pentru fiecare nivel de recursivitate. Cu toate acestea, pentru experimentele realizate ne-am limitat doar la primele patru niveluri (N0 – N3), respectiv 1, 2, 4 sau 8 procese.

După ce nivelul de recursivitate va depăși nivelul până la care facem distribuirea pe procese, algoritmul va rula în continuare în mod secvențial, apelurile recursive ale funcției având loc în cadrul procesului părinte.

Fiecare dintre aceste apeluri vor sorta lista și vor returna rezultatele apelului părinte. În cazul proceselor lansate, acestea vor returna părintelui o structură de date de tip coadă care va conține lista sortată. Această coadă nu va fi un obiect global, ci unul local, accesibil doar de procesul părinte și cei doi copii ai săi. Procesul părinte după ce lansează cele doua procese copil urmărește această coadă pentru rezultate. Imediat ce obține cele două rezultate de la procesele copil le va concatena cu pivotul (dacă este cazul) și va returna rezultatul procesului său părinte. În cazul în care acesta este deja procesul rădăcină, va returna lista sortată.

Deoarece coada este un obiect local, aceasta va fi creată de fiecare proces părinte care urmează să lanseze sub-procese. Acest Inter-Process-Communication (IPC) se va realiza prin intermediul cozii doar între procese părinte-copil separate. Între apeluri de funcție obișnuite, comunicarea se va realiza prin returnare obișnuită a valorii.

Pseudo-codul pentru o funcție Quicksort secvențială ar putea arăta în felul următor:

```
quicksort(lista):
    dacă lungime(lista) <= 1:
        returnează lista
    altfel:
        alege pivot
        pentru i din lista:
            dacă i < pivot:
                adaugă i la sub-lista elemente_mai_mici
            altfel:
                adaugă i la sub-lista elemente_mai_mari
        elemente_mai_mici = quicksort(elemente_mai_mici)
        elemente_mai_mari = quicksort(elemente_mai_mari)
        returnează elemente_mai_mici + pivot + elemente_mai_mari
```

După cum observăm, cele două apeluri ale quicksort(), pe cele două sub-liste se vor executa în mod secvențial. Dacă dorim să lansăm procese separate pentru fiecare apel al funcției va trebui să modificăm codul. De asemenea va trebui să ținem cont și de nivelul de recurență până la care lansăm procese. Astfel vom avea o variabilă *nivel*, care ne va ajuta în determinarea nivelului de recurență la care ne aflăm la un moment dat. Putem modifica codul în felul următor:

```
quicksort(lista, nivel):
    dacă lungime(lista) <= 1:
        returnează lista
    altfel:
        alege pivot
        pentru i din lista:
            dacă i < pivot:
                adaugă i la sub-lista elemente_mai_mici
            altfel:
                adaugă i la sub-lista elemente_mai_mari
        dacă nivel > 0:
            nivel-=1
            lansează proces quicksort(elemente_mai_mici, nivel)
            lansează proces quicksort(elemente_mai_mari, nivel)
        altfel:
            elemente_mai_mici = quicksort(elemente_mai_mici)
            elemente_mai_mari = quicksort(elemente_mai_mari)
        returnează elemente_mai_mici + pivot + elemente_mai_mari
```

În exemplul de față am exclus partea de comunicare între procese și manipularea structurilor de tip coadă, pentru a ne concentra asupra părții esențiale ale algoritmului. Când vom apela funcția va trebui să îi specificăm și nivelul până la care dorim să lansăm procese separate. Odată ce acest nivel va ajunge la 0 se va opri lansarea de procese și algoritmul va trece la execuția secvențială a algoritmului. Spre exemplu, dacă nivelul va fi egal cu 2, atunci vom lansa procese până la N2.

Folosind această metodă, vom putea anterior să combinăm doi algoritmi într-un algoritm hibrid.

