CR3.3B
Zane Livia
Proiect la Algoritmi paraleli si distribuiti

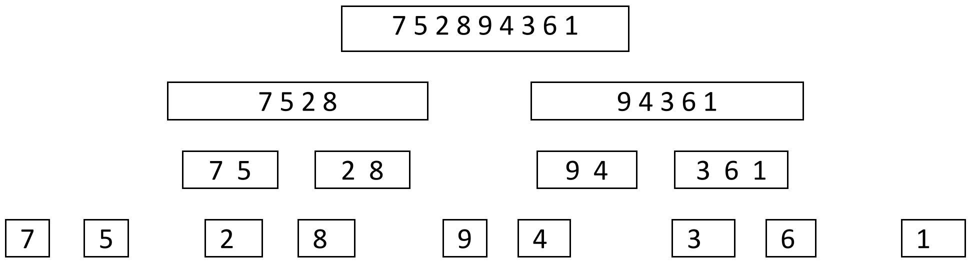
Algoritmul de sortare merge-sort

Generalitati

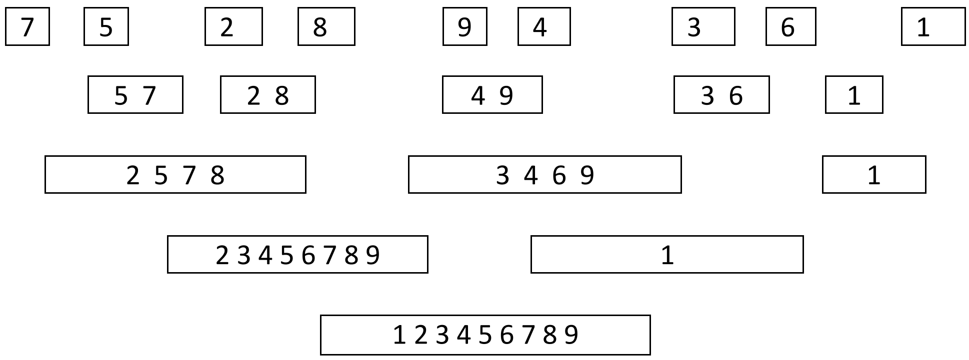
Este un algoritm de sortare care funcționează prin împărțirea unei liste în sub-liste mai mici, sortând fiecare sub-lista și apoi îmbinând înapoi sub-listele sortate, pentru a forma lista sortată finală.

Merge-Sort este un algoritm recursiv de tip Divide&Cucereste, care împarte continuu lista în jumătati (Divide) până când nu mai poate fi împărțită în continuare (fiecare sub-lista contine cel mult un element). Aceasta înseamnă că dacă sub-lista devine goală sau mai are un singur element , împărțirea in subliste se va termina, fiind cazul de bază pentru oprirea recursivitatii. Dupa fiecare injumatatire, pentru ambele sub-liste se reapeleaza functia Merge-Sort.

Exemplu de functionare a algoritmului:



Pentru partea de Cucereste si Combinare solutii partiale, se foloseste o functie de imbinare (Merge) care reconstruieste lista, imbinand (Combina)doua cate doua sub-liste intre ele (ex. A si B), sortand totodata elementele (Cucereste) atunci cand le insereaza in sub-lista (C) care va contine elementele din cele doua sub-liste (A si B).



Mai concret, daca listele A si B au un index i si j, mai este necesara inca o lista C cu index k, in care se vor insera elementele celor doua liste astfel:

- se pleaca cu indexul i, j si k, se compara A[i] cu B[j], iar cel mai mic element se depune in C[k];
- se incrementeaza indexul elementului cel mai mic si se continua algoritmul pana cand se ajunge la max_i sau max_j;
- daca raman elemente intr-una din listele A sau B, aceste elemente se adauga in lista C.

Algoritmul Merge-Sort Recursiv (top-down) in pseudocod este urmatorul:

```
Merge-Sort (list, first-index, last-index)
   if (first-index < last-index)
      middle-index = (first-index  + last-index) / 2
      Merge-Sort(list, first-index , middle-index)        /* apelare  recursiva pentru prima jumatate*/
      Merge-Sort(list, middle-index, last-index)          /* apelare  recursiva pentru a doua jumatate*/
      Merge(list, first-index, middle-index, last-index)
```

Algoritmul Merge-Sort Iterativ (bottom-up) in pseudocod este urmatorul:

```
Merge-Sort-Iterative (list, size)
   for (width = 1; width < size; width = 2 * width)           /* width is the size of working list in Merge*/
      for (i = 0; i < size; i = i + 2 * width)
         Merge(list, i, min(i+width, size), min(i+2*width, size))
```

Functia Merge in pseudocod, valabila pentru ambii algoritmi, este urmatoarea:

```
Merge(list, left, middle, right)
   let leftSubList[middle - left], rightSubList[right - middle]
   for (i = 0; i < middle - left; i++)                        /* copy first half of list in leftSubList*/
      leftSubList[i] = list[left + i]
   for (j = 0; j < right - middle; j++)                       /* copy second half of list in rightSubList*/
      rightSubList[i] = list[middle + j]
   let i=j=k=0
   while(i < leftSize && j < rightSize)                       /* move back in list, the smaller element between*/
      if (leftSubList[i] <= rightSubList[j])                  /* leftSubList[i] and rightSubList[j]), then increment*/
         list[k++] = leftSubList[i++]                         /* k and i or j */
     else
         list[k++] = rightSubList[j++];
  while (i < left)                                            /* move remaining elements from leftSubList, if any*/
     list[k++] = leftSubList[i++]
while (j < right)                                             /* move remaining elements from rightSubList, if any*/
     list[k++] = rightSubList[j++]
```

Analiza complexitatii algoritmului

Algoritmul Merge-Sort are o complexitate de timp O(n log n), in toate cazurile (cel mai bun, mediu, cel mai rau), fiind o sortare stabilă, ceea ce înseamnă că ordinea elementelor cu valori egale este păstrată în timpul sortării. Avand in vedere ca algoritmul utilizeaza o lista temporara pentru impartirea in sub-liste, este necesar un spatiu suplimentar $\Theta(n)$.

