README TEMA 1 APD - AGREGATOR DE STIRI

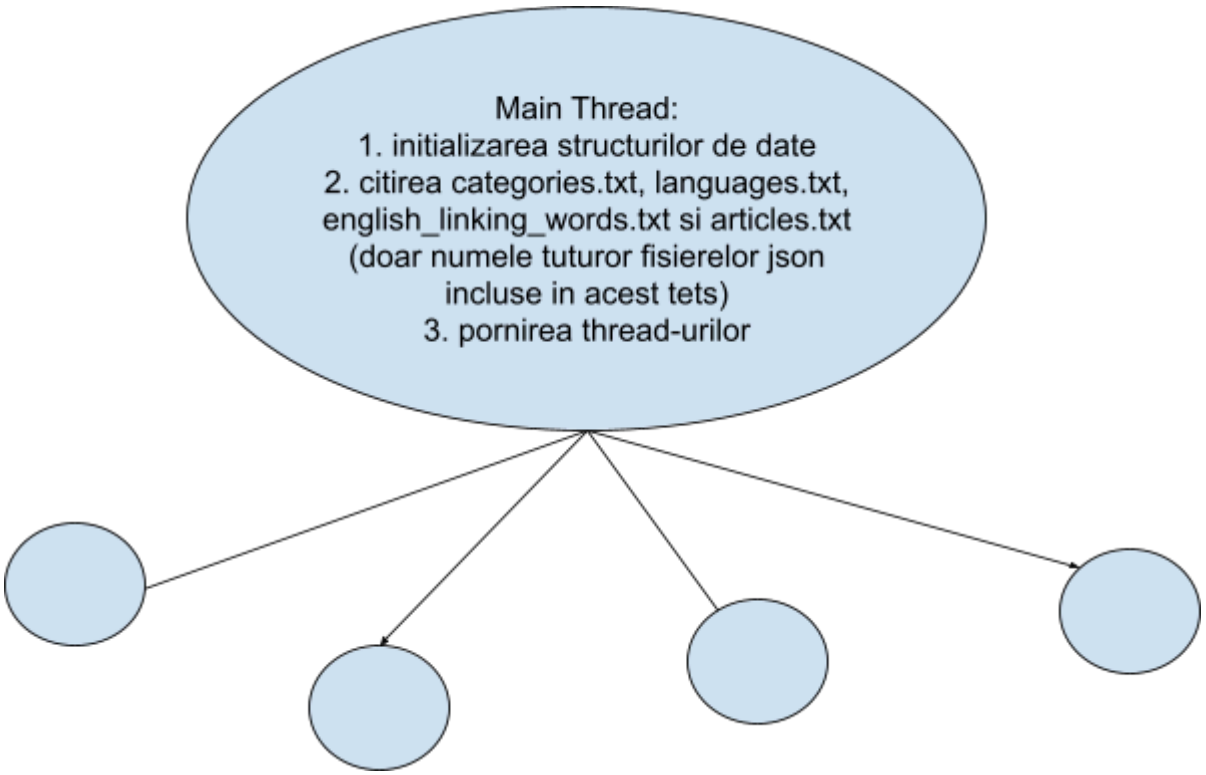
Stanciu Anastasia-Steliana 334CC

1. Feedback:

Durata implementare: 2 zile (scris cod) + 3 zile (imbunatatire performanta)
Puncte forte: Am putut vedea cum moduri DIFERITE de a paraleliza (structuri de date thread safe puse la dispozitie de Java /vs/ structuri de date cu rezulate partiale agregate intr-o structura de date cu rezultate globale) dau randament diferit in functie de context, problema, numarul de thread-uri sau dimensiunea datelor de intrare.

2. Strategia de paralelizare:

Flow-ul programului are loc dupa cum urmeaza:



Orice alta funcionalitate a codului are loc **paralel** in proportiile si modul descrise mai jos:

Citirea articolelor - readArticles()

In main s-a citit articles.txt si s-a creat un vector cu toate fisierele .json care fac parte din acest test. Fiecare thread va citi o parte din aceste .json, populand o structura de date locala cu rezultatele sale partiale. Doar thread0 insumeaza rezultatele locale in rezultate globale.