Cap.4. Rezultate experimentale ale algoritmilor

În continuare vom observa performanțele a unei serii de algoritmi, folosind diverse date de intrare. Testele s-au realizat în felul următor:

- Fiecare algoritm a fost rulat având ca date de intrare: o listă aleatoare, o listă sortată crescător și o listă sortată descrescător.
- Fiecare algoritm a fost rulat de 10 ori pe fiecare listă de intrare, după care s-a făcut o medie aritmetică din timpii de execuție. Timpii sunt în milisecunde și rotunjiți în jos.
- Fiecare algoritm a fost rulat pe liste de 10.000 (10k), 100.000 (100k), 1.000.000 (1M) și 5.000.000 (5M) de elemente.
- Elementele din listă sunt numere naturale. În cazul listelor aleatoare, acestea au valori de la 0 la (nr de elemente al listei) * 10.
- Algoritmii au fost rulați atât în mod secvențial, pe un procesor/core, cât și în mod paralel (cei care permit acest lucru) pe 2 procesoare/cores și pe 6 procesoare/cores.
- Toți algoritmii implementați au fost testați pentru corectitudine confruntându-le rezultatele cu cele ale algoritmului de sortare standard din Python.

4.1. În mod secvențial

Pentru o pre-selecție, am ales algoritmii Quicksort, Mergesort și Heapsort. În cazul Quicksort am ales mai multe variante de pivot: primul element al listei (quicksort_first), elementul din mijlocul listei (quicksort_middle), media a trei elemente din listă (quicksort_avg3) și media dintre minimul și maximul din listă (quicksort_minmax).

Graficele următoare ilustrează rezultatele obținute. Nu vom afișa în lucrarea prezentă toate graficele obținute, ci doar câteva relevante.

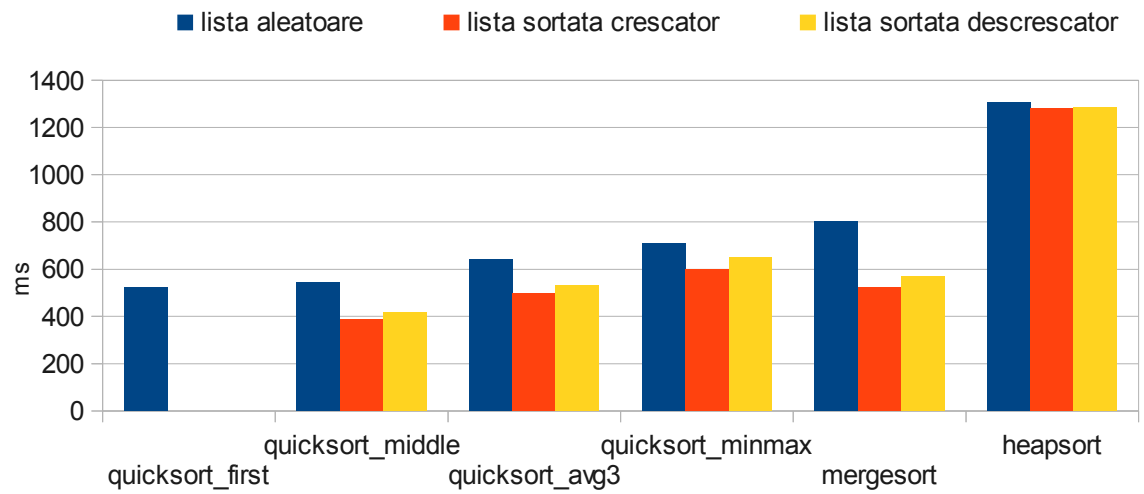

Fig. 3: 100k elemente, secvențial

Este imediat evident din figura 3 faptul că Quicksort cu pivot pe primul element nu are rezultate pentru liste sortate crescător și descrescător. Aceasta se întâmplă din cauza că cele două liste reprezintă cazul nefavorabil pentru algoritm, arborele de recurență transformându-se într-unul liniar. Rezultatele nu au putut fi calculate deoarece se depășesc limita de recursivitate.

De asemenea, este de așteptat ca quicksort_first și quicksort_middle să fie cei mai rapizi algoritmi, deoarece găsirea elementului pivot durează $O(1)$ în ambele cazuri. Pentru

quicksort_minmax, unde trebuie să găsim minimul și maximul din listă, operația va avea o durată de $O(n)$, ceea ce se reflectă în performanțele algoritmului.

