Quicksort Implementation – Proiect APD

Student: Ristea Andreea-Elena
Grupa: CR3.3A
Specilializarea: Calculatoare Romana

In cadrul acestui proiect am ales sa implementez algoritmul de sortare Quicksort sub urmatoarele forme:

- Implementarea secventiala standard, in C++
- Implementarea secventiala utilizand functia ***qsort*** din libraria ***"cstdlib"***
- Implementarea paralela folosind MPI, in C++
- Implementarea paralela folosind CUDA

I. Ideea algoritmului quicksort (principiul de functionare)

Ideea principala a acestui algoritm este de a impartii un vector mare in doi sub-vectori mai mici: elementele mai mici decat un element ales numit pivot, si elementele mai mari decat pivotul ales.

Procesul de sortare se realizeaza recursive pe sub-vectorii rezultati astfel incat toti sub-vectorii la final sa aiba dimensiunea 1 sau 0, moment in care algoritmul se opreste.

Pivotul se poate alege ca fiind orice element din vector.

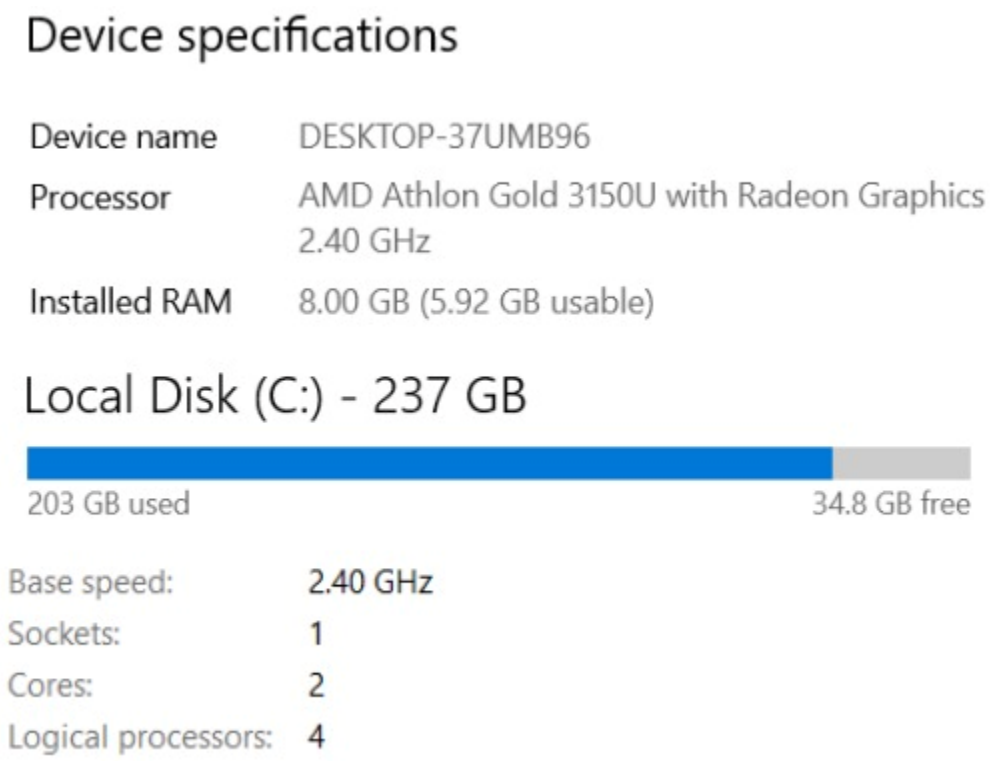
In procesul de sortare, se foloseste o tehnica de partitionare pentru a separa elementele mai mici decat pivotul, respectiv elementele mai mari.

Partitionarea incepe cu extremitatile sub-vectorilor: i pentru la stanga, j pentru la dreapta. Indicatorii se deplaseaza catre centru pentru a identifica elementele care nu sunt in ordine. In cazul in care se gaseste un element care trebuie mutat, el se interschimba cu alt element indetificat ca fiind pe pozitia gresita.

In cele din urma, pivotul ajunge pe pozitia corecta, in mijlocul sub-vectorilor. Astfel se obtine un sub-vector stang care contine toate elementele mai mici decat pivotul, iar cel drept contine toate elementele mai mari ca pivotul.

