Documentatie Tema 1 – APD

1.Feedback

Din punctul meu de vedere a fost o tema foarte interesanta, cu o dificultate simpla spre medie. Ca si durata de timp as spune ca a durat cam 6 – 8 ore pentru realizarea acesteia, mai mult din cauza faptului ca checkerul de pe moodle dureaza foarte mult si pe langa asta da si foarte multe rezultate random, adica poti sa ai odata 75 odata 70.5 in functie de noroc. Mi-a placut foarte mult sa scriu multithreaded in java, as spune sincer ca este partea mea favorita dintre cele trei oferite de aceasta materie.

2. Strategia de paralelizare

Rezolvarea mea executa urmatorii pasi, citirea fisierelor de input in main (articles.txt, inputs.txt si fisierele mentionate in ele), dupa care articolele sunt salvate ca ArrayNode-uri intr-o lista care a poi este impartita in mode egal pe numarul de threaduri dupa formula clasica de start = thread_id * (n / n_of_threads) si end = (thread_id + 1) (n/ n_of_threads). In aceasta portiune folosim hashmapuri si seturi pentru a retine elementele duplicate si toate uuid-urile intalnite. Dupa ce gasim toate uuid-urile le parcurgem pe taote si verificam daca este un articol duplicat sau nu. In cazul in care este sarim peste el, iar daca nu este il procesam. In acest proces am folosti alte HashMap-uri in care retin autori, categorii, limbi si cuvinte, dar si seturi penru a le retine individual. Urmatoarea etapa din rezolvarea mea precalculeaza majoritatea rezultatelor din reports.txt, partea cu numerele de articole, autorii, categoria cea mai populara si cea mai populara limba. Se impart aceste 4 taskuri in mod egal la threaduri. Pentru etapa de afisare am folosit un queue ca sa impart munca cat mai egal pe threaduri pentru ca unele afisari dureaza mult mai mult decat celelalte. Incep cu keywords_count.txt pentru ca este mereu cea mai costisitoare si imediat dupa fac si reports pentru ca din sortare obtin si cel mi popular cuvant.

Am folosit mai multe metode si elemente pentru aceasta sincronizare. Am folosit formula clasica cu start si end, rezultate partiale pe threaduri(pentru cel mai recent articol si numarul total de articole). Am folosit si Clase specifice pentru lucrul multithreader precum ConcurrentHashMap(pentru Autori, limbi si categorii, cuvinte) , liste sincronizate(pentru articole), seturi concurente(limbi, categorii, cuvinte, autori), atomic integer pentru autori si cuvinte pentru a rétine de cate ori apar fiecare, dar si ArrayBlockingQueue pentru partea de afisare. Intre cele 4 etape prezentate mai sus am folosit cate o bariera pentru a ma asigura ca nu trec la urmatoarea etapa panca cand toate threadurile au terminat partea lor.