Puncte tari :

- rapid
- stabil
- poate fi implementat intr-o abordare paralela

Puncte slabe:

- itereaza chiar si pentru cazul cand lista initiala este sortata
- utilizeaza memorie suplimentara.

Metoda de testare

Pentru realizarea comparatiilor dintre variantele secventiale si cele paralele de rulare a algoritmilor merge-sort recursive si secvential, ***am folosit secvente de numere naturale*** de la 1 pana la n (n cu valori mici, medii si mari), ***pe care le-am*** amestecat random in cate o lista, apoi am ***salvat*** lista ***intr-un fisier***.

Respectivul fisier a constituit date de intrare pentru toate variantele algoritmului, pentru a ne asigura ca ***fiecare algoritm ruleaza acelasi set de date de intrare***.

Pe langa algoritmii implementati, am folosit si sortarea oferita de limbajul de programare folosit (C++), respective functia **sort()** aplicata pe **list** (lista dubla inlantuita) si **forwardlist** (lista simplu inlantuita).

Asa cum am precizat anterior, algoritmii au fost implementati in C++, folosind CodeBlocks IDE.

Viteza de executie a algoritmilor a fost masurata cu ajutorul functiilor high_resolution_clock::now() si duration_cast.count() din biblioteca <chrono>, iar memoria folosita, cu ajutorul functiei GlobalMemoryStatusEx din biblioteca <windows.h>.

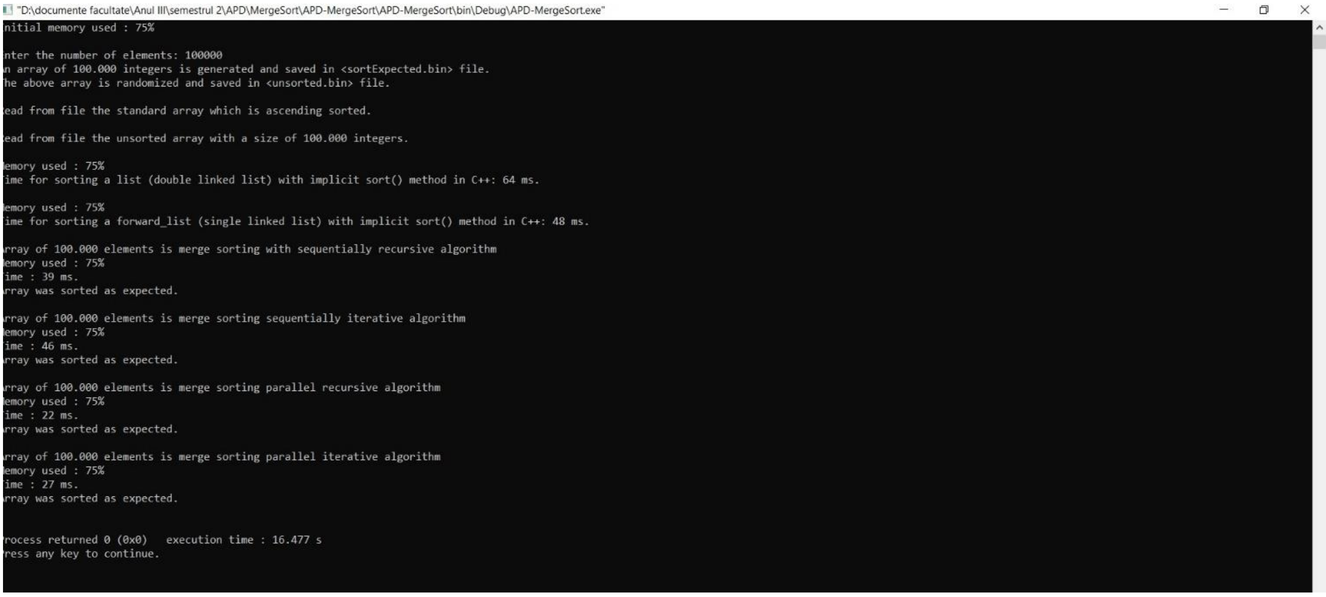
Pentru partea de testare am folosit doua calculatoare (laptopuri), cu urmatoarele specificatii:

- Laptop 1 – CPU: AMD Ryzen 5, 6 core/12 threads, 4.20 GHz; RAM: 8 GB;
- Laptop 2 – CPU: Intel i5, 2 core/4 threads, 2.30 GHz; RAM: 8 GB,

