README

-Serban Alexandru Dumitru 332CD-

1. Feedback

Tema a fost in mare parte buna, utila pentru a exersa progamarea paralela si nici foarte grea. As avea de mentionat doar ca poate masurarea acceleratiei nu a fost facuta intocmai bine, testul 1 este destul de mic, iar pentru o solutie eficienta, care sa ruleze undeva de la 2.8-3 secunde, speedup de 1.75x este destul de mare calculat per program, partea paralela putand sa devina mai putin semnificativa ca timp de rulare fara de cea secventiala cand setul de date este redus. Cred ca am rezolvat cerinta destul de eficient. Implementarea a durat undeva la 20 de ore de lucru.

2. Strategia de paralelizare – Descrierea implementarii

Programul functioneaza astfel:

Citesc prima oara fisierul cu fisierele cu articole si le pun intr-o coada blocanta pentru a putea sa extrag cate unul cu threadurile. Citesc apoi cele 3 fisiere de input si pun datele in set-uri (au acelasi format am creat o functie pentru a le citi). Folosesc acelasi tip de cozi si pentru categorii si pentru limbi. Le initializez, initializez si o bariera si pornesc executia threadurilor.

Am impartit munca intre thread-uri pe etape: Citirea fisierelor JSON si procesarea datelor (unde si scriem fisierele de limbi si categorii).

In prima etapa threadurile extrag cate un fisier cu articole din coada si il citesc/proceseaza: citeste fiecare articol, creeaza obiectul articol, il introduce in alta coada blocanta cu articole prin care voi trece in a 2-a faza pentru procesarea articolelor, numara articolul citit, adauga articolele in HashMap-urile concurente pentru limba si categorie pe care le voi folosi la scrierea

fisierelor si stabileste duplicatele folosind alte 2 HashMap-uri dupa titlu si uuid (criteriile de duplicate). Cand nu mai gaseste fisiere de citit in coada asteapta la bariera pana vin toate threadurile ca sa trecem la urmatoarea faza de procesare (trebuie sa fi trecut prin toate articolele pentru a cunoaste toate duplicatele).

A doua faza de procesare consta in: extragerea a cate unui articol din coada numararea articolelor per autor intr-un HashMap si numararea keywords – luam text-ul de la articol, aplicam normalizarea si impartirea si verificam ce cuvinte nu sunt linking_words si le numaram tot intr-un HashMap (evident cele 2 Map-uri sunt Concurrent); apoi extragerea a cate unei limbi/categorii din cozile create la inceput, numararea acesteia, dar si scrierea fisierului corespunzator ei, folosind Map-urile construite in citire, verificand care articole nu sunt duplicate.

Dupa finalizarea executiei threadurilor si revenirea pe main mai execut urmatoarele secvential: Creez array-ul de articole unice, dintr-un HashMap create de threaduri (cel dupa titluri), iar cu acesta scriu fisierul all_articles.txt; Sortez apoi keywords si creez fisierul keywords_count.txt; Apoi fac ultimele calculi pentru rapoartele statistice: nr de duplicate, nr de unice, iau autorul cu cele mai multe articole, caut categoria si limba cea mai intalnita, cel mai recent articol si cel mai aparut keyword si le scriu pe toate in fisierul reports.txt.

Descrierea mecanismelor de sincronizare:

- in primul rand, threadurile isi impart munca luand cate un fisier cu articole/articol/limba/categorie dintr-un LinkedBlockingQueue, care este o coada blocanta ce permite thread-urile sa extraga sigur cate un item; folosesc acest tip de sincronizare (stil replicated workers) deoarece daca foloseam impartirea clasica (pe chunck-uri) puteam avea munca impartita inegal (un thread primeste fisiere mici si unul mari), asa toate threadurile isi iau de munca imediat cum termina cu thread-ul curent.

- pentru numararea articolelor am folosit o metoda folosita la laborator si la testul de laborator, si anume fiecare thread isi aduna intr-un element dintr-un vector, iar apoi main va aduna aceste valori; daca punteam toate threadurile sa adune in acelasi contor aveam foarte mult timp pierdut in care threadurile asteptau sa scrie in acel contor.
- am folosit destul de multe ConcurrentHashMap pentru a putea retine informatii sub forma de cheie, valoare dar care sa fie blocante, pentru a putea scrie/citi in/din ele pe threaduri in timp ce facem citirea, procesarea; am 2 pentru duplicate (dupa titlu si uuid), 2 pentru fisierele ce trebuie sa le generez din cerinta (dupa limba si categorie), si inca 2 pentru a numara aparitiile de keywords si autori.
- sincronizarea dintre cele 2 faze ale thread-urilor se face cu un CyclicBarrier, initializat cu numarul de threaduri primit ca argument in linia de comanda; aceasta bariera asigura ca pana a incepe procesarea/scrierea in fisiere vor fi fost citite toate articolele si identificate toate duplicatele (asta asigura corectitudinea rezolvarii).