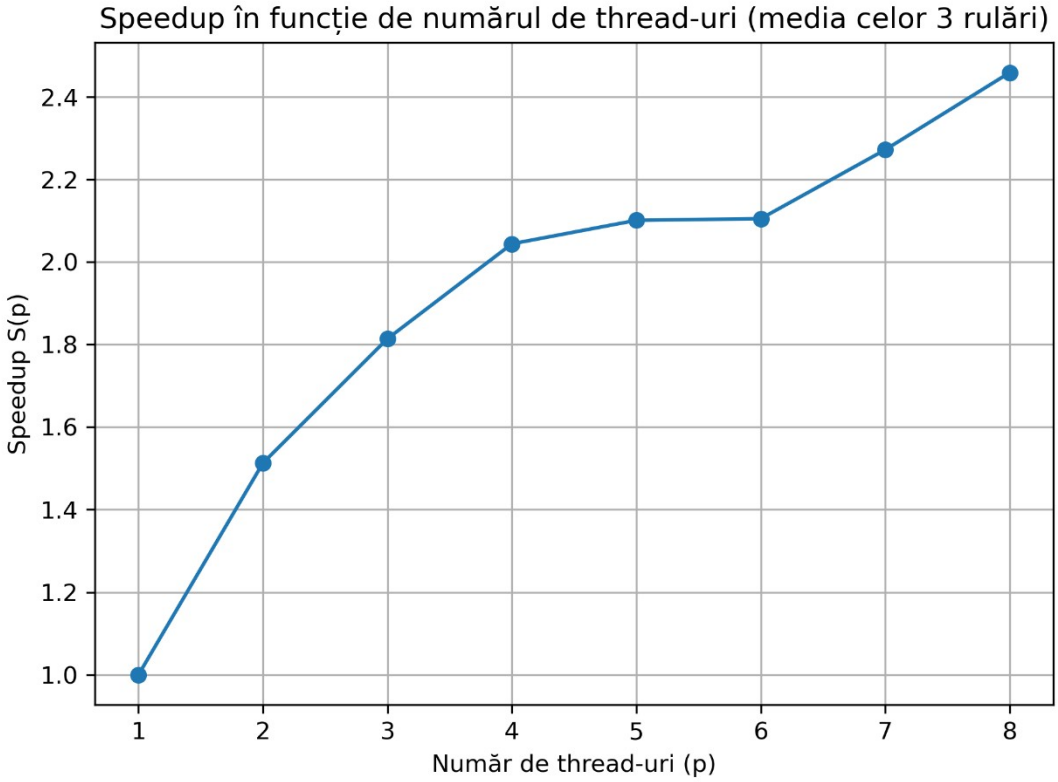
Eu zic ca solutia mea este buna si optima pentru ca : incearca sa imparta in mod cat mai egal munca pe numarul de threaduri primit, mai ales pentru partea de afisare unde nu este o idée buna sa le impartim in mod egal deoarece all_articles si keywords_count sunt mult mai mari si mai dificile si ar trebui sa ne asiguram ca ele se fac in paralel. Un alt motiv pentru care consider ca solutia mea este buna este pentru ca este threadsafe si ofera un rezultat constant pe aceleasi teste.