In continuare se va aplica acelasi proces si pe cei doi subvectori, recursiv, pana cand tot vectorul va fi sortat.

II. Specificatiile masinii pe care au fost rulate implementarile



III. Datele care se vor testa si pentru care se vor obtine rezultatele comparative pentru cele 4 implementari

Cele 4 implementari ale algoritmului quicksort vor fi rulate pe 9 teste de dimensiuni diferite, dupa cum urmeaza:

1. Fisierul “test1.txt” va contine 10 elemente, numere intregi, cuprinse intre 1 si 100
2. Fisierul “test2.txt” va contine 50 de elemete, numere intregi, cuprinse intre 1 si 1000
3. Fisierul “test3.txt” va contine 100 elemete, numere intregi, cuprinse intre 1 si 10000
4. Fisierul “test4.txt” va contine 150 elemete, numere intregi, cuprinse intre 1 si 100000
5. Fisierul “test5.txt” va contine 200 elemete, numere intregi, cuprinse intre 1 si 1000000
6. Fisierul “test6.txt” va contine 1000 elemete, numere intregi, cuprinse intre 1 si 1000000

7. Fisierul “test7.txt” va contine 5000 elemete, numere intregi, cuprinse intre 1 si 1000000
8. Fisierul “test8.txt” va contine 10000 elemete, numere intregi, cuprinse intre 1 si 1000000
9. Fisierul “test9.txt” va contine 50000 elemete, numere intregi, cuprinse intre 1 si 1000000

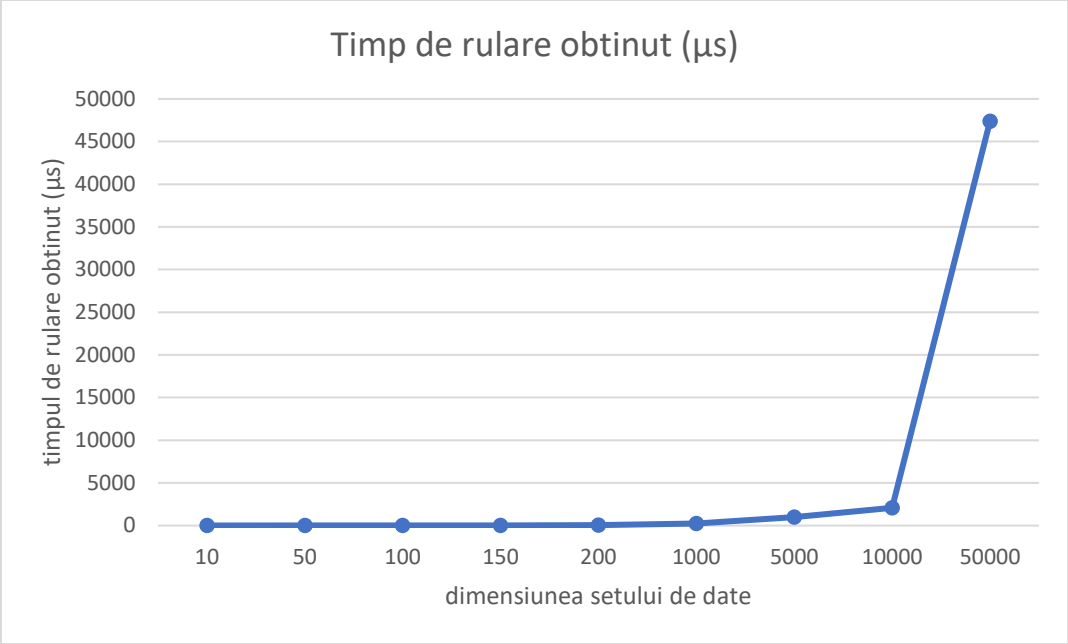
   Aceste teste au fost obtinute in urma rularii functiei test_generator().

IV. Implementarea secventiala standard, in C++

A. Pentru aceasta varianta de implementare am folosit functiile:
   - Partition (vector, left, right) => pentru realizarea partitionarii vectorului in sub-vectori in jurul unui element ales ca pivot.
   - Interchange_values (address1, address2) => pentru interschimbarea a doua valori care nu sunt asezate correct in sub-vector.
   - Quick_sort(vector, left, right) => pentru realizarea sortarii propriu-zise, in mod recursive, cu ajutorul pivotului ales si al functiei de partitionare.

