Міністерство освіти і науки України
Національний технічний університет України „КПІ”
Факультет інформатики та обчислювальної техніки

Кафедра автоматизованих систем обробки
інформації та управління

ЗВІТ

з лабораторної роботи № 1
дисципліни
“ТЕХНОЛОГІЇ ПАРАЛЕЛЬНОГО ПРОГРАМУВАННЯ В УМОВАХ ВЕЛИКИХ ДАНИХ”
на тему:

„Паралельні обчислення в моделі зі спільною пам’яттю”

Виконали:
студенти групи ІТ-01мн
Панасюк Станіслав
Лесогорський Кирило

Перевірив:
доц. Жереб К. А.

Київ – 2021

Зміст

1. Постановка задачі

Для обраної задачі необхідно реалізувати послідовну (однопоточну) реалізацію, а також мультипоточну реалізацію зі спільною пам’яттю. У якості задачі було обрано побудову системи пошуку схожих зображень. У ядрі системи лежатиме використання D-hash для знаходження хешу зображення. D-hash дозволяє точно та швидко шукати схожі зображення. Він стійкий до скейлінгу зображеня, але погано справляються з обрізаними та повернутими під кутом зображеннями. Тому цю техніку аугментовано за допомогою наступного прийому: при завантаженні зображення воно буде аугментовано за допомогою декількох фільтрів, при цьому для кожного фільтру буде згенеровано хеш і збережено у базу даних. При пошуку зображення буде використовуватись оператор XOR для знаходження зображень зы схожеми хешами.

2. Обрані інструменти

Для виконання перших двох частин лабораторної роботи буде використано стандартні інструменти Java. З самого початку буде використано фреймворк *Spring* для створення веб-інтерфейсу у майбутньому. *Spring Data* буде використано для доступу до бази даних. *Lombok* буде використано для зменшення кількості бойлерплейту. *JUnit* буде використано для тестування. Вбудована быблыотека *AWT* буде використана для роботи з зображеннями.

3. Високорівнева архітектура системи

На високому рівні система виглядає наступним чином:

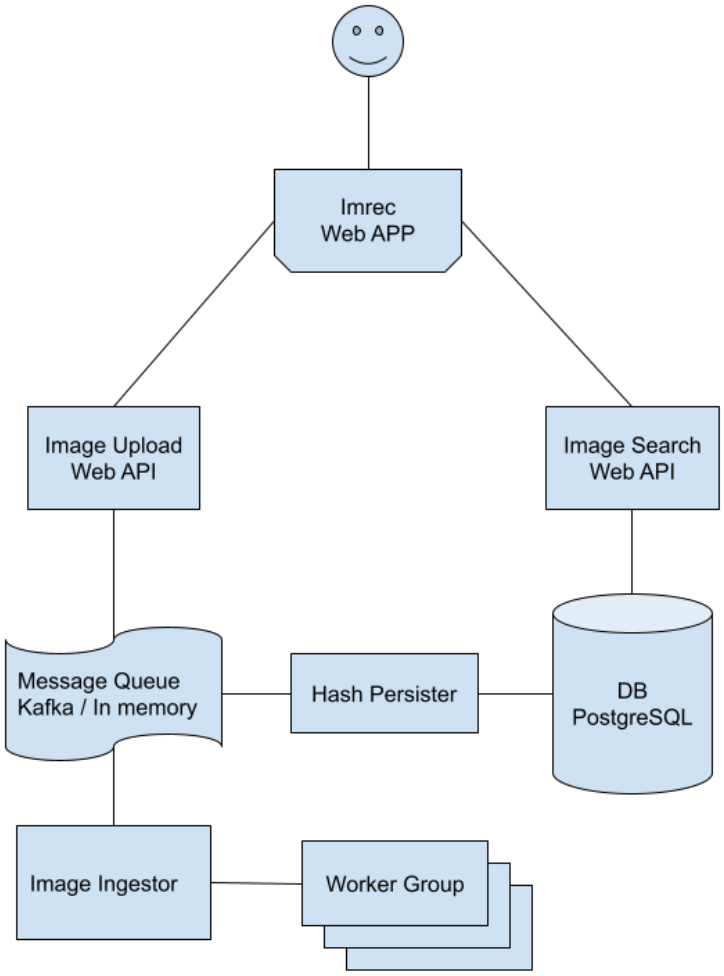

Рис. 3.1 – Високорівнева архітектура

1. **Imrec Web APP –** браузерний додаток, який надає інтерфейс для користувача. Підтримується додавання нового зображення і пошук у переліку уснуючих.
2. **Image Upload Web API** – веб сервер для завантаження зображень. Виконує валідацію запиту, зберігає файл у персистентне сховище і передає його на подальшу обробку у Image Ingestor через Kafka або InMemory чергу.
3. **Image Ingestor** – оркестратор процесу обробки зображень. Отримує повідомлення з черги і передає їх на обробку воркерам. Після обчислення хешу, передає інформацію назад у чергу.
4. **Worker** – обчислює необхідні хеші для зображення.
5. **Hash Persister** – зберігає отриманні хеші у персистентне сховище.
6. **Image Search Web API** – веб сервер, який выдповідає за пошук серед вже існуючих зображень.

Така архітектура має низьку зчепність і дуже модульна. Слід зазначити, шо кожен воркер буде обчислювати хеш для зображеня з фільтрами, це дозволяє знизити затримку при обратній збірці результатів при розподілені. Також це дозволить винести усю роботу, яка потребує багато обчислювальних ресурсів на окремі машини, дозволяючи інджестору працювати у однопоточному режимі при умові використання асинхронних інтерфейсів вводу-виводу. Також такий підхід дозволить знизити навантаження на мережу, коли декілька воркерів завантажують досить великі фотографії (більше 10 мегабайт) з файлового сховища.