De asemenea, observăm că lista sortată reprezintă cazul favorabil pentru toți algoritmii, ceea ce se reflectă în timpii foarte mici de execuție comparativ cu lista aleatoare. Doar Heapsort are performanțe constante în acest caz, nefiind afectat la fel de mult ca ceilalți algoritmi de tipul listei de intrare.

Nu am inclus Selection sort în grafice deoarece acesta are performanțe atât de slabe (și implicit durează mult execuția), încât devine irelevant. Vom reveni asupra lui la algoritmi hibrizi pentru a exploata o proprietate unica a sa.

4.2. În mod paralel

În mod paralel avem mai multe variante de rulare a algoritmilor. În primul rând vom putea lansa câte 2, 4 sau 8 procese pentru fiecare algoritm. Această metodă va împărți lista de intrare, și o va distribui la mai multe procese care o vor putea sorta în paralel.

De asemenea, vom rula acești algoritmi pe rând pe o stație cu 2 procesoare/cores și pe una cu 6 procesoare/cores. Cea din urmă permițându-ne să observăm comportamentul algoritmilor cu un grad mare de paralelism.

Nu vom lansa mai mult de 8 procese, deoarece după cum vom observa, comunicarea dintre prea multe procese va afecta în mod negativ performanțele algoritmului.

Am ales Quicksort cu pivot pe mijloc și Mergesort pentru a rula în paralel.

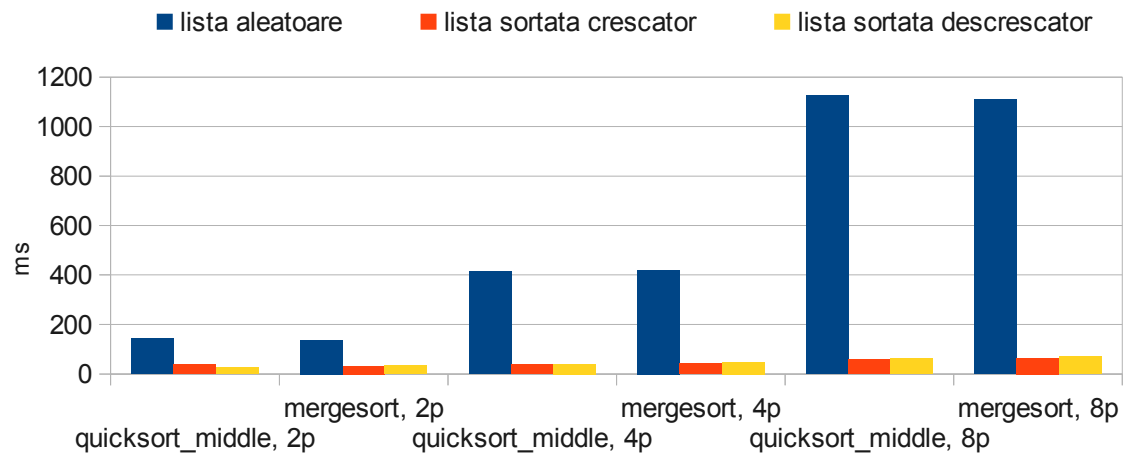

Fig. 4: 10k elemente, paralel, 2 cores

2p, 4p și 8p reprezintă numărul de procese la care am distribuit algoritmul. Acestea corespund cu o paralelizare până la nivelurile N1, N2 și respectiv N3.

Putem deja observa din figura 4 faptul că pentru liste de intrare de dimensiuni mici, comunicarea între procese va afecta semnificativ performanțele algoritmului. Odată cu creșterea mărimii listei, aceste diferențe se vor estompa, după cum putem observa în continuare.