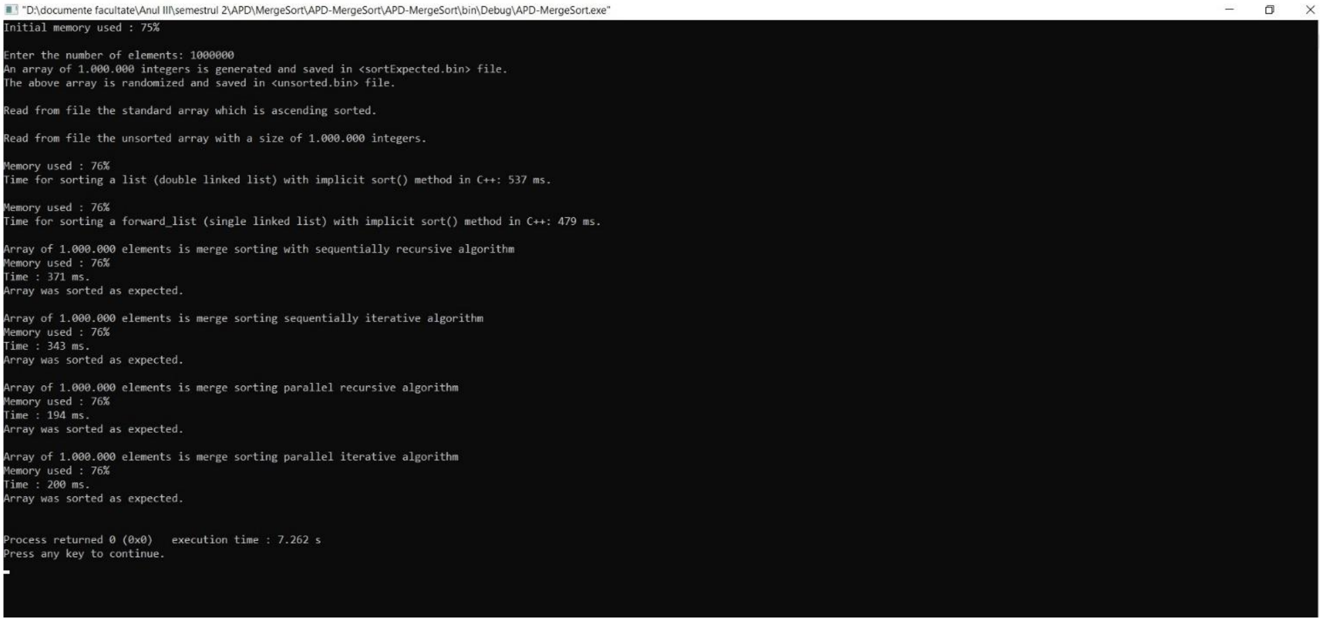
ambele ruland SO Windows 10 Pro - 64bit, pe SSD M.2.

Exemple de rulare a codului

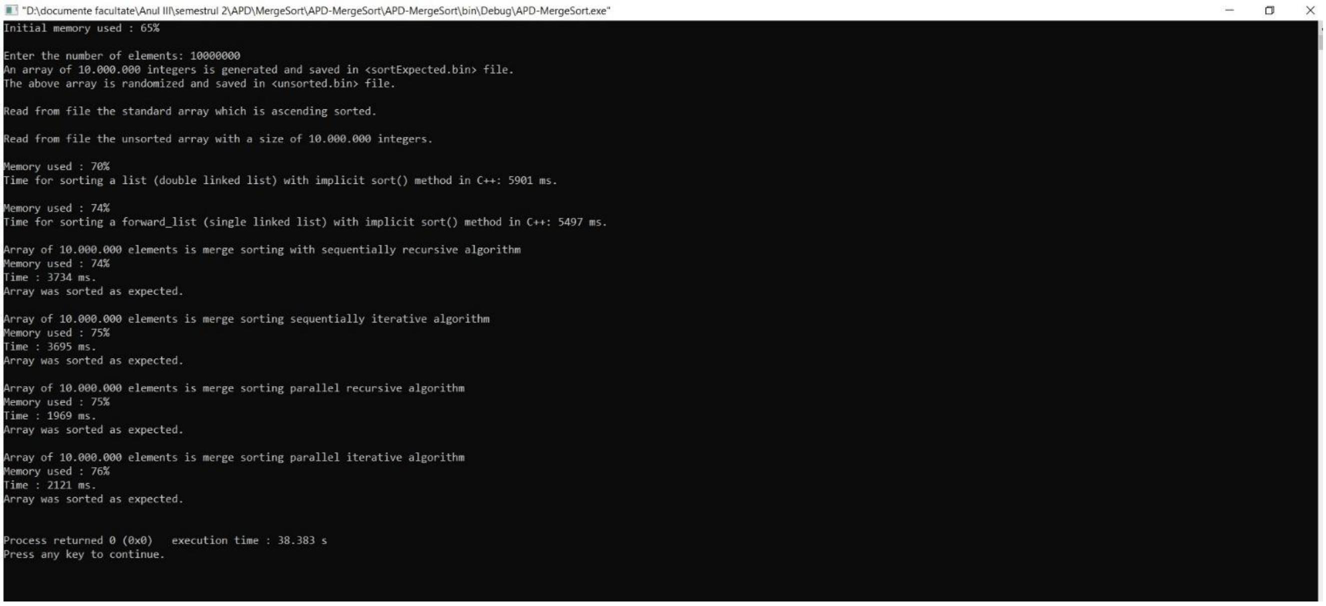
Prezint in continuare exemplul de rulare a codului pentru o baterie de test de 100.000 de numere intregi (timpii de rulare pentru un numar mai mic de elemente este insignifiant):