B. In urma rularii acestei implementari pe testele descrise anterior au fost obtine urmatoarele rezultate

Dimensiune test	Timp de rulare obtinut (µs)
10	9
50	13
100	15
150	29
200	39
1000	228
5000	983
10000	2062
50000	47362



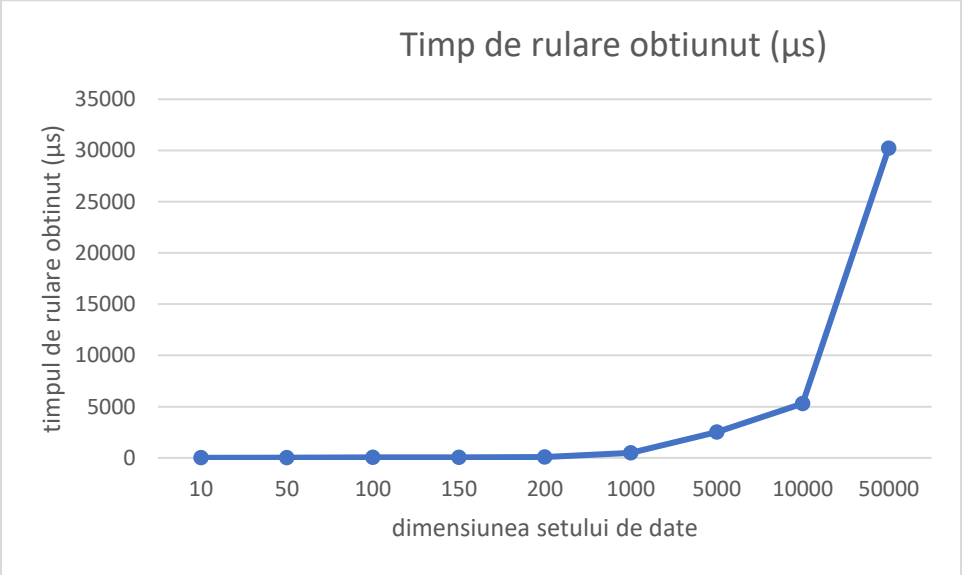
V. Implementarea secventiala utilizand functia ***"qsort"*** din libraria ***"cstdlib"***

A. Pentru aceasta implementare am folosit functia de sortare qsort (vector, dim, sizeof(int), compare).

Compare(const void* a, const void* b) => functie care compara doua elemente. Intoarce -1 daca primul argument este mai mic decat al doilea, 1 daca primul argument este mai mare decat al doilea, respective 0, daca ambele argumente sunt egale.

B. In urma rularii acestei implementari pe testele descrise anterior au fost obtine urmatoarele rezultate

Dimensiune test	Timp de rulare obtinut (µs)
10	10
50	21
100	42
150	62
200	80
1000	480
5000	2507
10000	5286
50000	30227