Eliminarea duplicatelor - duplicatesRemoval()

Vectorul **articles** este impartit intre thread-uri. Fiecare thread are 2 hashmap-uri de frecventa locale: partialArticlesByTitle si partialArticlesByUuid. La final, doar thread0

insumeaza vectorii de frecventa locali in doi vectori de frecventa globali (unul in functie de titlu si unul in functie de uuid). Se elimina duplicatele din ambele categorii de catre thread0.

Clasificarea dupa limba si categorie - categoryClassification() & languageClassification()

Fiecare thead creeaza un hashmap de frecventa local pentru sectiunea de articole asignata. Fiecare thread isi adauga rezultatele partiale in hashmap-ul de frecventa global care este un ConcurrentHashmap.

Cuvinte de interes - interestWords()

Analog cu sectiunea de mai sus.

Scrierea

Am ales sa fac scrierea separat pentru a nu ingreuna cursul programului cu multe bariere. Astfel, cele 4 executii (eliminarea duplicatelor, impartirea pe categorii, pe limbi si numararea cuvintelor de interes) au loc fara piedici, in paralel, iar scrierea se face tot in paralel la finalul executiei tuturor celor 4 task-uri. Astfel am facilitat scalabilitatea codului, evitand utilizarea excesiva a barierelor de sincronizare.

Scrierile sunt distribuite thread-urilor in functie de numarul lor.

Statistici

bestAuthor(), mostRecentArticle() - thread-urile cauta in segmentul asociat din vectorul **articles** un “maxim” local (autorul cel mai popular local sau articolul cel mai recent local). Rezultatele globale sunt calculate la scriere doar de catre thread0 pentru a evita din nou utilizarea excesiva a barierelor. Vom vedea in sectiunile urmatoare.

topLanguage(), topCategories(), topEnglishWords() - Acestea sunt toate operatii efectuate in O(1):

- primele 2 presupun iterarea prin nu mai mult de 50 de categorii/limbi (dimensiunea datelor de intrare - adica nr de articole - nu afecteaza aceste operatii)
- topEnglishWord() este O(1) pentru ca presupune doar returnarea valorii aflate pe prima pozitie din vectorul de frecventa global al cuvintelor, calculat de functia interestWords() care a lasat acest vector **ordonat descrescator dupa nr de aparitii**

Deci aceste operatii nu prezinta interes in potentarea scalabilitatii.

Scrierea

Inainte de scriere se vor calcula rezultatele globale restante (bestAuthor si mostRecentArticle). Apoi scrierea (care este O(1) - mereu scriem 6 rezultate) este efectuata doar de thread0.

3. Analiza de performanta si scalabilitate

Setup de testare:
Model name: 13th Gen Intel(R) Core(TM) i7-13620H
CPU(s): 16
Architecture: x86_64