Exemplul de rulare pentru 1.000.000 elemente:

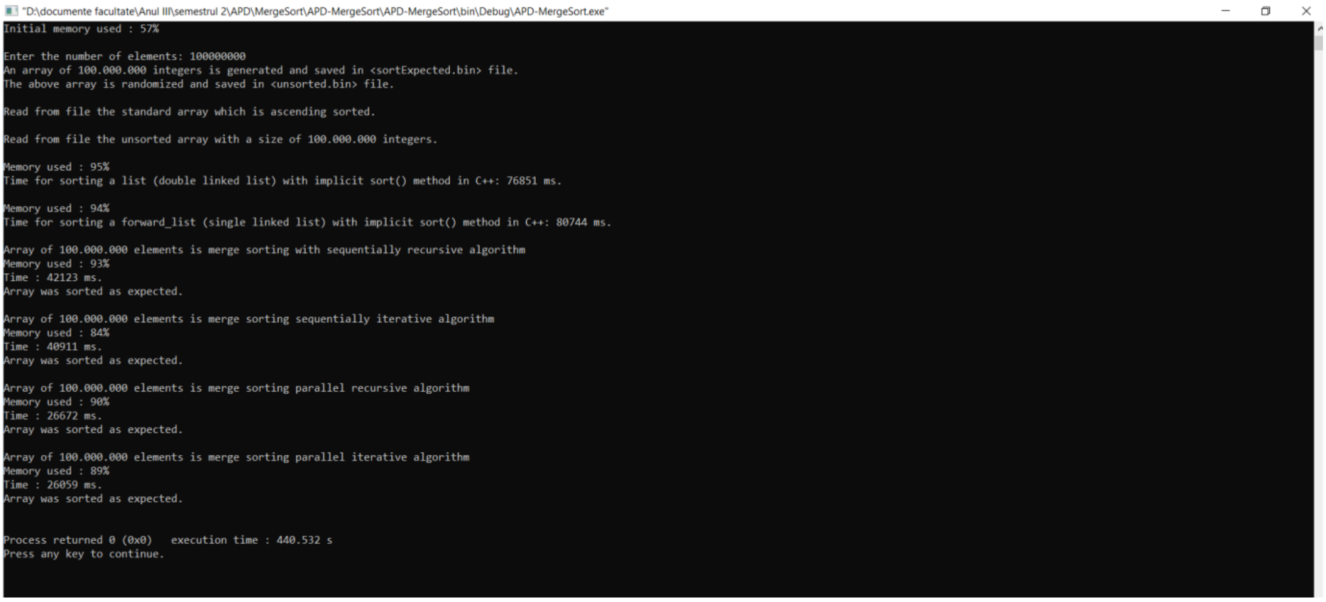


Exemplul de rulare pentru 10.000.000 elemente:



```
"D:\documente facultate\Anul III\semestrul 2\APD\MergeSort\APD-MergeSort\APD-MergeSort\bin\Debug\APD-MergeSort.exe"
Initial memory used : 65%

Enter the number of elements: 10000000
An array of 10.000.000 integers is generated and saved in <sortExpected.bin> file.
The above array is randomized and saved in <unsorted.bin> file.

Read from file the standard array which is ascending sorted.

Read from file the unsorted array with a size of 10.000.000 integers.

Memory used : 70%
Time for sorting a list (double linked list) with implicit sort() method in C++: 5901 ms.

Memory used : 74%
Time for sorting a forward_list (single linked list) with implicit sort() method in C++: 5497 ms.

Array of 10.000.000 elements is merge sorting with sequentially recursive algorithm
Memory used : 74%
Time : 3734 ms.
Array was sorted as expected.

Array of 10.000.000 elements is merge sorting sequentially iterative algorithm
Memory used : 75%
Time : 3695 ms.
Array was sorted as expected.

Array of 10.000.000 elements is merge sorting parallel recursive algorithm
Memory used : 75%
Time : 1969 ms.
Array was sorted as expected.

Array of 10.000.000 elements is merge sorting parallel iterative algorithm
Memory used : 76%
Time : 2121 ms.
Array was sorted as expected.


Process returned 0 (0x0)   execution time : 38.383 s
Press any key to continue.
```

Exemplul de rulare pentru 100.000.000 elemente:



```
"D:\documente facultate\Anul III\semestrul 2\APD\MergeSort\APD-MergeSort\APD-MergeSort\bin\Debug\APD-MergeSort.exe"
Initial memory used : 57%

Enter the number of elements: 100000000
An array of 100.000.000 integers is generated and saved in <sortExpected.bin> file.
The above array is randomized and saved in <unsorted.bin> file.

Read from file the standard array which is ascending sorted.

Read from file the unsorted array with a size of 100.000.000 integers.

Memory used : 95%
Time for sorting a list (double linked list) with implicit sort() method in C++: 76851 ms.

Memory used : 94%
Time for sorting a forward_list (single linked list) with implicit sort() method in C++: 80744 ms.

Array of 100.000.000 elements is merge sorting with sequentially recursive algorithm
Memory used : 93%
Time : 42123 ms.
Array was sorted as expected.

Array of 100.000.000 elements is merge sorting sequentially iterative algorithm
Memory used : 84%
Time : 40911 ms.
Array was sorted as expected.

Array of 100.000.000 elements is merge sorting parallel recursive algorithm
Memory used : 90%
Time : 26672 ms.
Array was sorted as expected.

Array of 100.000.000 elements is merge sorting parallel iterative algorithm
Memory used : 89%
Time : 26059 ms.
Array was sorted as expected.


Process returned 0 (0x0)   execution time : 440.532 s
Press any key to continue.
```

Pentru un numar mai mare de elemente, listele standard (list si forward_list) s-au blocat din cauza memoriei insuficiente. Din cauza timpului ridicat, nu au fost continuate nici testele pentru algoritmii implementati, insa concluziile obtinute pana la testele de pana la 10^8 elemente sunt relevante pentru a concluziona cu privire la eficienta algoritmilor vizati.