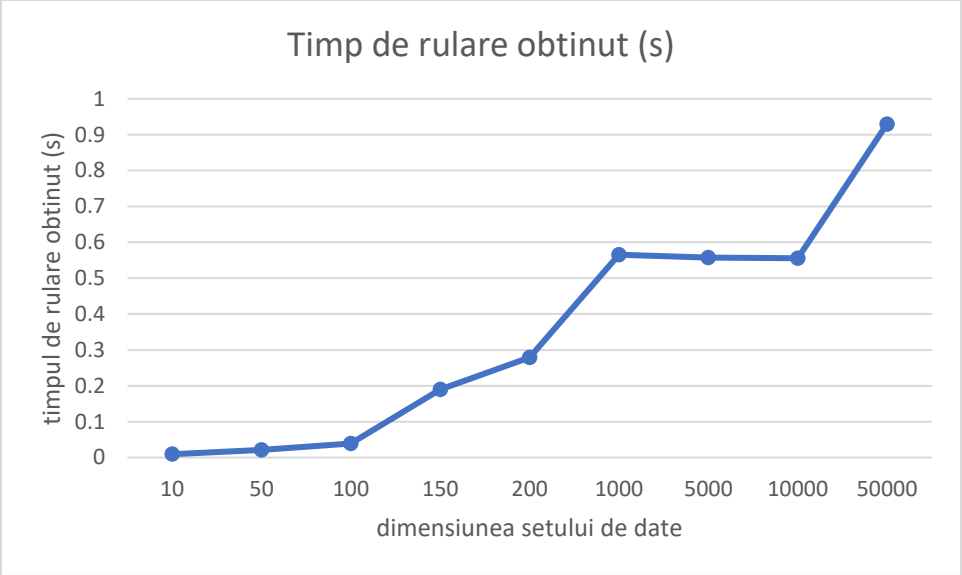


VI. Implementarea paralela folosind MPI, in C++

A. Si in acest caz sortarea propriu-zisa se face cu ajutorul functiei quick_sort(vector, left, right), doar ca acum va fi apelata pentru diferite bucati din vector, corespunzatoare anumitor procese. Procesul radacina va trimite portiuni egale din vector catre celelalte procese. Fiecare proces va avea cate o portiune diferita de date. Apoi pentru fiecare portine corespunzatoare unui process se apeleaza functia “quick_sort”, iar prin intermediul functiei MPI_Gather procesul radacina aduna datele de la celealte procese intr-un buffer.
Asupra datelor ajunse la procesul radacina se va mai apela functia pentru sortare, astfel incat tot vectorul sa fie sortat.

B. In urma rularii acestei implementari pe testele descrise anterior au fost obtine urmatoarele rezultate

Dimensiune test	Timp de rulare obtinut (s)
10	0.0092314
50	0.0215655
100	0.0387179
150	0.189447
200	0.278640
1000	0.565778
5000	0.557515
10000	0.555386
50000	0.929792

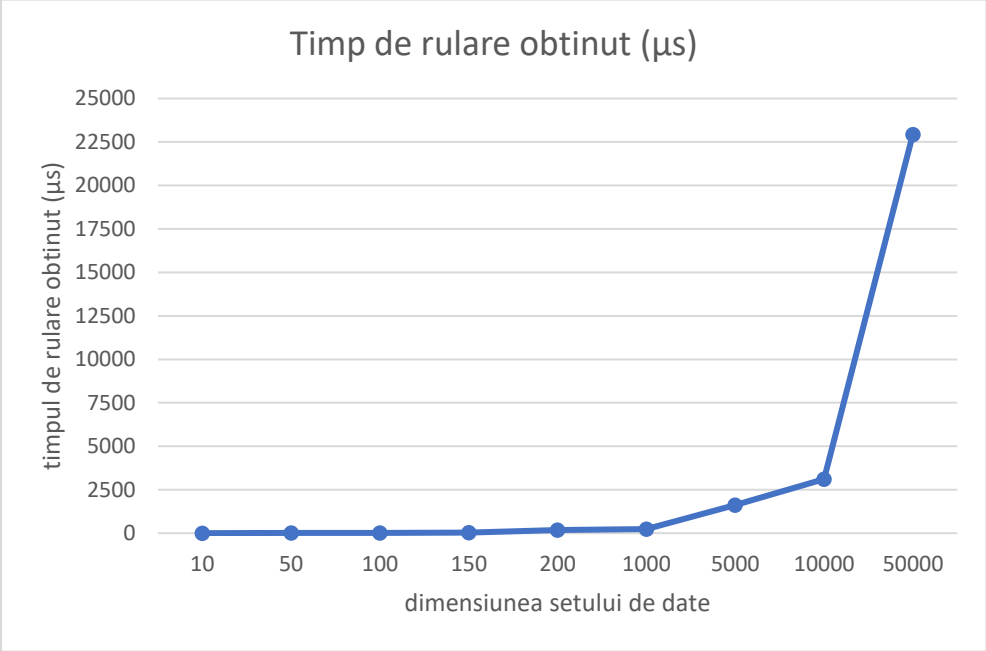


VII. Implementarea paralela folosind OpenMP, in C++

A. Pentru realizarea algoritmului de sortare folosind OpenMP s-a folosit biblioteca “opm.h”. Secventa “#pragma omp parallel sections” indica faptul ca urmatorul cod ce va urma se va executa in mod paralel utilizand sectiuni paralele. Secventa “#pragma omp section” indica faptul ca, codul cuprins intre prima si urmatoarea secventa de tipul respectiv, va fi executat in paralel de mai multe fire de executie.