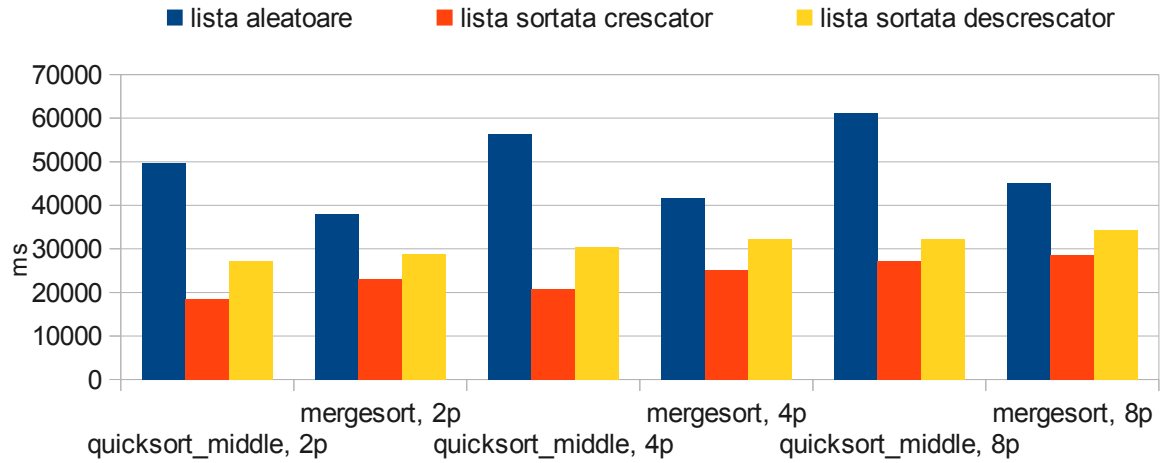

Fig. 5: 5M elemente, paralel, 2 cores

De asemenea, din figura 5 observăm faptul că Mergesort este mai rapid decât Quicksort, odată cu creșterea mărimii listei de intrare: 38119ms vs. 49726ms, pentru 2p.

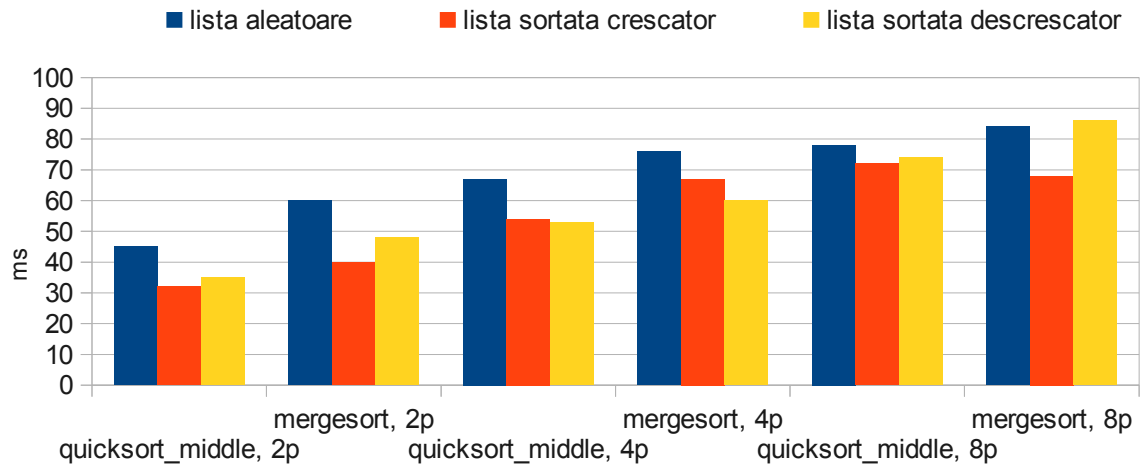

Fig. 6: 10k elemente, paralel, 6 cores

Este imediat aparent în cazul rulării pe 6 cores (figura 6) faptul că diferențele între 2p și 8p sunt mult mai mici decât în cazul rulării pe 2 cores, deși are loc aceeași comunicare între procese. De aici tragem concluzia că odată ce numărul de procese depășește semnificativ numărul de procesoare/cores, performanțele algoritmului se vor degrada simțitor datorită limitării hardware. Și în cazul a 6 cores, diferențele se estompează odată cu creșterea listei de intrare la 5 milioane de elemente.

4.3. O comparație între secvențial și paralel

Pentru a vizualiza rezultatele mai ușor va trebui sa facem o comparație directă între algoritmii rulați în mod secvențial și cei rulați în paralel. În primul rând va trebui să eliminăm din volumul mare de rezultate obținute și să le păstrăm doar pe cele relevante. Vom elimina rezultatele pentru liste sortate crescător și descrescător. Vom păstra doar rezultatele pentru liste aleatoare, deoarece acestea sunt cazurile cele mai apropiate de condițiile reale.

