Sortarea prin interclasare, în serie și în paralel. Studiu comparativ

Vlad-Ion Barbu

vlad.barbu99@e-uvt.ro

Universitatea de Vest Timișoara
Facultatea de Matematică și Informatică

Rezumat

Sortarea înseamnă ordonarea unor elemente după niște criterii prestabilite, oferind datelor calitatea de a fi ordonate. Astfel operații precum: căutarea sau indexarea devin mult mai facile în contextul accesării bazelor de date. Procesarea în paralel a devenit un punct cheie în calculul computațional. Odată cu creșterea volumului de date, a crescut și nevoia de putere de procesare. Pentru o performanță crescută putem sorta în paralel. Acestă lucrare își propune să compare două implementări ale aceluiași algoritm: în serie și în paralel. Algoritmul ales este Merge Sort, ușor de executat în paralel. Vor fi analizate teoria, pseudocodul și implementarea în Python.

1 Introducere

Aranjarea unor obiecte a fost dintotdeauna un subiect ce a intrigat mintea umană, ajungând să fie parte din modul nostru de gândire rațională. În informatică aranjarea pe categorii, sau ordonarea unor elemente, se numește sortare. Sortarea reprezintă punctul de start în rezolvarea unor probleme, de exemplu: căutarea, găsirea unei perechi de obiecte, frecvența distribuirii elementelor în complexitate O($\log(n)$); sau: unicitatea unui element, selectarea elementului k în complexitate O(n).[13]

De-a lungul timpului volumul de date procesate a crescut considerabil. Industria tehnologiei informației s-a adaptat fenomenului în două direcții: software și hardware. Din punct de vedere software, algoritmii de sortare au evoluat de la timp de execuție în $O(n^2)$ (Bubble sort, Insertion sort etc.) la $O(n \log(n))$ (Quick sort, Merge sort etc.)[9]. Pe partea de hardware lucrurile sunt încă și mai interesante, atunci când vorbim despre procesoare în mod special. În 1965 Gordon Moore (CEO al Intel) făcea predicția că numărul de tranzistori dintr-un circuit integrat se va dubla odată la (aproximativ) 2 ani[12]. În 2015 CEO-ul Intel anunță că: "Ritmul actual este mai aproape de doi ani și jumătate decât doi."[10]. În 2019 compania AMD anunță lansarea pe piață a unui procesor pe tehnologie de 7nm[14]. Dimensiunile din ce în ce mai reduse ale tranzistorilor ating limitări în ceea ce privește fizica cuantică(efectul de tunel) [15].

Astfel, o altă modalitate de a crește performanța este fabricarea procesoarelor multi-core, în esență o singură componentă, UCP-ul, având 2 sau mai multe unități independente de procesare, numite nuclee. De la primul

dual-core comercial (IBM Power4[19]), astăzi sunt disponibile pe piață procesoare cu până la 32 de nuclee (de exemplu: AMD Ryzen Threadripper 2990WX[18]).

În concluzie procesarea în paralel devine o necesitate, iar sortarea în acest mod este extrem de relevantă. În acesta lucrare autorul își propune să studieze performanța, într-un mediu multi-core, al unui algoritm de complexitate $O(n \log(n))$. Algoritmul ales este Merge sort, se va compara implementarea în serie cu cea în paralel. Sortarea se va face pe liste de numere naturale, de la 0 la n, numerele fiind aranjate la întâmplare; dar pentru aceeași listă (generată la întâmplare) se vor testa ambele implementări, pentru a nu există discrepanțe.

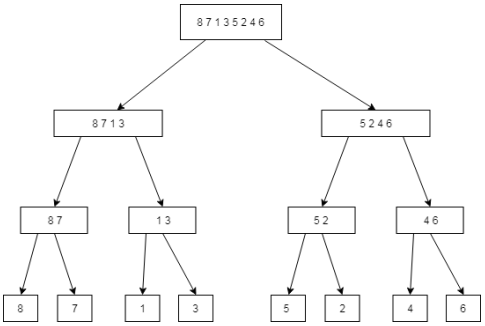
2 Prezentarea formală a algoritmului

Merge sort este un algoritm eficient de sortare bazat pe compararea elementelor. Sortarea rezultată este stabilă, ordinea relativă a elementelor rămâne neschimbată. Este bazat pe tehnică Divide And Conquer, merge sort a fost inventat în 1945 de John von Neumann[7].

