APD – Tema 1

Bîrsan Alexandru-Vlad
Grupa: 333CC

Sectiunea 1 – Feedback

Timp de implementare : 20 de ore (Citire cerinta, documentare, implementare si scriere README)

Tema a avut un timp de rezolvare bun intru totul, am simtit ca am reusit sa invat si sa sedimentez conceptele pe care nu le am putut intelege la laboratoare si sa ajung sa pot sa pornesc de la 0 cu un proiect de genul, spre deosebire de laborator unde avem de implementat doar mici portiuni de cod. As fi ajustat putin cerinta, este foarte greu de digerat la prima vedere. Am avut probleme cu inceperea MakeFile-ului si cu rezolvarea pasilor necesari pentru a face checker-ul sa mearga. Mi-ar fi placut de asemenea sa fii vazut punctaje partiale la teste in checker sau sa vad problemele pe care le are codul meu mai usor.

Sectiunea 2 – Strategia de paralelizare

Am inceput implementarea temei dupa ce am facut laboratorul care ne-a introdus ExecutorService impreuna cu ThreadPool-uri si atunci prima incercare a fost sa vad daca ar fi o varianta mai buna sa incep cu aceasta strategie. Am incercat sa gasesc solutii dar in final m-am intors deoarece nu stiam cum as putea face diferite etape ale procesarii folosind acea solutie deoarece nu o stapaneam.

Am ales pana la urma sa folosesc varianta cu implementarea interefetei Runnable in clasa Worker pe care am facut-o pentru thread-uri (pe care le-am pornit cu .start()) din main. Folosesc bariere pentru a delimita etapele functiei run. Am impartit munca in felul urmator:

1. Citirea argumentelor din linia de comanda se intampla in main care creeaza Lista de fisiere de Json-uri si un obiect de tip InputsData (clasa creata de mine pentru a tine minte datele din inputs.txt)
2. Crearea thread-urilor si pasarea argumentelor necesare pentru procesarea datelor adica ex: lista de fisiere, lista goala de articole, diferite HashMap-uri pentru a tine contor pentru cerintele temei.
3. Primul lucru pe care il fac in run() este sa construiesc lista care contine toate articolele de la fisierul de file si in acelasi timp incrementez cele 2 contoare care tin uuidCounts-urile si titleCounts-urile pentru a ma ajuta la urmatoarea etapa.
4. La aceasta etapa, dupa ce thread-urile asteapta la o bariera deoarece aveam nevoie de lista completa de articole. Impart munca thread-urilor folosind balansare statica de genul:

```
int size = sharedList.size();
int sizeForThread = size / totalThreads;
int startIndex = sizeForThread * id;
int endIndex = (id == totalThreads - 1) ? size : sizeForThread * (id + 1);
```

5. Dupa o alta bariera (deoarece aveam nevoie de lista fara duplicate - finalCleanList) am ajustat index-ul pentru start si pentru end la noua dimensiune a listei si am inceput prelucrarea datelor cerute (AuthorCounts, completarea map-urilor pentru language si categorii, wordCounts). Aici se termina partea cerintei paralelizata. De acum incolo totul se intampla in main.

6. Dupa ce am dat join la threads in main() am inceput sa calculez diferitele metrici de care aveam nevoie pentru scrierea fisierelor.

7. Apelez metode de printare si creare a fisierelor cerute (tot secvential)

Mecanisme de sincronizare folosite:

1. Concurrent HashMap – de departe cel mai folosit in implementarea acestei teme l-am folosit pentru:

```
private ConcurrentHashMap<String, Integer> titleCounts;
private ConcurrentHashMap<String, Integer> uuidCounts;

private ConcurrentHashMap<String, List<String>> catMap;
private ConcurrentHashMap<String, List<String>> langMap;

private ConcurrentHashMap<String, Integer> keywordCounts;
private ConcurrentHashMap<String, Integer> authorCounts;
```

Deoarece imi permitea sa tin minte, sa scriu si sa citesc diferite keys si valori in paralel fara sa am probleme ca mai multe thread-uri incearca sa acceseze acelasi lucruri. Cu ele am putut prelucra majoritatea lucrurilor de care am avut nevoie in paralel ca dupa sa le pot avea stocate pentru printare. Am ales aceasta varianta in locul celei cu Collections.synchronisedMap deorece permite access concurent ajungand la performante relativ mai bune.

2. Atomic Integer – Am folosit acest mecanism pentru a crea un counter care imi foloseste la citirea path-urilor care fisierele Json la inceputul metodei run() . El functioneaza astfel: Este declarat in main si ii atribui valoarea 0, dupa care este pasat lui Worker , iar cand un thread are nevoie sa ia cate un fisier din lista de fisiere foloseste metoda .getAndIncrement() care ii da indexul curent si incrementeaza dupa get ca urmatorul thread sa nu ia acelasi numar. Fiecare thread poate astfel sa faca in siguranta files.get(i).