Design-ul este corect deoarece thread-urile scriu doar in zone blocante de memorie (este asigurata sincronizarea prin structurile prezentate mai sus), dar logica este asigura si de bariera dintre etape. Din aceste motive nu avem undefined behaviour, avem acelasi rezultat, cu oricate threaduri la oricate rulari. Algoritmul este eficient pentru procesarea paralela deoarece, am spus deja, munca se imparte cat de cat egal, decat sa riscam sa alocam mai multe fisiere mari unui thread si altele mai mici altui thread, rezultand in cel cu munca mai usoara sa stea apoi sa-l astepte pe cel ce a primit fisiere mari. De asemenea, parcurg toate articolele doar de 2 ori, incluzand citirea – este minimul deoarece este imposibil sa se resolve cerinta direct din citire avand nevoie sa citim toate articolele mai intai pentru a elimina duplicatele.

3. Analiza de performanta si scalabilitate

Setup de testare:

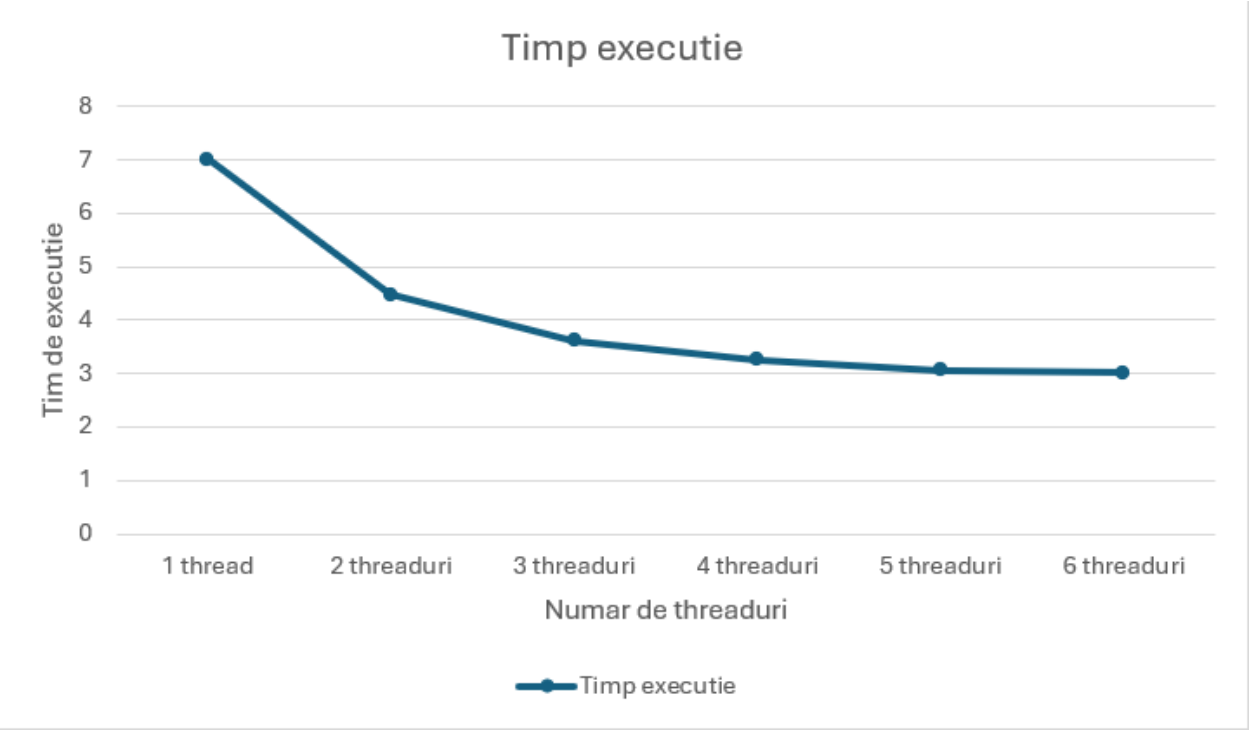
- CPU: AMD Ryzen 7 7840HS with Radeon 780M Graphics