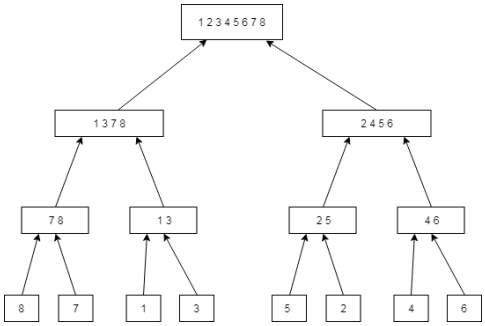
Merge sort în serie:
Algoritmul funcționează în modul următor:

1. Divide lista (de sortat) în n subliste, fiecare de dimensiune 1 (un element). O lista de 1 element este considerată sortată (cazul trivial).
2. Îmbină (merge) recursiv sublistele în subliste noi sortate, până când rămâne o singură sublistă. Lista rezultată este cea originală, sortată.

Vom folosi implementarea de tip *bottom-up*(Vezi figurile de mai sus): sublistele obținute prin 'Divide' sunt îmbinate într-o lista sortată (proces recursiv)
În pseudocod, varianta în serie:



(a) Divide



(b) Îmbină

Function `Merge`$(l1, l2)$**:**

$index1 \longleftarrow 0$;
$index2 \longleftarrow 0$;
$contor \longleftarrow 0$;
while $index1 < len(l1)$ & $index2 < len(l2)$ **do**
  if $l1[index1] < l2[index2]$ **then**
    $f[contor] \longleftarrow l1[index1]$;
    $index1 \longleftarrow index1 + 1$;
    $contor \longleftarrow contor + 1$;
  else
    $f[contor] \longleftarrow l2[index2]$;
    $index2 \longleftarrow index2 + 1$;
    $contor \longleftarrow contor + 1$;
  end
end
while $index1 <= len(l1)$ **do**
  $f[contor] \longleftarrow l1[index1]$;
  $index1 \longleftarrow index1 + 1$;
  $contor \longleftarrow contor + 1$;
end
while $index2 <= len(l2)$ **do**
  $f[contor] \longleftarrow l2[index2]$;
  $index2 \longleftarrow index2 + 1$;
  $contor \longleftarrow contor + 1$;
end
return f;

Function `MergeSort(`l`):`

```
if len(l) <= 1 then
    return l;
end
l_stanga ⟵ MergeSort(jumtatea-stanga);
l_dreapta ⟵MergeSort(jumtatea-dreapta);
return Merge(l_stanga, l_dreapta);
```

Merge sort în paralel:
Sortarea este, în teorie, ușor de implementat în paralel, dată fiind natura divide-and-conquer a algoritmului. Ideea de bază ar fi: divide și împarte fiecărui procesor sarcinile, apoi rezolvă-le în *paralel*. În pseudocodul de mai jos se folosește un *thread pool*[8][6] pentru a împărții aceste sarcini. Funcțiile **Merge** și **MergeSort** rămân la fel ca la implementarea în serie.

```
Function MergeSortParalel(l):
    np ⟵ nr_de_procesoare_disponibile;
    dimesiune_partiție ⟵ len(l)/n;
    for i ← 0 to np do
        contor ←0;
        for j ← (dimesiune_partiție ∗ i) to
         (i + 1) ∗ dimesiune_partiție do
            l_partiție[i][contor] ←l[j];
            contor ←contor+1;
        end
    end
    Pentru fiecare 'procesor' din −pool− apeleaza
     MergeSort(l_partiție[i]);
    nr_partiții ←np;
    i←0;
    while nr_partiții > 1 do
        j←0;
        while i < nr_partitii do
            l[j] ←Merge(l_partiție[i], l_partiție[i+1]);
            j ← j+1 ;
            i ← i+2 ;
        end
        nr_partiții ←nr_partiții/2;
    end
    return l_partiție[0];
```

Pseudocodul de mai sus execută 3 pași:

1. Împarte lista de sortat în partiții de dimensiune egală, numărul de partiții este egal cu numărul de procesoare/unități de prelucrare.