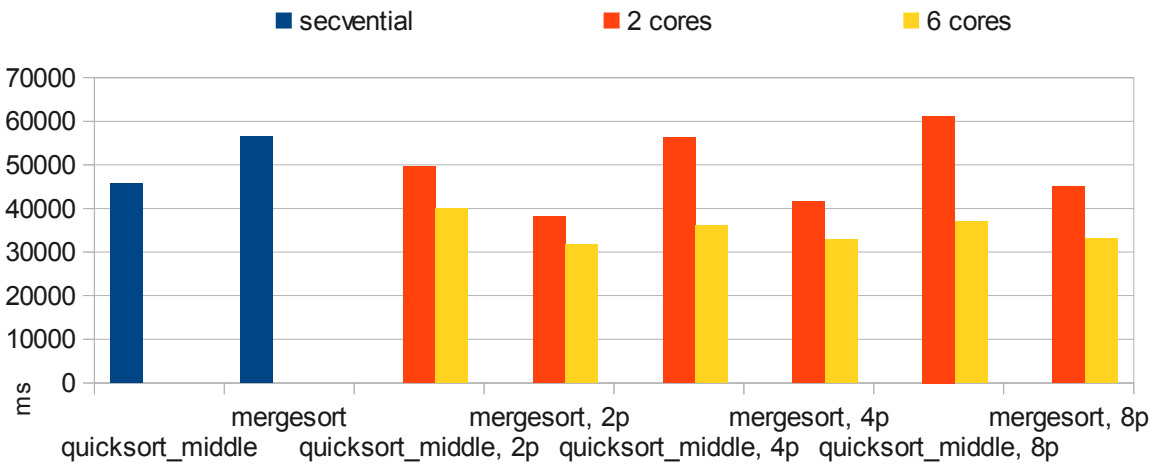

Fig. 7: 5M elemente, comparație secvențial vs. paralel, listă aleatoare

Din figura anterioară observăm că performanțele Mergesort sunt mai bune decât ale Quicksort în varianta paralelă a algoritmului. În mod secvențial el având în toate cazurile performanțe mai slabe decât Quicksort cu pivot pe mijloc.

Mergesort va beneficia de îmbunătățiri majore de pe urma paralelizării, diferențele în cazul anterior fiind de: 56404ms pentru Mergesort secvențial vs. 31698ms pentru Mergesort paralel cu 2 procese. Quicksort în schimb nu va beneficia de îmbunătățiri la fel de semnificative în urma paralelizării, în unele cazuri observându-se o degradare.

Cap.5. Construirea algoritmilor hibrizi

Algoritmii hibrizi sunt în general compuși din doi algoritmi, unul de tipul divide et impera cum este Quicksort și Mergesort, și unul secvențial, cum este Selection, Insertion, Heapsort. Ideea din spatele lor este că odată ce algoritmul principal, divide et impera, trece de un anumit nivel de recursivitate acesta va avea performanțe slabe, datorită listelor de intrare foarte mici și a *overhead-ului* creat de apelurile recursive pe acestea.

Din aceste motive, de la un anumit nivel de recursivitate, vom trece la un algoritm secvențial. Cum alegem acest nivel? Până acum în teste am avut liste de: 10k, 100k, 1M și 5M de elemente. Putem determina nivelul de recursivitate pentru fiecare lungime:

- $10\text{k} \approx 2^{13}$
- $100\text{k} \approx 2^{17}$
- $1\text{M} \approx 2^{20}$
- $5\text{M} \approx 2^{22}$

Lungimile listelor de intrare nu vor determina crearea unui arbore de recurență complet, de aici și aproximarea ca puteri a lui 2. În continuare, pentru construirea algoritmilor hibrizi va trebui sa analizăm performanțele algoritmilor secvențiali pe liste de dimensiuni mici.