3. Analiza performanta si scalabilitate

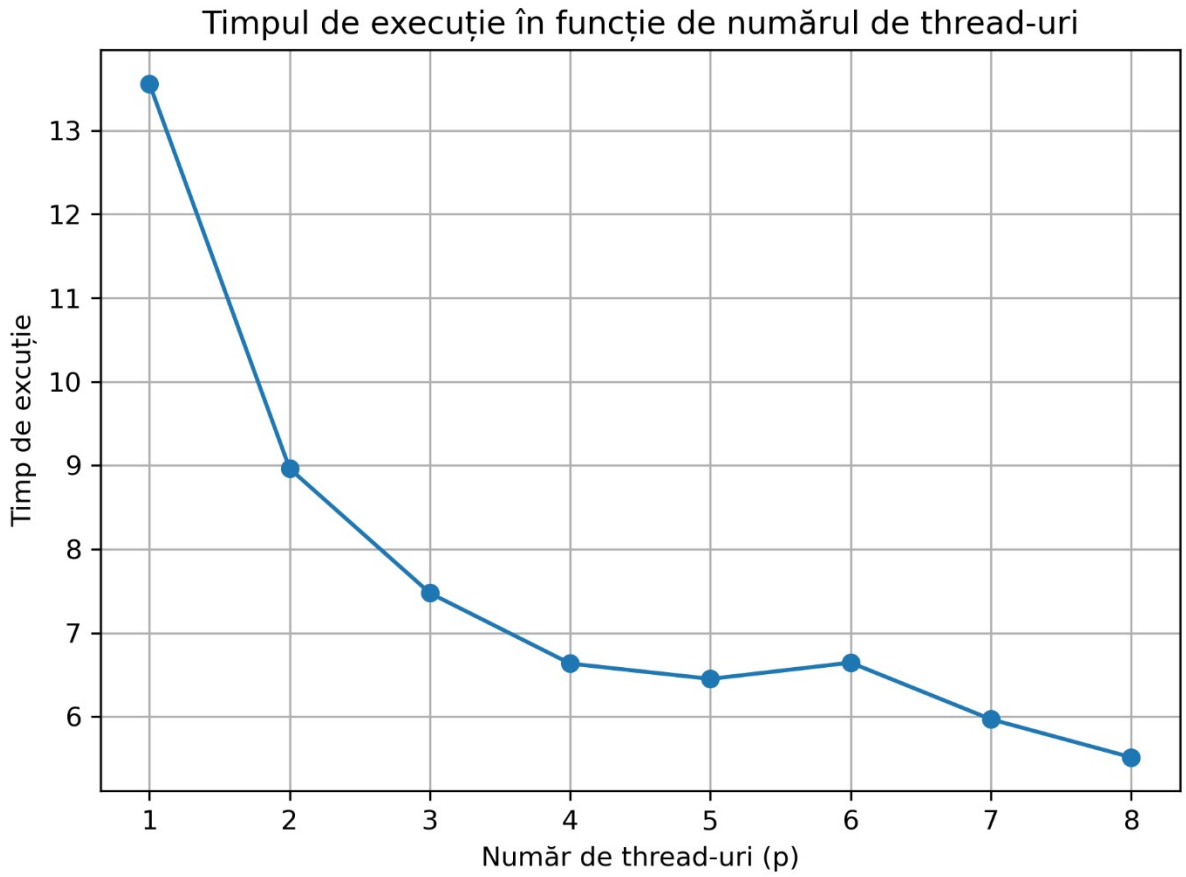
Tema a fost rulata pe un procesor 12th Gen Intel(R) Core(TM) i7-12700H (2.30 GHz) cu 14 nuclee, pe WSL Ubuntu, folosind versiunea Java 17.0.17. Pentru datasetul de test am folosit testul 5 din tema deoarece este cel mai lung.

Nr. Th R	1	2	3	4	6	8
R1	13.215	8.306	6.618	7.044	6.621	5.285
R2	13.604	8.628	8.312	6.961	6.039	5.477
R3	13.866	9.953	7.495	5.894	6.661	5.781
Avg	13.561	8.962	7.475	6.633	6.440	5.514

S(1) = 1, S(2) = 1.513, S(3) = 1.814, S(4) = 2.044, S(5) = 2.101, S(6) = 2.105, S(7) = 2.272, S(8) = 2.459



E(1) = 1, E(2) = 0.756, E(3) = 0.602, E(4) = 0.508, E(5) = 0.420, E(6) = 0.350, E(7) = 0.324, E(8) = 0.307



Comportamentul observat

- **Creşterea performanţei (Zona 1 - 4 thread-uri):**
  - Se observă o îmbunătățire drastică a timpului de execuție la trecerea de la 1 la 2, și apoi la 3 și 4 thread-uri. Timpul scade de la aprox. **13.5s** (1 thread) la aprox. **6.5** (4 thread-uri).
  - Aceasta este zona de scalare eficientă, unde adăugarea de resurse de procesare (nuclee) contribuie direct la rezolvarea mai rapidă a task-urilor paralele.
- **Stabilizarea (Zona 4 - 8 thread-uri):**
  - Începând cu **4 thread-uri**, curba timpului de execuție se aplatizează ("kneel of the curve").
  - Beneficiile adăugării de noi thread-uri (5, 6, 7, 8) devin marginale. De exemplu, între 5 și 6 thread-uri, *Speedup-ul* stagnează în jurul valorii de 2.1x.
  - Deși la 8 thread-uri se atinge minimul absolut de timp (< 6s) și maximul de speedup (~2.45x), costul adăugării a 4 thread-uri suplimentare (de la 4 la 8) aduce un câștig de performanță foarte mic comparativ cu saltul inițial.