2. Pentru fiecare procesor(unitate de prelucrare) din *pool* asociază o partiție,

și execută MergeSort pe ea.

3. Îmbină partițiile sortate înapoi într-o singură listă.

În imaginea de mai jos putem observa cum sunt împărțite sarcinile procesoarelor. Am considerat o listă de 8 elemente și 4 fire de execuție.

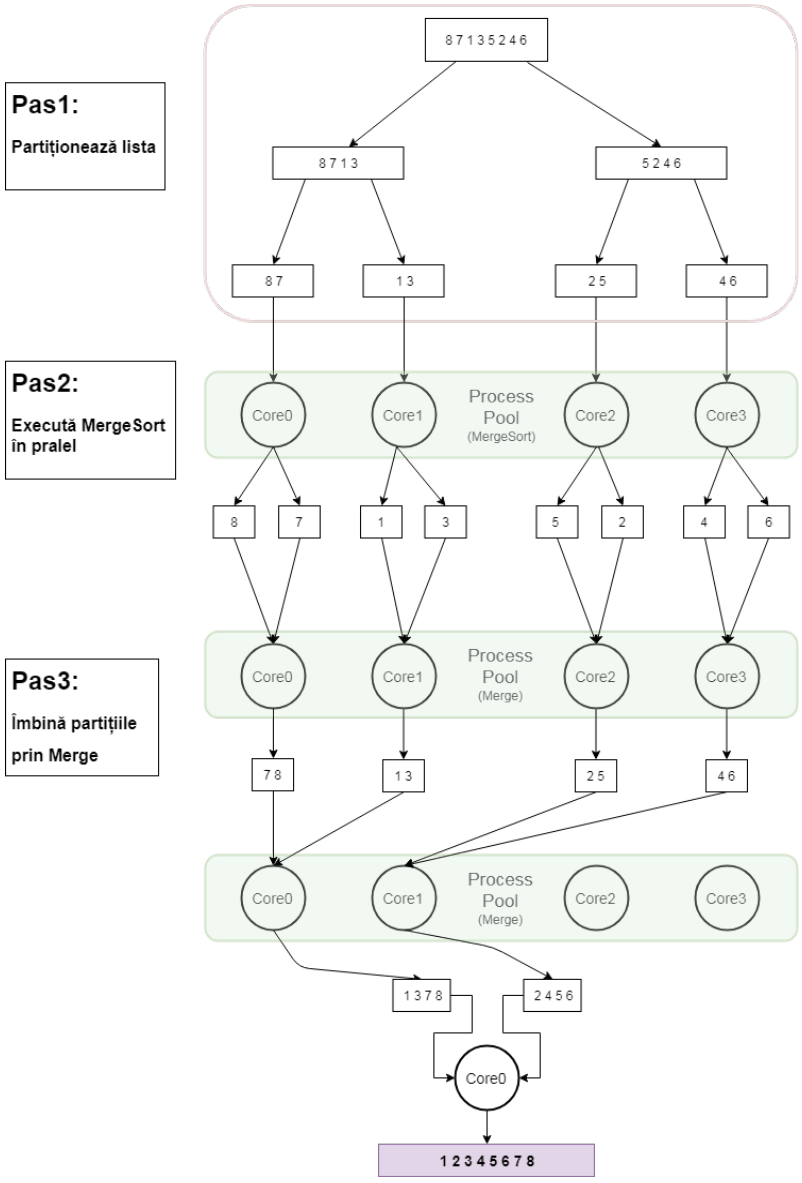


Figura 1: MergeSortParalel 8,7,1,3,5,2,4,6-Pool(4 nuclee)-1,2,3,4,5,6,7,8

Varianta în pseudocod a algoritmului poate aduce puțină confuzie datorită folosirii noțiunii de *thread pool*, implementarea fiind strâns legată de limbajul de programare. Un *thread pool*, sau *process pool*, este un model software pentru obținerea unui program ce rulează concurent/paralel. Explicarea în detaliu a noțiunii este dincolo de scopul acestei lucrări, recomand consultarea literaturii [17][20] [3]. În secțiunea următoare 3, este prezentată o variantă în limbajul Python, folosind un obiect *Pool*[5].