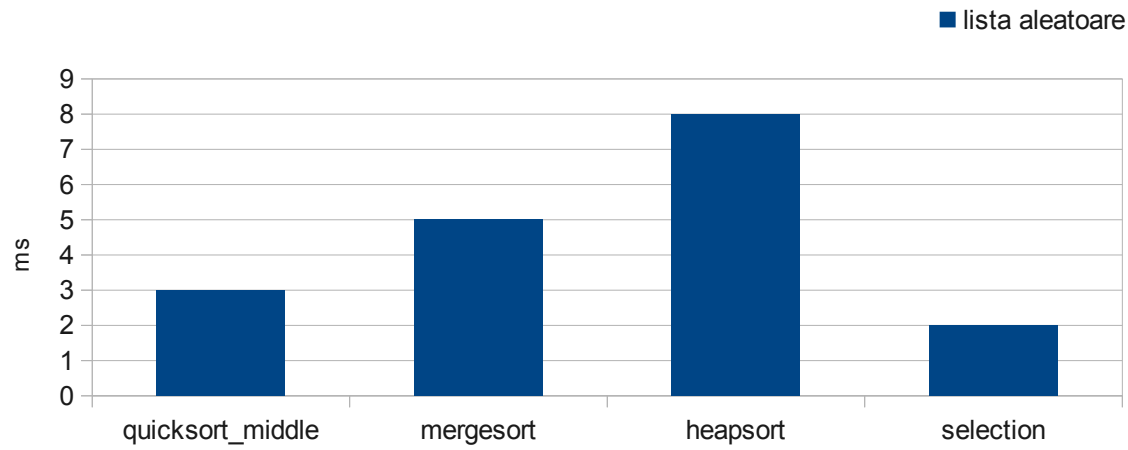

Fig. 8: 1000 elemente, secvențial

După cum observăm, Selection sort va avea performanțe mai bune decât ceilalți algoritmi pentru o listă de dimensiuni reduse cum este cea de 1000 elemente. Cu toate acestea, odată cu trecerea de aprox. 4000 elemente, performanțele sale se degradează simțitor.

Pentru algoritmii hibrizi creați am ales nivelul 12 de recurență ca nivel optim, unde algoritmul principal se oprește și continuă sortarea cel secvențial. Putem să determinăm lungimea listei sortate de către acești sub-algoritmi. Ținând cont că $2^{12} = 4096$, vom avea:

- $10k / 4096 \approx 2$ elemente
- $100k / 4096 \approx 24$ elemente
- $1M / 4096 \approx 244$ elemente
- $5M / 4096 \approx 1220$ elemente

Putem descrie algoritmul hibrid complet astfel:

Până la nivelul 1, 2 sau 3 vom lansa procese separate, respectiv 2, 4 sau 8 procese. După acest nivel și până la nivelul 12 vom rula algoritmul principal mai departe în mod secvențial. De la nivelul 12 vom rula sub-algoritmul pe listele de intrare mici. Mai menționăm că dacă lista inițială de intrare este mai mică de 4096 elemente, sub-algoritmul nu va mai intra în funcțiune. Astfel algoritmul nostru devine unul adaptiv.

Cap.6. Rezultate experimentale ale algoritmilor hibrizi

În urma testelor efectuate am obținut o serie de rezultate pentru algoritmii hibrizi. Dintre acestea le vom prezenta doar câteva relevante. În figura de mai jos avem o comparație între algoritmii secvențiali, paraleli și hibrizi. Am construit mai mulți algoritmi hibrizi, astfel:

- Quicksort, având ca sub-algoritm Heapsort.
- Quicksort cu sub-algoritm Selection sort.
- Mergesort cu sub-algoritm Heapsort.
- Mergesort cu sub-algoritm Selection sort.