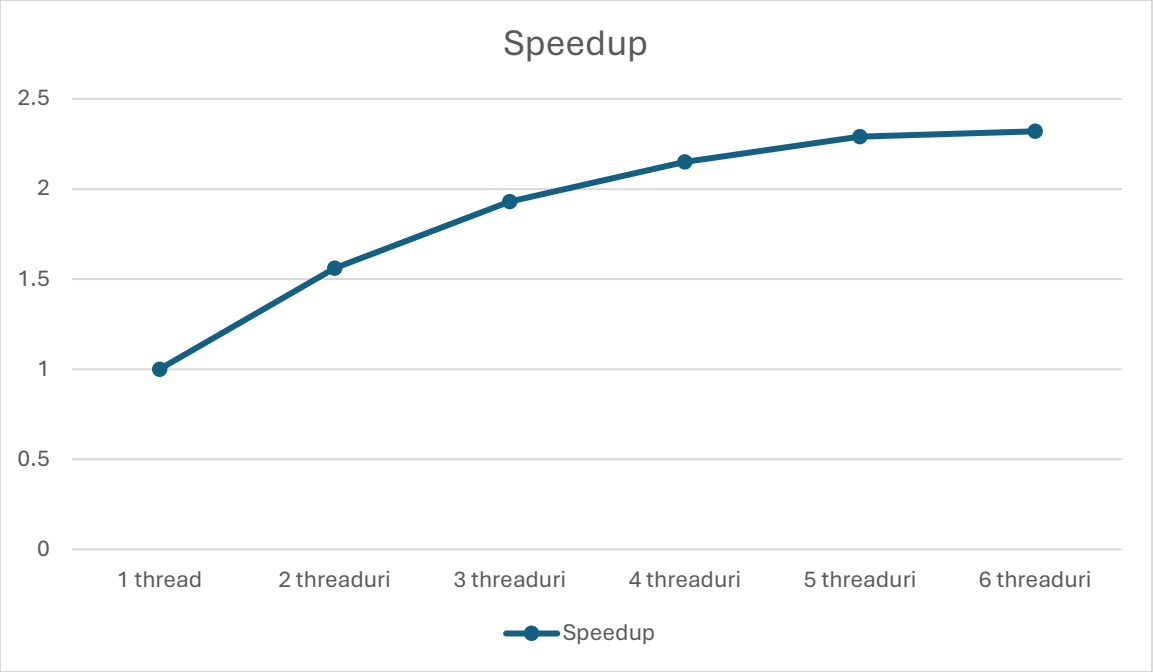
- Cores: 16
- Ram: 12 Gi
- OS: Ubuntu 24.04.3 LTS
- Versiune Java: openjdk 25.0.1 2025-10-21
- Voi folosi testul 3 din checker ca dataset – dimensiune: 8273 de fisiere JSON cu articole

Rezultate:

Nr. Threads	Run 1 (s)	Run 2 (s)	Run 3 (s)	Average (s)	Speedup	Efficiency
1	6.799	7.563	6.687	7.016	-	-
2	4.272	4.625	4.513	4.470	1.56	0.78
3	3.523	3.806	3.539	3.622	1.93	0.64
4	3.305	3.258	3.201	3.254	2.15	0.53
5	2.935	3.013	3.219	3.055	2.29	0.45
6	2.941	3.118	2.990	3.016	2.32	0.38

Grafice:





Analiza si concluzii:

Se observa ca programul scalea odata cu cresterea numarului de thread-uri. Cu cat avem mai multe threaduri cu atat scade timpul de executie/creste speedup-ul (Se observa in cele 2 grafice). Din aceasta analiza si set de date se observa ca performanta creste de la 3 thread-uri si se stabilizeaza pe la 5,6 thread-uri. Aceasta limitare (plafonarea scalarii de la 6 thread-uri in colo) vine probabil din dimensiunea setului de date care nu este foarte mare. Cred ca numarul optim de thread-uri pentru acest program pe system-ul meu este undeva la 4-5, deoarece acolo exceleaza speedup-ul, si nu merita sa ocupam mai multe core-uri pentru o crestere nu foarte mare in performanta.