3 Implementarea în limbajul Python

În continuare puteți găsi varianta, în limbajul Python, corespunzătoare pseudocodului de la secțiunea 2. Codul a fost scris și testat în *Python3.6*, acesta nu va rula pentru versiuni mai vechi de *3.4*, deoarece a fost folosită biblioteca **multiprocessing** [5] pentru a accesa capacitățile concurente[4] ale limbajului. Codul complet poate fi găsit ***aici***.

Functia MergeSort

```
def merge_sort(l):
    '''Functionalitate:
    1)Divide lista in subliste de dimensiune 1
    2)Apleleaza merge() pentru a le imbina intr-o lista sortata
    '''
    if len(l)<=1:
        return l

    l_stanga = merge_sort(l[:len(l)//2])
    #l_stanga = l[0...mijloc]

    l_dreapta = merge_sort(l[len(l)//2:])
    #l_dreapta = l[mijloc...final]
    return merge(l_stanga, l_dreapta) #imbina listele
```

Functia Merge

```
def merge(l1, l2):
    '''Functia 'imbina' cele doua liste (l1 si l2) prin sortate
    '''
    f=[]#lista sortata returnata
    l1_index=0;#iterator in lista l1
    l2_index=0;#iterator in lista l2

```

```
    #compar unul cate unul elementele din liste si memorez pe cel mai mic
    while l1_index<len(l1) and l2_index<len(l2):
        if l1[l1_index] < l2[l2_index]:
            f.append(l1[l1_index])
            l1_index+=1#incrementez iteratorul
        else: # l1[l1_index] > l2[l2_index]
            f.append(l2[l2_index])
            l2_index+=1
    #memorez eventualele elemente ramase, necomparate:
    while l1_index <= (len(l1)-1):#elemente necititie in l1
        f.append(l1[l1_index])
        l1_index+=1

    while l2_index <= (len(l2)-1):#elemente necititie in l2
        f.append(l2[l2_index])
        l2_index+=1
    return f
```

Functia MergeSortParallel

```
def merge_sort_parallel(l):
    '''Creeaza un pool de procese, egal cu nr. de nuclee
    Separa lista in partitii, de marimi egale, pentru fiecare worker\proces din pool,
    apoi executa un merge_sort pentru fiecare partitie
    La final se 'imbina' partitiile astefel sortate'''
    import multiprocessing
    np=multiprocessing.cpu_count()#np =  numarul de procesoare/nuclee
    size = int(math.ceil(float(len(l)) / np )) # dimensiunea fiecarei partitii aproximate superior
    l_partitii=[l[i * size:(i + 1) * size] for i in range(np)]#impartirea in partitii a listei l

    pool = multiprocessing.Pool(processes=np) #creem un obiect pool de fire de executie
    l_partitii = pool.map(merge_sort, l_partitii)#sortam prin fiecare partitie
   #trebuie sa imbinam partitiile obtinute
    while len(l_partitii) > 1:
        pereche = [(l_partitii[i], l_partitii[i+1]) for i in range(0, len(l_partitii), 2)]#formez perechi de cate 2 partitii
        l_partitii = pool.map(mergeWrap, pereche)
    return l_partitii[0]#l_partitii=[ [unica partitie] ] i.e:[0]
```

Funcția *merge_sort_parallel* va returna un rezultat complet doar pentru un număr de elemente divizibil cu *np* (nr.de procesoare). Amănunt neesențial în demonstrarea conceptului.

4 Rezultate empirice și discuție

Mediul de test:

Algoritmul testat este implementat în limbajul *Python 3.6 (64-bit)*, vezi secțiunea 3, pe un procesor *Intel® Core™ i5-4570S*, la frecvența de baza de 2.90 GHz, folosind numărul total de nuclee disponibile (patru). Sistemul de operare folosit este *Windows 7 (64-bit)*.