Argumentare:

Am folosit un design simplu fara foarte multe mecanisme complicate de sincronizare sau metode de rezolvare alternative cum ar fii ExecutorService cu threadPool fix. Acesta este corect pentru nevoile temei si suficient de performant cat sa treaca testele necesare, in timp ce ramane usor de inteles folosind doar 2 bariere in partea paralela a codului. In rest am optat pentru varianta secventiala intrucat nu am crezut ca exista vreun gain mai mare daca as fi paralelizat mai mult de atat, ori parea foarte complicat de facut.

Sectiunea 3 – Analiza de performanta si scalabilitate

Setup de testare:

```
CPU:
  Info: 6-core model: AMD Ryzen 5 5500U with Radeon Graphics bits: 64
    type: MT MCP arch: Zen 2 rev: 1 cache: L1: 384 KiB L2: 3 MiB L3: 8 MiB
  Speed (MHz): avg: 1580 high: 2922 min/max: 1400/2100 boost: enabled
cores:
    1: 2922 2: 1397 3: 1527 4: 1434 5: 1400 6: 1400 7: 1397 8: 1397 9: 1397
    10: 1751 11: 1476 12: 1469

Memory:
   total: 16 GiB note: est. available: 14.95 GiB

System:
  Kernel: 6.8.0-88-generic arch: x86_64 bits: 64 compiler: gcc v: 13.3.0
  Desktop: GNOME v: 46.0 Distro: Ubuntu 24.04.1 LTS (Noble Numbat)

Versiune Java:
  javac 21.0.9
  java
  openjdk version "21.0.9" 2025-10-21
  OpenJDK Runtime Environment (build 21.0.9+10-Ubuntu-124.04)
  OpenJDK 64-Bit Server VM (build 21.0.9+10-Ubuntu-124.04, mixed mode,
sharing)
```

```
Dimensiune dataSet de test:
     Am folosit testele date de checker test_2 si test_4 care au dimensiuni
de:
     Test_2:                                  Test_4:
          5516 fisiere JSON                        11031 fisiere JSON

Daca iau un average de 5 articole per fisier de JSON ajungem la concluzia
ca primul test are in jur de 27580 articole, iar cel de al doilea test are
in jur de 55155 articole care trebuie procesate de program.
```

Tabel cu timpii de executie (s)

Am ales sa rulez pentru 1, 2, 3, 4, 6 si 8 thread-uri (Am ales 6 si 8 pentru a vedea mai clar plafonarea speedup-ului si a arata numarul optim de thread-uri necesar pentru rularea programului)

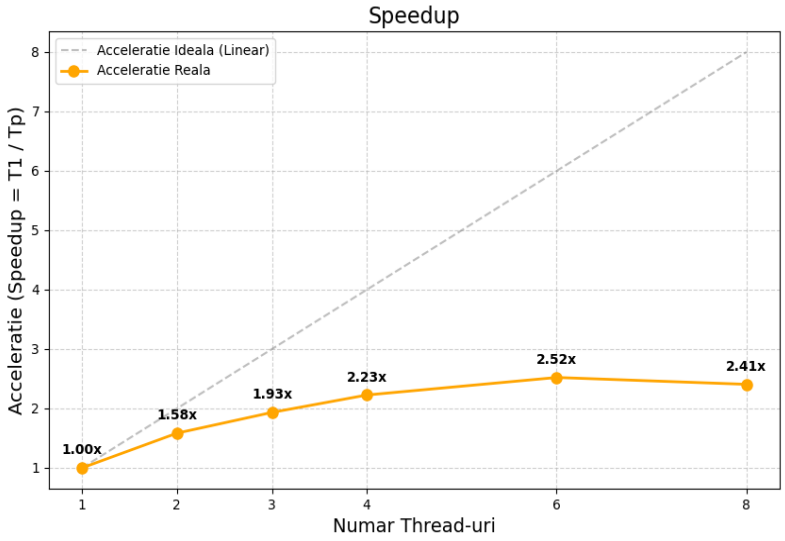
TEST_1

Nr Threads	1	2	3	4	6	8
Rulare 1	8.689	5.581	4.438	3.786	3.384	3.563
Rulare 2	8.538	5.454	4.524	3.917	3.562	3.848
Rulare 3	8.794	5.387	4.497	4.000	3.373	3.481
Medie	8.673	5.474	4.486	3.895	3.439	3.604