Fisierele in care au fost salvate rezultatele rularii codurilor

Prezint in continuare fisierul (extras) pentru rularea pe un calculator cu procesor AMD Ryzen 5, 6 core/12 threads, 4.20 GHz; RAM: 8 GB:

*tests.txt - Notepad

File Edit Format View Help

```
Type, Size, Threads, InitialMemory, FinalMemory, Time[ms]
List,100000,,35,35,31724
ForwardList,100000,,35,35,15690
SecventiallyRecursive,100000,,35,35,31297
SecventiallyIterative,100000,,35,35,15625
ParallelRecursive,100000,12,35,35,15674
IterativeRecursive,100000,12,35,35,15377
List,300000,,29,29,93845
ForwardList,300000,,29,29,125060
SecventiallyRecursive,300000,,29,29,62536
SecventiallyIterative,300000,,29,29,62936
ParallelRecursive,300000,12,29,29,15607
IterativeRecursive,300000,12,29,29,15969
List,100000000,,29,78,37419514
ForwardList,100000000,,29,72,60893631
SecventiallyRecursive,100000000,,29,79,23270014
SecventiallyIterative,100000000,,29,83,23603100
ParallelRecursive,100000000,12,29,82,14823412
IterativeRecursive,100000000,12,29,83,6854445
List,1000000,,21,22,500785
ForwardList,1000000,,21,22,752854
SecventiallyRecursive,1000000,,21,22,344358
SecventiallyIterative,1000000,,21,22,330000
ParallelRecursive,1000000,12,21,22,125401
IterativeRecursive,1000000,12,21,22,109609
List,10000000,,21,26,5418561
ForwardList,10000000,,21,31,5648038
SecventiallyRecursive,10000000,,21,31,2155035
SecventiallyIterative,10000000,,21,32,2206738
ParallelRecursive,10000000,12,21,32,598496
IterativeRecursive,10000000,12,21,33,719552
AMD Ryzen 5 4500U with 6 core, 4G
```

Prezint in continuare fisierul (extras) pentru rularea pe un calculator cu procesor Intel i5, 2 core, 2.30 GHz; RAM: 8 GB:

*tests.csv - Notepad

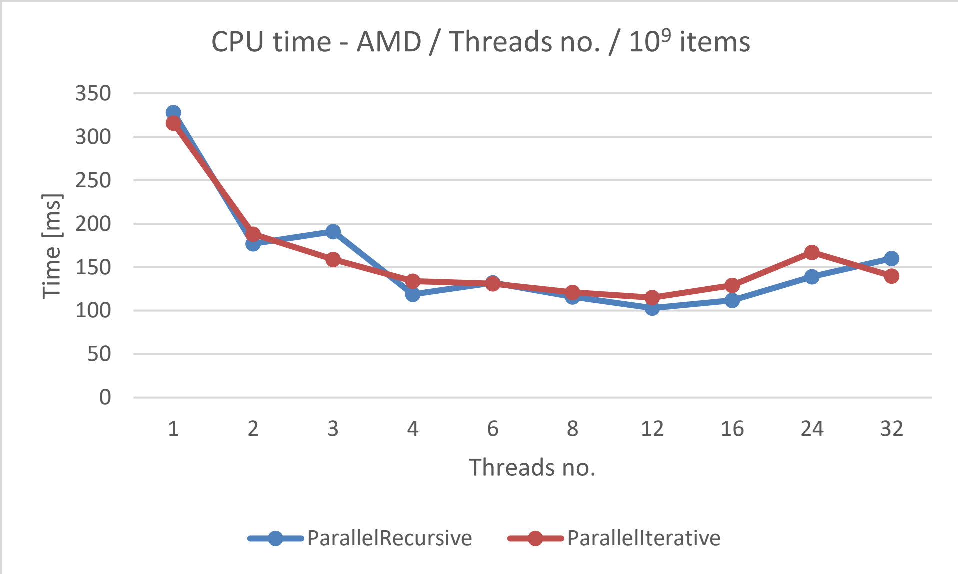
File Edit Format View Help

```
Type, Size, Threads, InitialMemory, FinalMemory, Time[ms]
List,100000000,,57,92,58491
ForwardList,100000000,,57,85,58948
SecventiallyRecursive,100000000,,57,82,34991
SecventiallyIterative,100000000,,57,90,35420
ParallelRecursive,100000000,2,57,88,27147
ParallelRecursive,100000000,2,57,88,26927
List,1000000,,75,76,537
ForwardList,1000000,,75,76,479
SecventiallyRecursive,1000000,,75,76,371
SecventiallyIterative,1000000,,75,76,343
ParallelRecursive,1000000,4,75,76,194
ParallelRecursive,1000000,4,75,76,200
List,1000000,,64,65,551
ForwardList,1000000,,64,65,460
SecventiallyRecursive,1000000,,64,66,373
SecventiallyIterative,1000000,,64,66,353
ParallelRecursive,1000000,4,64,66,192
ParallelRecursive,1000000,4,64,66,200
List,10000000,,65,70,5901
ForwardList,10000000,,65,74,5497
SecventiallyRecursive,10000000,,65,74,3734
SecventiallyIterative,10000000,,65,75,3695
ParallelRecursive,10000000,4,65,75,1969
ParallelRecursive,10000000,4,65,76,2121
List,100000000,,66,95,69949
ForwardList,100000000,,66,89,86366
SecventiallyRecursive,100000000,,66,91,64416
SecventiallyIterative,100000000,,66,80,44602
ParallelRecursive,100000000,4,66,84,20736
ParallelRecursive,100000000,4,66,88,21409
List,100000000,,57,95,76851
ForwardList,100000000,,57,94,80744
SecventiallyRecursive,100000000,,57,93,42123
SecventiallyIterative,100000000,,57,84,40911
ParallelRecursive,100000000,4,57,90,26672
ParallelRecursive,100000000,4,57,89,26059
```