Modul de testare:

Testele au fost executate pe liste de numere naturale din intervalul 0 până la n, elementele fiind aranjate la întâmplare. Acuratețea măsurării timpului de execuție este oferită de modulul **time**[11], din biblioteca standard *CPython*. Pentru a garanta o măsurare imparțiala și corectă sortarea în *serie* și cea în *paralel* au fost testate *pe aceeași lista generată* (la întâmplare) *inițial*. Rezultatele sunt exprimate în *secunde* și reprezintă *media aritmetică* a **trei** iterări. Codul complet poate fi găsit ***aici***.

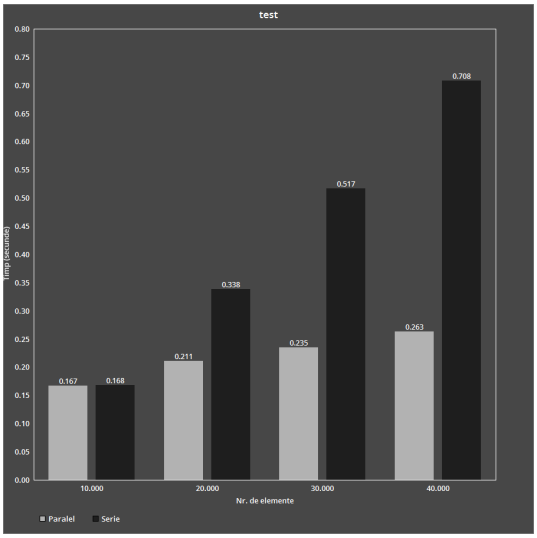
Rezultate empirice:

Rezultate în medie		
Nr. de elemente	**Serie**	**Paralel**
100	0.001	0.153
1.000	0.0143	0.144
10.000	0.163	0.172
100.000	1.868	0.454
1.000.000	21.537	3.881

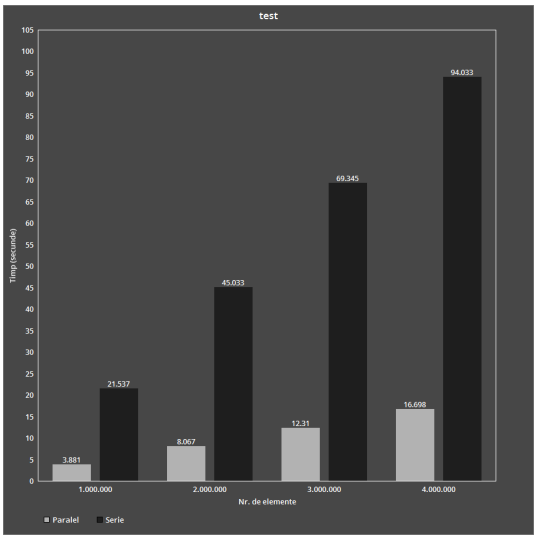
Tabela 1: Rezultate pentru liste mici, medii și mari (**în secunde**)

Rezultate în detaliu:

În diagramele de mai jos culoarea *gri* este timpul de execuție în **paralel**, iar *negru* este timpul de execuție în **serie**



(a) 10.000-40.000 elemente



(b) 1.000.000-4.000.000 elemente

Figura 2: Diagrame cu timpii de execuție pentru liste mari

Discuție

Am realizat un studiu comparativ a algoritmului Merge Sort în două variante: serie(pe un singur fir de execuție) și paralel(pe 4 fire de execuție). După ce am experimentat pe liste de numere predefinite(100, 1000, 10000, 100000, 1000000), am constatat următoarele:
Conform Tabelei 1: pentru liste mici (100-10.000 elemente) execuția în serie este mai rapidă (în special pentru liste mai mici de 1000 de elemente). La 10.000 de numere observăm că cele două implementări sunt aproape egale. Pentru dimensiuni mari (peste 100.000) varianta paralelă este mai rapidă. Pentru liste de peste 1.000.000 (figura 2 b) de elemente observăm că în paralel algoritmul este de (aproximativ) 5,5 ori mai rapid ca execuția în serie. Având timpii de execuție putem calcula *de câte ori este mai rapid merge sort* ***în paralel*** *decât merge sort* ***în serie***, aflând x din ecuația:
timp_paralel * x = **timp_serie**.