B. In urma rularii acestei implementari pe testele descrise anterior au fost obtine urmatoarele rezultate

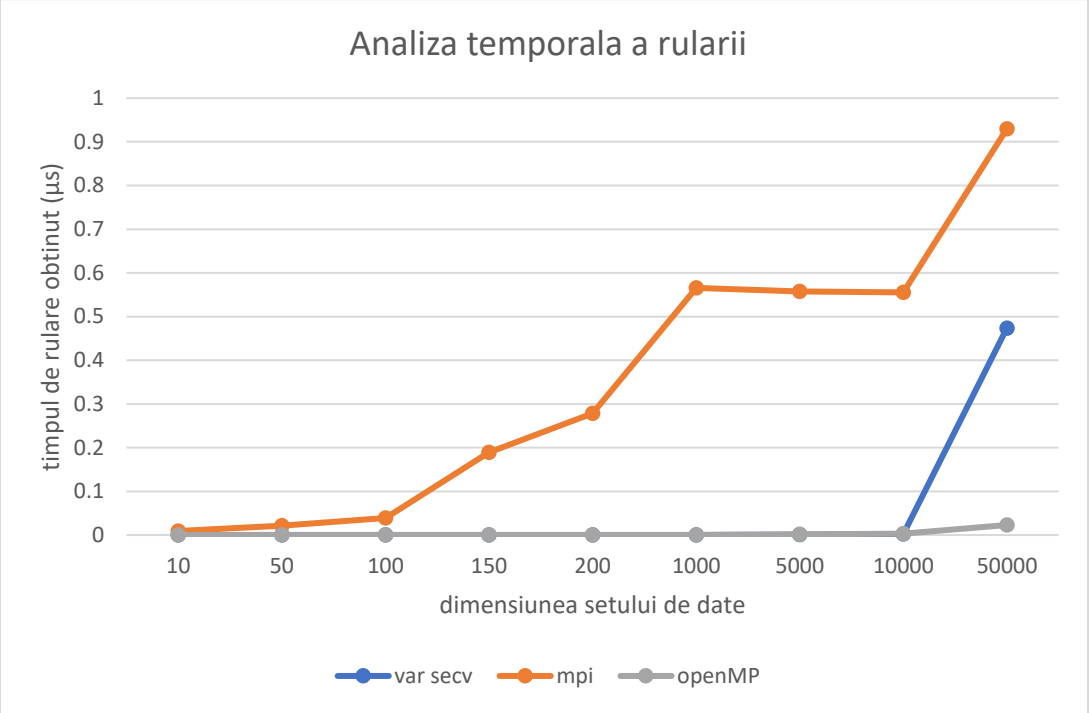
Dimensiune test	Timp de rulare obtinut (µs)
10	2.2
50	11.5
100	23.7
150	33.7
200	175.8
1000	238.6
5000	1626.2
10000	3114.7
50000	22937.7



VIII. Analiza comparativa a rezultatelor obtinute

Dimensiunea setului de date	Timp de rulare varianta secventiala (s)	Timp de rulare varianta paralela MPI (s)	Timp de rulare varianta OpenMP (s)
10	0.000009	0.009231	0.0000022
50	0.000013	0.021565	0.0000115
100	0.000015	0.038717	0.0000237

150	0.000029	0.189447	0.0000337
200	0.000039	0.27864	0.0001758
1000	0.000228	0.565778	0.0002386
5000	0.000983	0.557515	0.0016262
10000	0.002062	0.555386	0.0031147
50000	0.47362	0.929792	0.0229377



IX. Concluzii:

- Eficienta algoritmului de sortare Quicksort intr-o implementare secventiala sau paralela poate varia in functie de dimensiunea setului de date asupra caruia este apelat, cat si in functie de varianta de implementare abordata (varianta paralela MPI, varianta paralela OpenMP).
- Varianta secventiala si varianta paralela OpenMP au o eficienta aproximativ echivalenta in rularea pe seturile de date testate.
- Se observa ca varianta paralela cu MPI obtine valori mai mari pentru timpul de rulare comparativ cu celealate doua variante de implementari

Referinte:
- https://www.programiz.com/cpp-programming/library-function/cstdlib/qsort
-https://www.geeksforgeeks.org/implementation-of-quick-sort-using-mpi-omp-and-posix-thread/
-https://www.geeksforgeeks.org/quick-sort/