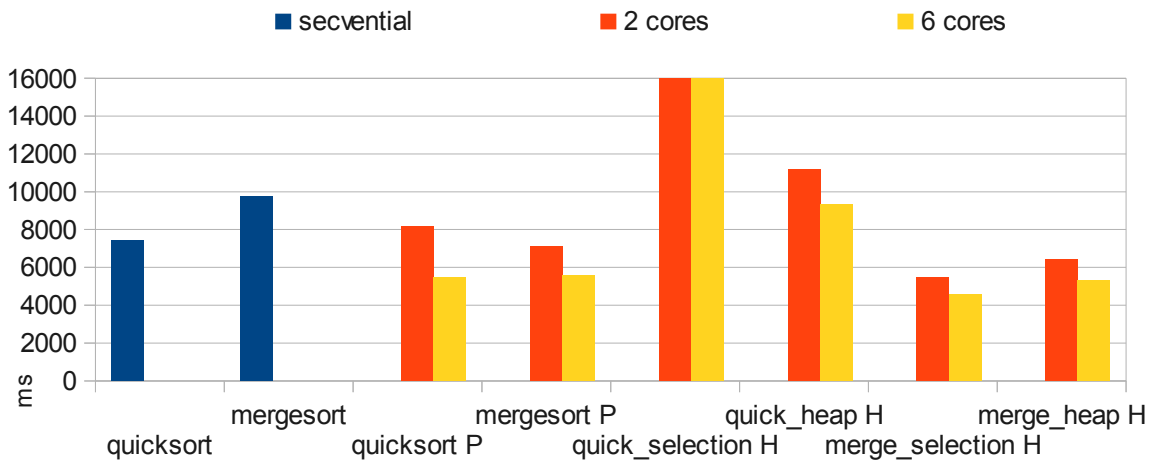

Fig. 9: 1M elemente, comparație secvențial vs. paralel vs. hibrid, listă aleatoare, 2 procese

Quicksort cu Selection sort a avut în general rezultate foarte slabe. Nici varianta cu Heapsort nefiind mai avantajoasă. Putem trage concluzia că Quicksort nu este un candidat bun pentru algoritmii hibrizi, acesta având performanțe mult mai bune în varianta paralelă simplă.

Mergesort în schimb, după cum am observat și până acum, a continuat să-și îmbunătățească timpii de execuție. Acesta a câștigat foarte mult în performanță odată cu paralelizarea simplă, iar după hibridizare a devenit mai rapid, chiar dacă saltul nu a fost atât de spectaculos. Astfel algoritmii hibrizi având la bază Mergesort s-au dovedit a fi cei mai rapizi din toată seria de teste rulate până acum.

Putem ilustra performanțele obținute astfel:

Pentru 2 procese:

- 1M elemente, merge_selection este de 2.14 ori mai rapid decât mergesort secvențial și de 1.22 ori mai rapid decât mergesort paralel simplu.
- 5M elemente, merge_heap este de 1.70 ori mai rapid decât mergesort secvențial, dar de 1.05 ori mai lent decât mergesort paralel simplu.

Pentru 4 procese:

- 1M elemente, merge_selection este de 2.00 ori mai rapid decât mergesort secvențial și de 1.15 ori mai rapid decât mergesort paralel simplu.
- 5M elemente, merge_selection este de 1.77 ori mai rapid decât mergesort secvențial și de 1.05 ori mai lent decât mergesort paralel simplu.

6.1. Concluzii ale experimentelor

Putem declara merge_selection ca fiind algoritmul cel mai rapid, fiind în medie de 1.90 ori mai rapid decât mergesort secvențial și de 1.10 ori mai rapid decât mergesort paralel simplu. Varianta care rulează pe 4 procese oferă performanțe constante atât pe 2 cores cât și pe 6 cores, pe când varianta cu 2 procese este dispusă mai mult la fluctuații.

Trebuie să ținem cont că algoritmul prezentat este adaptiv în condițiile în care lista inițială de intrare este de dimensiuni mici (sub 4096). În aceste condiții sub-algoritmul nu va intra în funcțiune, iar performanțele per ansamblu vor fi constante.

Cap.7. Posibilități de îmbunătățire

Am observat până acum că algoritmii de tip divide et impera beneficiază de îmbunătățiri majore ale performanțelor atunci când sunt paralelizați. Datorită naturii lor, și a lipsei necesității de coordonare între firele de execuție aceștia pot fi paralelizați oricât permite arborele de recurență. Din moment ce două fire de execuție nu vor avea aceleași date de intrare, acestea nu interferează între ele și astfel putem rula algoritmul pe un număr mare de procesoare.

Un tip de arhitectură care oferă un grad ridicat de paralelism sunt generațiile recente de plăci grafice. Tehnologii precum CUDA sau OpenCL permit rularea programelor generice direct pe procesoarele plăcilor grafice. Spre deosebire de procesoarele obișnuite (CPU), procesoarele grafice (GPU) conțin un număr mare de cores, care pot rula aceeași secvență de cod pe date de intrare diferite. Acest număr de cores poate ajunge până la 480 dacă folosim cele mai puternice plăci grafice actuale. Aceste arhitecturi se mai numesc și many-core.