- 100 elemente: Varianta în paralel este de **0,006** ori mai rapidă ca cea în serie.
- 1000 elemente: Varianta în paralel este de **0,099** ori mai rapidă ca cea în serie.
- 10.000 elemente: Varianta în paralel este de **0,94** ori mai rapidă ca cea în serie.
- 100.000 elemente: Varianta în paralel este de **4,1** ori mai rapidă ca cea în serie.
- 1.000.000 elemente: Varianta în paralel este de **5,55** ori mai rapidă ca cea în serie.

Care sunt punctele slabe în soluția propusă?
Un amănunt important atunci când vorbim de acuratețea rezultatelor: **Se pot face îmbunătățiri?**, sau: **Există o abordare mai pertinentă?** Răspuns: *da* și *da*.

La implementarea în Python, din secțiunea 3, am folosit biblioteca **multiprocessing** și un obiect **Pool** prin care am accesat potențialul paralel al limbajului. Aici modulul multiprocessing a fost utilizat ca **black-box** (*cutie neagră*) în ceea ce privește modul de gestionare al procesorului. Astfel am creat o implementare mai ușor de înțeles la nivel tehnic, dar am renunțat la mare parte din controlul accesului la UCP.

Am utilizat ca sistem de operare **Windows 7**, care se ocupă de crearea și gestionarea proceselor ce accesează UCP-ul [1]. Fiind un sistem de operare cu kernel hibrid există[16][1] procese de fundal ce consumă unele resurse. Aceste lucruri ar putea fi o consecință a faptului că în paralel sortarea este mai înceată pentru seturi de numere cu puține elemente(sub 1000).

Procesorul pe care s-au efectuat testele are 4 nuclee, ne putem pune întrebarea: cum se comportă algoritmul la un număr mai mare de fire de execuție? Cum crește performanță? Cu alte cuvinte: **este implementarea scalabilă**?[2]. Toate aceste întrebări merită studiate într-o lucrare viitoare.

5 Concluzii și direcții viitoare

Am ajuns la concluzia că Sortarea prin Merge sort este mai rapidă în paralel decât în serie (pentru un UCP cu 4 fire de execuție) pentru liste de peste 10.000 de elemente, ajungând să fie de până la 4 ori mai rapidă pentru liste de peste 100.000 de elemente, de 5,5 ori mai rapidă pentru liste de peste 1.000.000 de elemente. Pentru seturi de numere mici (sub 10.000 elemente) sortarea în serie este mai rapidă.
Am menționat faptul că platforma de test are o importanță majoră în executarea experimentului, performanța procesorului, sistemul de operare cât și limbajul de programare joacă un rol esențial în obținerea rezultatelor.

Problema sortării în paralel

Problema sortării nu este una nouă în informatică, experții din domeniu au studiat și au realizat o multitudine de articole și lucrări în acest sens. Și sortarea în paralel a reprezentat tema unor studii efectuate în vederea obținerii unei sortări mai rapide. MergeSort în paralel a fost abordat și de alți autori, de exemplu Cormen, Leiserson, Rivest, și Stein în cartea *Introduction to Algorithms*[3] propun câteva variante paralele a acestui algoritm. O abordare ce merită studiată este sortarea prin placa video (GPU), acesta având sute sau mii de procesoare specializate pentru calcul paralel.
Literatura nu a reușit să ofere un răspuns final problemei sortării.

Direcții viitoare

Prin acestă lucrare am demonstrat utilitatea sortării în paralel, subiectul rămâne unul deschis pentru domeniul informaticii.
Datorită evoluției hardware, procesarea concurentă/distribuită/paralelă capătă mai multă importantă ca oricând, astfel apare necesitatea adaptării algoritmilor la nouă generație de hardware.

> "*Procesoarele moderne sunt ca niște mașini alimentate cu nitro, excelează la curse în linie dreaptă. Din păcate limbajele de programare moderne sunt ca Monte Carlo, pline de întorsături și curbe*" - **David Ungar**

Bibliografie

[1] Peter Baer Galvin și Greg Gagne Abraham Silberschatz. *Operating system concepts*. 2013. ISBN: 9781118063330.