2. Cauzele posibile ale limitărilor

Faptul că Speedup-ul maxim este doar **~2.45x** (și nu se apropie de 8x la 8 thread-uri) indică existența unor limitări clare:

1. **Overhead-ul de Sincronizare:**
    - Gestionarea structurilor de date partajate (ConcurrentHashMap, SynchronizedList) sau așteptarea la CyclicBarrier consumă timp.
    - Când ai multe thread-uri (ex: 8) care încearcă să scrie în aceleași hărți sau să acceseze aceleași resurse (chiar și cu optimizarea pe task-uri), apare *contention* (competiție pe resurse), ceea ce reduce eficiența.
2. **Dimensiunea Dataset-ului:**
    - Dacă procesarea per fișier/articol este foarte rapidă (milisecunde), timpul petrecut pentru crearea task-urilor și coordonarea thread-urilor devine comparabil cu timpul efectiv de muncă.
    - Timpul depinde foarte mult de marimea celor mai mari fisiere de scriere anume keywords_count.txt sii all_articles.txt, deoarece la afisare este posibil ca alte threaduri sa astepte dupa aceste taskuri.

3. Numărul optim de thread-uri

Bazat pe grafice, numărul optim de thread-uri pentru sistemul tău este **4**.

Justificare:

- La **4 thread-uri** obții cel mai bun raport între consumul de resurse și viteza de execuție.
- Deși rularea cu 8 thread-uri este puțin mai rapidă, eficiența per thread scade dramatic după 4.
- Graficul indică probabil că ai un procesor cu **4 nuclee fizice**.

```
Laptop-Alex
[BUILD] Building Java project...
[BUILD] make clean...
rm -f *.class *.txt
[BUILD] make build...
javac -cp ".:libs/*" *.java
[BUILD] Done

========== test_1 ==========
[RUN] Rulare secventiala cu 1 thread...
  ✓ Timp: 3.70s

[RUN] Rulare paralela cu 2 thread(s) (3 rulari)...
  ✓ Timp mediu: 2.206s, Acceleratie medie: 1.67x

[RUN] Rulare paralela cu 4 thread(s) (3 rulari)...
  ✓ Timp mediu: 1.813s, Acceleratie medie: 2.04x

========== test_2 ==========
[RUN] Rulare secventiala cu 1 thread...
  ✓ Timp: 6.48s

[RUN] Rulare paralela cu 2 thread(s) (3 rulari)...
  ✓ Timp mediu: 4.270s, Acceleratie medie: 1.51x

[RUN] Rulare paralela cu 4 thread(s) (3 rulari)...
  ✓ Timp mediu: 3.266s, Acceleratie medie: 1.98x

========== test_3 ==========
[RUN] Rulare secventiala cu 1 thread...
  ✓ Timp: 9.00s

[RUN] Rulare paralela cu 2 thread(s) (3 rulari)...
  ✓ Timp mediu: 5.710s, Acceleratie medie: 1.57x

[RUN] Rulare paralela cu 4 thread(s) (3 rulari)...
  ✓ Timp mediu: 4.273s, Acceleratie medie: 2.10x

========== test_4 ==========
[RUN] Rulare secventiala cu 1 thread...
  ✓ Timp: 12.12s

[RUN] Rulare paralela cu 2 thread(s) (3 rulari)...
  ✓ Timp mediu: 7.530s, Acceleratie medie: 1.60x

[RUN] Rulare paralela cu 4 thread(s) (3 rulari)...
  ✓ Timp mediu: 5.870s, Acceleratie medie: 2.06x

========== test_5 ==========
[RUN] Rulare secventiala cu 1 thread...
  ✓ Timp: 13.82s

[RUN] Rulare paralela cu 2 thread(s) (3 rulari)...
  ✓ Timp mediu: 9.123s, Acceleratie medie: 1.51x

[RUN] Rulare paralela cu 4 thread(s) (3 rulari)...
  ✓ Timp mediu: 6.700s, Acceleratie medie: 2.06x

==========================

Scalabilitate: 45/45
Corectitudine: 30/30
Total:        75/75
alex@Laptop-Alex:~/tema1/checker$
```