Rezultatele testarii si concluzii

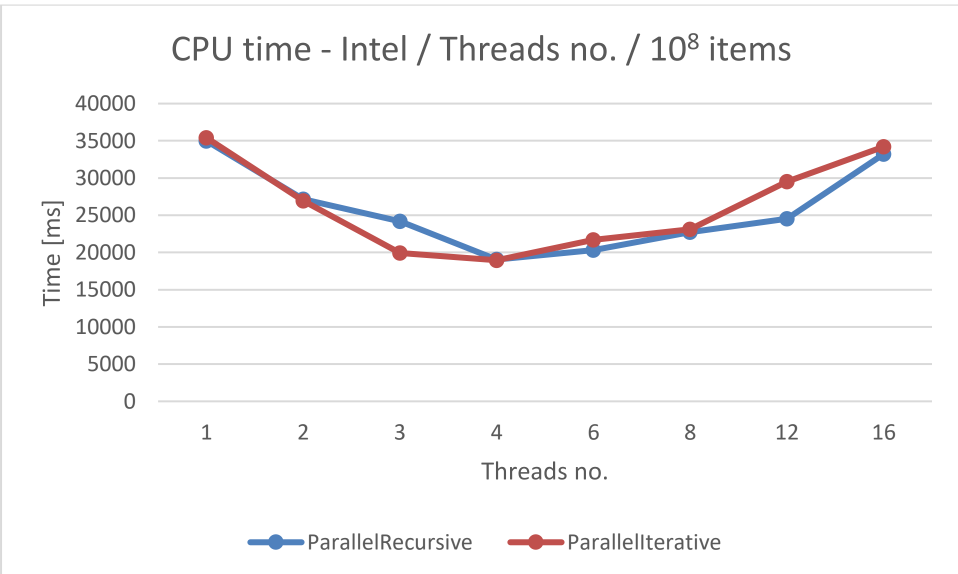
Primul set de teste a vizat comportamentul algoritmilor paraleli ruland pe mai multe fire de executie.

Prezint mai jos rezultatele obtinute pe calculatorul cu procesor AMD Ryzen 5, 6 core/12 threads, 4.20 GHz, ruland algoritmii de sortare merge-sort pe un set de test de 10^9 elemente:



Algorithm type	Time[ms] - CPU: AMD Ryzen 5, 6 core/12 threads, 4.20 GHz; RAM: 8 GB									
ParallelRecursive	328	177	191	119	132	116	103	112	139	160
ParallelIterative	316	188	159	134	131	121	115	129	167	140
Threads	1	2	3	4	6	8	12	16	24	32

Mai jos prezint rezultatele obtinute pe calculatorul cu procesor Intel i5, 2 core/4 threads, 2.30 GHz, ruland algoritmii de sortare merge-sort pe un set de test de 10^8 elemente:

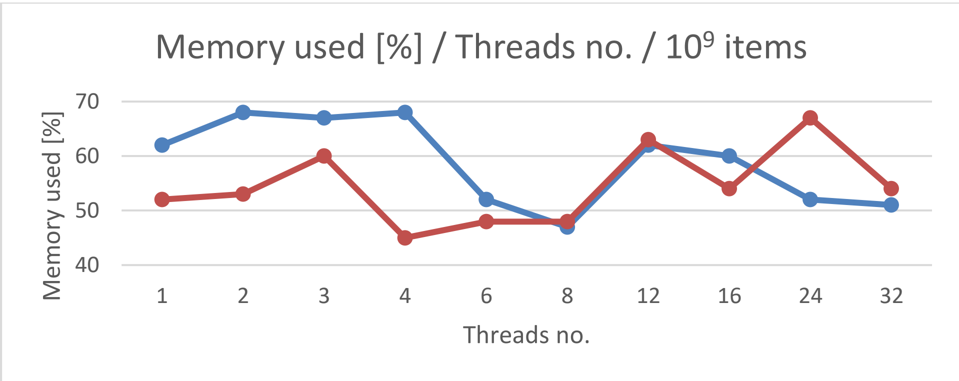


Algorithm type	Time [ms] - CPU: Intel i5, 2 core/4 threads, 2.30 GHz; RAM: 8 GB							
ParallelRecursive	34,991	27,147	24,173	19,052	20,311	22,708	24,530	33,221
ParallelIterative	35,420	26,927	19,916	18,957	21,694	23,094	29,500	34,201
Threads	1	2	3	4	6	8	12	16