Astfel, am putea implementa algoritmii de sortare prezentați pentru a beneficia de acest nivel ridicat de paralelism și pentru a obține performanțe mai bune decât folosind CPU.

Programarea GPU este totuși diferită de programarea procesoarelor clasice. Cores are unui GPU vor rula un număr de fire de execuție în paralel. Acestea sunt organizate în matrici și blocuri, având fiecare adrese pentru identificare [10]. De asemenea, pentru a rula un algoritm pe plăcile grafice, trebuie să încărcăm atât codul (numit kernel) pe GPU cât și datele de intrare în memoria plăcii grafice. Astfel vom avea două entități care comunică între ele: unul fiind *host* (CPU cu memoria calculatorului) iar celălalt fiind *guest* (GPU cu memoria plăcii grafice).

Probabil singurul dezavantaj al acestei tip de arhitectură este faptul că încă nu este răspândită pretutindeni.

Concluzii

În lucrarea de față am făcut o prezentare a algoritmilor de sortare și am parcurs conceptele teoretice din spatele acestor algoritmi. Am analizat atât complexitatea lor în timp și spațiu cât și arborii de recurență. Am studiat metode de paralelizare a lor și folosind aceste cunoștințe am implementat o serie de algoritmi atât secvențiali cât și paraleli. Mai departe am dezvoltat acești algoritmi în algoritmi paraleli hibrizi adaptivi care au beneficiat de sporuri de performanță.

Implementările efectuate ne-au ajutat în obținerea de rezultate experimentale pentru fiecare categorie de algoritmi: secvențiali, paraleli sau hibrizi. Folosind aceste rezultate am subliniat diferențele dintre algoritmi, comportamentul lor în situații diferite și slăbiciunile sau punctele lor forte pentru diverse date de intrare.

Per ansamblu putem declara algoritmul hibrid Mergesort cu Selection sort pe patru procese paralele, ca fiind cel mai rapid și constant dintre cele studiate și experimentate.

De asemenea am amintit că un tip de hardware cum sunt plăcile grafice ne-ar putea ajuta în obținerea de performanțe mai bune ale acestor algoritmi, datorită procesoarelor specializate pentru paralelism.

Cu toate acestea trebuie să ținem cont că este important să avem algoritmi performanți prin construcția lor. Odată ce am făcut acest pas putem să îi implementăm pe diverse tipuri de hardware pentru a obține sporurile de performanță oferite de hardware-ul respectiv.

Bibliografie

1. Insertion sort is O(n logn) - Michael A. Bender, Martin Farach-Colton, Miguel Mosteiro - 2006 - http://www.cs.sunysb.edu/~bender/newpub/BenderFaMo06-librarysort.pdf
2. Bead-Sort: A Natural Sorting Algorithm - Joshua J. Arulanandham, Cristian S. Calude Michael J. Dinneen - 2002 - http://www.cs.auckland.ac.nz/~jaru003/research/publications/journals/beadsort.pdf
3. The Spreadsort High-performance General-case Sorting Algorithm - Steven J. Ross - 2002
4. Tim sort - Tim Peters - 2002 - http://bugs.python.org/file4451/timsort.txt
5. The Art of Computer Programming, Vol. 3, pag. 105, 158, 138 - Donald. E Knuth - 1998
6. Introduction to Algorithms, pag. 145, 130, 73 - Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein - 2001
7. A killer adversary for quicksort - M. D. McIlroy - 1999 - http://www.cs.dartmouth.edu/~doug/mdmspe.pdf
8. Algorithms and Complexity, pag. 32 - Herbert S. Wilf - 1994 - http://www.math.upenn.edu/%7Ewilf/AlgoComp.pdf
9. Algorithms and Data Structures, Mergesort and Quicksort - Robert Sedgewick, Kevin Wayne - 2005 - http://www.cs.princeton.edu/courses/archive/spr07/cos226/lectures/sort2.pdf
10. Nvidia CUDA C Programming Guide , version 3.2, pag. 8 - 2010