[2] André B. Bondi. *Characteristics of scalability and their impact on performance*. 2000. DOI: https://www.win.tue.nl/~johanl/educ/2II45/2010/Lit/Scalability-bondi\%202000.pdf.

[3] Thomas Cormen. *INTRODUCTION TO ALGORITHMS, THIRD EDITION*. 2009. Chap. 27 Multithreaded Algorithms, pp. 772–775. ISBN: 978-0-262-03384-8.

[4] Python software foundation. *17. Concurrent Execution*. URL: https://docs.python.org/3.4/library/concurrency.html.

[5] Python software foundation. *17.2. multiprocessing — Process-based parallelism*. URL: https://docs.python.org/3.4/library/multiprocessing.html.

[6] Brian Goetz. *Thread pools and work queues*. URL: https://www.ibm.com/developerworks/java/library/j-jtp0730/index.html.

[7] J. von Neumann H.H. Goldstine. "Planning and coding of problems for an electronic computing instrument". In: *John yon Neumann Collected Works, Volume V: Design of Computers, Theory of Automata and Numerical Analysis* (1963), pp. 152–214. DOI: https://library.ias.edu/files/pdfs/ecp/planningcodingof0103inst.pdf.

[8] Rajat P. Garg și Ilya Sharapov. *Techniques for Optimizing Applications: High Performance Computing*. 2001. Chap. CHAPTER 12 Optimization of Explicitly Threaded Programs, p. 394.

[9] Donald Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition*. 1998. Chap. 5-Sorting, pp. 80–168. ISBN: 0-201-89685-0.

[10] Brian Krzanich. *The last two technology transitions have signaled that our cadence today is closer to two and a half years than two*. URL: https://www.businessinsider.com/intel-ceo-brian-krzanich-suggests-moores-law-is-over-2015-7.

[11] Gabriele Lanaro. *Python High Performance, Second edition*. Packt Publishing, 2017. Chap. Chapter 1: Benchmarking and Profiling.

[12] Gordon Moore. “Cramming more components onto integrated circuits”. In: *Electronics Magazine* (2006 (Republicat)). DOI: http://www.cs.utexas.edu/~fussell/courses/cs352h/papers/moore.pdf.

[13] Steven S. Skiena. *The Algorithm Design Manual*. Springer-Verlag London Limited, 2008, pp. 104–105. ISBN: 978-1-84800-069-8.

[14] Stephen Nellis Sonam Rai. *AMD shows off 7nm next-gen chips at CES*. URL: https://www.reuters.com/article/us-tech-ces-amd/amd-shows-off-7nm-next-gen-chips-at-ces-aims-at-intel-and-nvidia-idUSKCN1P32DC.

[15] Ed Sperling. *Quantum Effects At 7/5nm And Beyond*. URL: https://semiengineering.com/quantum-effects-at-7-5nm/. (publicat: 23.05.2018).

[16] Linus Torvalds. *Hybrid kernel, not NT*. URL: https://www.realworldtech.com/forum/?threadid=65915&curpostid=65936. (Mai, 2006).

[17] ANTHONY WILLIAMS. *C++ Concurrency in Action. PRACTICAL MULTITHREADING*. 2012. Chap. 9-Advanced thread management, p. 274. ISBN: 9781933988771.

[18] www.amd.com. *AMD Ryzen^{TM} Threadripper^{TM} 2990WX Processor*. URL: https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-2990wx.

[19] www.ibm.com. *The First Multi-Core, 1GHz Processor*. URL: https://www.ibm.com/ibm/history/ibm100/us/en/icons/power4/.

[20] Giancarlo Zaccone. *Python Parallel Programming Cookbook*. 2015. Chap. Chapter 3: Process-based Parallelism, pp. 95–97. ISBN: 9781785289583.

Glosar

AMD Advanced Micro Devices. 1

nm nanometru. 1

UCP Unitatea Centrala de Procesare. 1, 11, 12