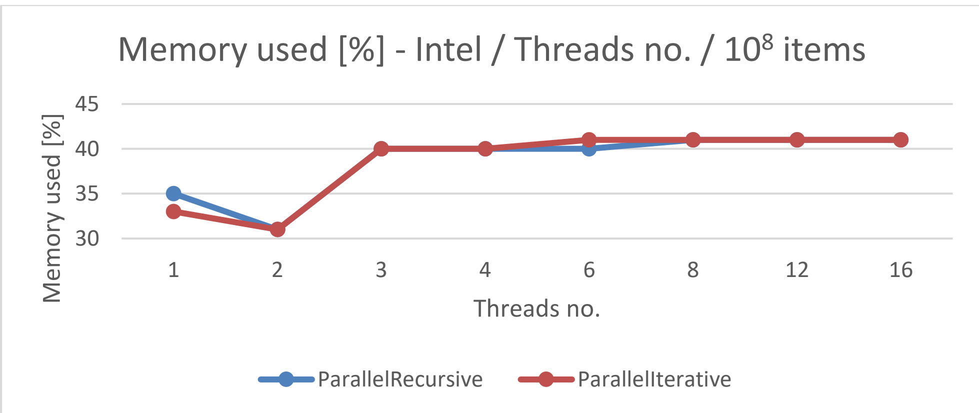
Dupa cum se observa din graficele si tabelele de mai sus, cele mai bune performante, din punct de vedere al timpului/vitezei, se obtin atunci cand ***numarul firelor de executie pentru rularea algoritmilor este egal cu cel din specificatiile tehnice ale procesorului***.

O alta concluzie este aceea ca, pentru algoritmii analizati, ***cresterea numarului de fire de executie, peste cel din specificatiile procesorului, nu aduce nici o imbunatatire a timpului de executie***, ba din contra, acesta creste, deoarece sunt necesare mai multe intreruperi ale firului de executie curent si transferal executiei catre alt fir de executie, odata cu salvarea starii respectivelor fire de executie.

In ceea ce priveste memoria utilizata, situatia a fost urmatoarea:



Algorithm type	Memory used [%] - CPU: AMD Ryzen 5, 6 core/12 threads, 4.20 GHz; RAM: 8 GB									
ParallelRecursive	62	68	67	68	52	47	62	60	52	51
ParallelIterative	52	53	60	45	48	48	63	54	67	54
Threads	1	2	3	4	6	8	12	16	24	32

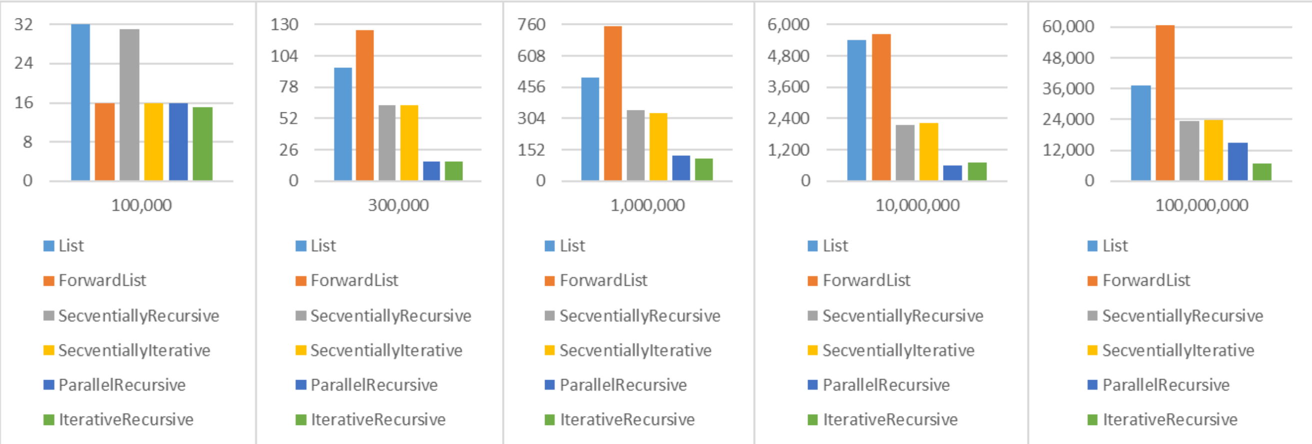


Algorithm type	Memory used [%] - CPU: Intel i5, 2 core/4 threads, 2.30 GHz; RAM: 8 GB							
ParallelRecursive	35	31	40	40	40	41	41	41
ParallelIterative	33	31	40	40	41	41	41	41
Threads	1	2	3	4	6	8	12	16

Referitor la utilizarea memoriei, cea mai eficienta alocare este in zona in care numarul firelor de executie ale algoritmilor paraleli este egal cu cel al miezurilor (core-urile) procesoarelor, deoarece fiecare fir de executie ruleaza pe un singur miez si nu mai este necesara salvarea starilor firelor de executie alternante.

Al doilea set de teste a vizat analiza comparativa a eficientei algoritmilor paraleli, raportat la cei secventiali, pe de o parte si la algoritmii standard ai limbajului C++ pentru liste dublu inlantuite (list) si liste simplu inlantuite (forwardlist), pe de alta parte.