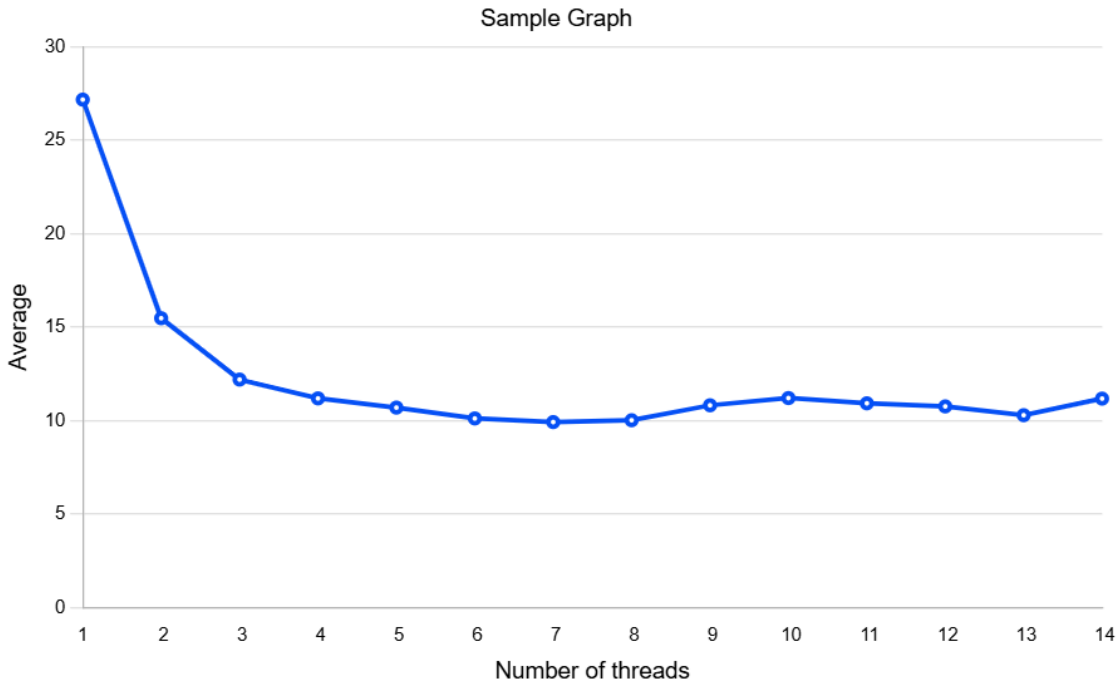
Java Version: OpenJDK 15

Rezultate:

Numar Thread-uri	**Timpi executie (3 rulari)**	**Media**
1	29.8422 25.6687 26.0079	**27.1730**
2	**15.7793** **15.1093** **15.5568**	**15.4818**
3	**13.0401** **12.1688** **11.3891**	**12.1993**
4	**11.5995** **10.6592** **11.3184**	**11.1924**
5	**10.2672** **11.3255** **10.5068**	**10.6998**
6	**10.3364** **10.1754** **9.8458**	**10.1192**
7	**9.6819** **9.8744** **10.2212**	**9.9258**
8	**10.2053** **10.1910** **9.6722**	**10.0228**
9	**9.9686** **10.5256** **11.9739**	**10.8227**
10	**11.8743** **11.2302** **10.5252**	**11.2099**

11	**11.4943 10.4153 10.8766**	**10.9287**
12	**10.7969 10.6448 10.8490**	**10.7635**
13	**10.3940 9.9508 10.5348**	**10.2932**
14	**11.2002 10.7790 11.5617**	**11.1803**

REPREZENATREA GRAFICA A DATELOR



Analiza si concluzii:

Comportament observat: La inceput, cresterea numarului de threaduri imbunatateste semnificativ performanta codului (2, 3, 4, 5 thread-uri). Insa dupa 6 thread-uri, performanta stagneaza (6, 7, 8 thread-uri). Chiar mai mult, remarcam cum pentru un numar mare de thread-uri (9, 10, 11, 14), performanta incepe sa scada! O motivatie solida o consituie overhead-ul adaugat de sistemele de sincronizare care, de la un punct incolo (vizibil pe grafic), contrabalanseaza beneficiile aduse de paralelism, avand un efect negativ asupra

performantei programului. Un alt motiv bun consta in faptul ca data set-ul fiind limitat, un numar foarte mare de thread-uri poate duce la crearea unui overhead mare pentru a prelucra putine date in paralel. Pentru testare s-a ales testul 3 din setul propus de tema.

Numarul optim de thread-uri: 7

O concluzie interesanta este ca numarul optim de thread-uri (pentru care s-a obtinut cel mai mic timp mediu de rulare) se afla FIX la jumatatea intervalului de numar de thread-uri testat! Astfel legea lui Amdhal se sustine.