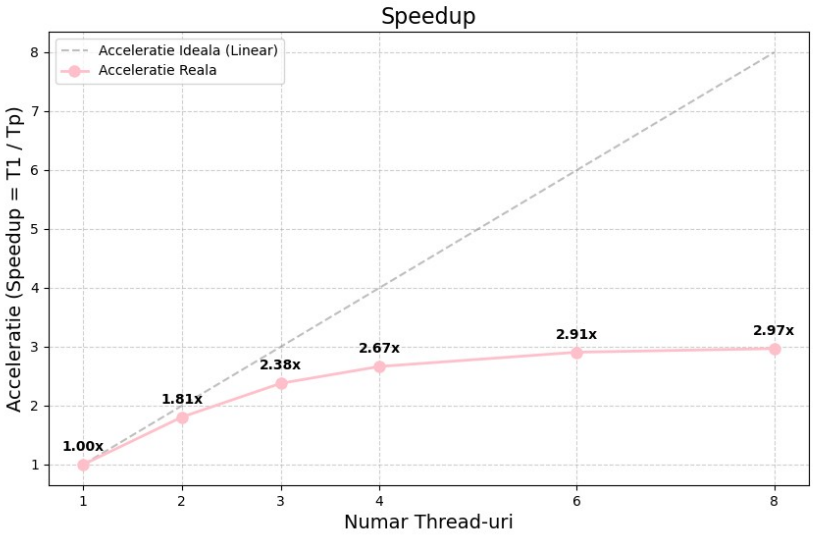
TEST_2

Nr Threads	1	2	3	4	6	8
Rulare 1	15.936	9.426	6.771	6.121	5.704	5.420
Rulare 2	16.023	8.863	7.021	6.031	5.856	5.497
Rulare 3	17.302	8.967	6.912	6.329	5.380	5.613
Medie	16.420	9.085	6.901	6.160	5.646	5.530

Grafic care reprezinta Speedup-ul pentru Test_1



Grafic care reprezinta Speedup-ul pentru Test_2



Timpul de rulare la fiecare test a fost foarte diferit am facut o medie pentru fiecare numar de thread-uri de minim 5 rulari si de aici am ajuns la aceste rezultate. De notat de asemenea ca am incercat sa fac testele cand laptop-ul era abia deschis fara nici un alt proces care ruleaza in afara de programul temei.

Tabele pentru vizualizare eficienta

Test_1

Nr Threads	1	2	3	4	6	8
Speedup	1	1.58	1.93	2.23	2.52	2.41
Eficienta	1	0.79	0.64	0.55	0.37	0.30

Test_2

Nr Threads	1	2	3	4	6	8
Speedup	1	1.81	2.38	2.67	2.91	2.97
Eficienta	1	0.90	0.79	0.66	0.48	0.37

Concluzii

Se poate observa, dupa testele facute, scalabilitatea sub-liniara a programului. Timpul de rulare scade cu cresterea numarului de thread-uri pana ajunge la o plafonare in jurul numarului de 6 thread-uri. Daca luam in considerare si eficienta ajungem la concluzia clara ca fiecare thread nou adaugat aduce gain-uri din ce in ce mai mici ex: 8 thread-uri la primul test E(p) = 0.30 – 30% si chiar scade speed-ul fata de 6 thread-uri aratand ca overhead-ul adus de crearea mai multor thread-uri intrece performantele aduse de acestea.

Probabil cea mai mare limitare a programului si de ce vedem eficienta destul de drastic scazuta este citirea si scrierea in fisiere care e in mare parte secventiala in cazul meu(desi citirea din fisierele JSON se face in paralel, restul adica citirea fisierelor de intrare si scrierea la final este secv. si nu este ajutata de adaugarea de thread-uri, Legea lui Amdahl). Overhead-ul de sincronizare adus de ConcurrentHashMap sau de bariere de asemenea introduce un cost suplimentar de coordonare ex: mai multe thread-uri incearca sa scrie in acelasi map , unul asteapta deci se reduce eficienta globala, la fel si pentru bariere, desi am impartit munca static si “egal” tot pot aparea momente de asteptare cand poate doar 1 thread nu a terminat munca deorece a primit un set de date mai mare.

Numarul optim pentru sistemul si programul meu poate fi:

-4 thread-uri deoarece acesta aduce un timp apropiat de cel mai bun dar eficienta mult mai buna ca alternativele (deci aduce un balans bun intre cele 2 metrici)

-6 thread-uri pentru performanta bruta , dar eficienta mai scazuta ca la 4 thread-uri (poate avea legatura cu faptul ca sistemul meu are 6 nuclee fizice si din acest motiv se vede aceasta plafonare dupa 6 thread-uri)