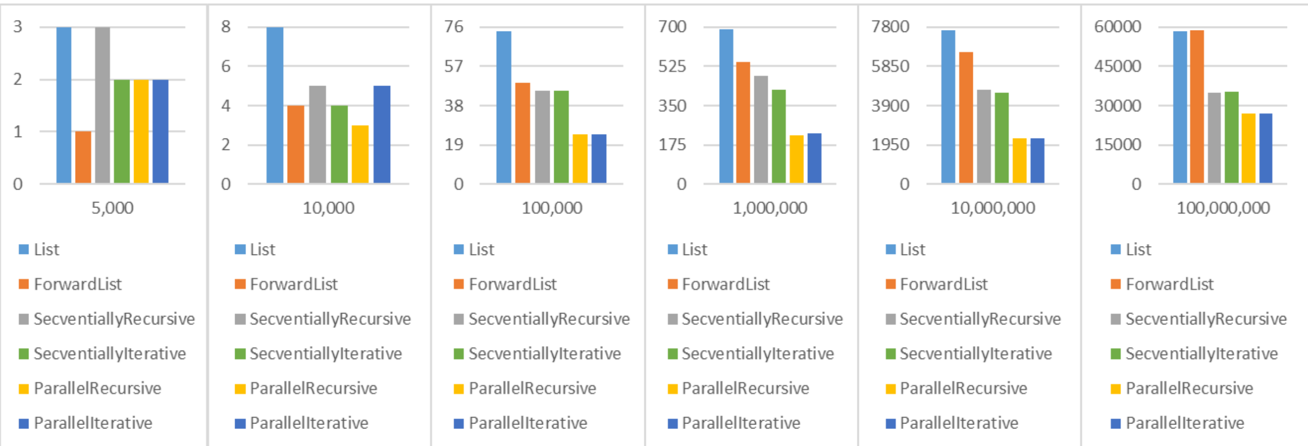
- Pentru procesorul AMD:



Algorithm type	Time[ms] - CPU: AMD Ryzen 5, 6 core/12 threads, 4.20 GHz; RAM: 8 GB				
List	32	94	501	5,419	37,420
ForwardList	16	125	753	5,648	60,894
SecventiallyRecursive	31	63	344	2,155	23,270
SecventiallyIterative	16	63	330	2,207	23,603
ParallelRecursive	16	16	125	598	14,823
IterativeRecursive	15	16	110	720	6,854
Items no.	100,000	300,000	1,000,000	10,000,000	100,000,000

Algorithm type	Memory used [%] - CPU: AMD Ryzen 5, 6 core/12 threads, 4.20 GHz; RAM: 8 GB				
List	35	29	22	26	78
ForwardList	35	29	22	31	72
SecventiallyRecursive	35	29	22	31	79
SecventiallyIterative	35	29	22	32	83
ParallelRecursive	35	29	22	32	82
IterativeRecursive	35	29	22	33	83
Items	100,000	300,000	1,000,000	10,000,000	100,000,000

- Pentru procesorul Intel:



Items	Time [ms] - CPU: Intel i5, 2 core/4 threads, 2.30 GHz; RAM: 8 GB					
List	3	8	74	690	7,651	58,491
ForwardList	1	4	49	543	6,567	58,948
SecventiallyRecursive	3	5	45	482	4,695	34,991
SecventiallyIterative	2	4	45	420	4,542	35,420
ParallelRecursive	2	3	24	216	2,279	27,147
ParallelIterative	2	5	24	226	2,294	26,927
Items no.	5,000	10,000	100,000	1,000,000	10,000,000	100,000,000

Items	Memory used [%] - CPU: Intel i5, 2 core/4 threads, 2.30 GHz; RAM: 8 GB					
List	35	35	35	36	40	72
ForwardList	35	35	35	36	44	65
SecventiallyRecursive	35	35	35	37	45	62
SecventiallyIterative	35	35	35	37	45	70
ParallelRecursive	35	35	35	37	46	68
ParallelRecursive	35	35	35	37	46	68
Items no.	5,000	10,000	100,000	1,000,000	10,000,000	100,000,000

Asa cum se observa din graficele si tabelel de mai sus, ***algoritmii paraleli*** implementati, ***sunt*** cu 2 pana la 4 ori ***mai rapizi decat cei*** implementati ***in varianta secventiala, precum*** si de 2 pana la 6 ori mai rapizi ***decat cei oferiti standard de program***.

La valori mici ale numarului de elemente care trebuie sortate, diferentele de viteza sunt mai mici sau inexistente intre algoritmi, insa ***la valori mari ale numarului de elemente care trebuie sortate, algoritmii implementati sunt mult mai rapizi***, dintre acestia, cei paraleli fiind de asemenea mai rapizi decat cei secventiali.

Cu privire la alocarea memoriei***, cu cat creste numarul de elemente care trebuie sortate, creste si memoria RAM ocupata***.

Testele s-au oprit la 10^8 elemente care au constituit date de intrare pentru algoritmii testate, deoarece listele oferite de standardul C++ (list si forwardlist) aveau nevoie de memrie RAM mai multa decat cele ale sistemelor pe care s-au realizat testele.

Concluzia finala este ca ***atunci cand timpul de executie este critic, se impune implementarea unor algoritmi propri de sortare***, inclusiv utilizand metode de calcul paralel.

Bibliografie:

1. https://en.wikipedia.org/wiki/Merge_sort
2. https://wiki.codeblocks.org/index.php/Basic_Tutorial
3. Cursuri si laboratoare ACE-UCV

Fisierele cu rezultatele de test, se regasesc in folderul files.