4. Опис роботи програмного забезпечення

У цій секції ми розглянемо задачі, виконані у рамках цієї лабораторної роботи. Метою було створити ядро системи, а саме прототип воркерів та інджестора, які можна буде перевикористати при подальшій розробці. Окрім цього було створено декілька допоміжних файлів, які будуть використані у наступній лабораторній роботі.

Почнемо з визначення інтерфейсу воркера. Це нам знадобиться для того, щоб потім полегшити винесення воркерів у окремі процеси. Для цього створимо дуже простий інетрфейс. Окрым цього, нам знадобиться інтерфейс фабрики воркерів, який буде використаний інджестором для аллокації воркерів. Далі нам необхідно реалізувати воркер, для чого нам необхідно два компоненти: хешер для обрахунку хешу і фільтр для створення картинки, повернутої під певним кутом. Для хешеру створимо відповідний інтерфейс та імплементація, та імплементація фільтру. Далі створимо дві імплементації воркеру: одно і багато поточну. Багатопоточна імплементація приймає ExecutorService, що дозволяє контролювати кількість використаних потоків. Далі створимо локальний інджестор, який теж може створювати нові воркери, для цього ми будемо використовувати також ExecutorService і передавати його у параметрах. Останнім компонентом стануть тести, які дозволяють заміряти швидкодію.

5. Отримані результати

У результаті виконання було отримано наступні результати тестування. При тестуванні було використано 50 зображені, для кожного з яких генерувалось 7 повернутих версій і обчислювалось 8 хешів:

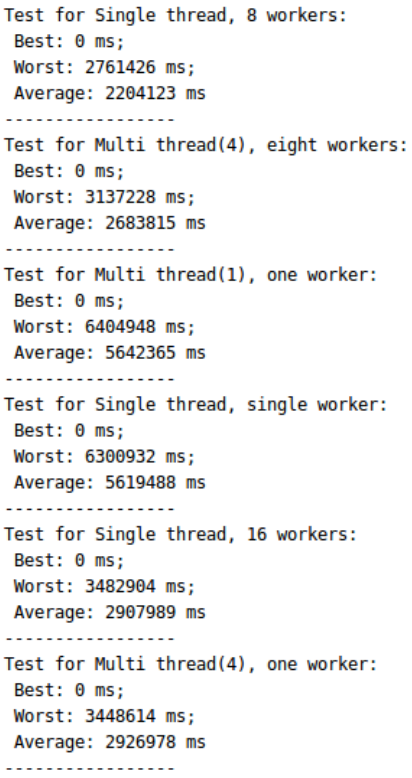

Рис. 5.1 – Результати тестування

Представимо результати у вигляді таблиці і скористуємось законом Амдала для аналізу послідовної частини прогарми. Рядки представляють з собою кількість тредів у кожного воркеру, а стовпчики представляють кількість воркерів в інжесторі. В теста час виміряється у **мікро секундах**, для зручності у таблиці значення будуть конвертовані у мілісекунди

Threads/ Workers	1	4	8	16
1	5619	-	2204	2907
4	2926	-	2683	-
8	-	-	-	-
16	-	-	-	-

Таблиця 5.1 – Залежність часу від кількості воркерів та потоків

Слід зазначити, що ЕВМ, на якій виконувались тести має 8-ми ядерний процесор. З результатів видно, що найліпше себе проявила конфігурація Single Thread, 8 workers. Цьому є декілька пояснень: по-перше, використовується максимальна кількість потоків, доступна на ЕВМ, тому знижаються витрати на переключення контексту потоків при виконанні обчислень. Також однопоточна реалізація воркерів не має витрат на синхронізацію результатів обчислень. Це добре видно при порівнянні одно поточної версії і багатопоточної версії з одним потоком.

Тепер обрахуємо степінь паралелізації обчислень за законом Амдала при збільшені кількості воркерів і збільшені потоків у воркерах:

5.1 Закон Амдала при збільшені кількості воркерів:

Дано: S = 5619 / 2204 = 2.54; P = 8;
Після підстановки у формулу отримаємо: a + (1 – a) / 8 = 1 / 2.54;
звідси 8a + (1 – a) = 8 / 2.54;
7a = 2.14
a = 0.3

Звідси послідовна частина та витрати на синхронізацію алгоритма складають 30%.

5.2 Закон Амдала при збільшені кількості потоків у воркерах:

Дано: S = 5619 / 2926 = 1.92; P = 4;
Після підстановки у формулу отримаємо: a + (1 – a) / 4 = 1 / 1.92;
звідси 4a + (1 – a) = 4 / 1.92;
3a = 1.08
a = 0.36

Звідси послідовна частина та витрати на синхронізацію алгоритма складають 36%.

5.3 Результати для інших оптимальних варіантів

Додатково розглянемо два варіанти: 4 воркери, 2 потоки у воркері та 2 воркери, 4 потоки у воркері:

```
Test for Multi thread(2), four workers:
 Best: 0 ms;
 Worst: 3067285 ms;
 Average: 2341633 ms
-----------------
Test for Multi thread(4), two workers:
 Best: 0 ms;
 Worst: 3245330 ms;
 Average: 2399643 ms
-----------------
```

Рис. 5.2 – Результати додаткового тестування

Як бачимо, результати дуже близькі до оптимальних, різницю у швидкості можна пояснити зовнішніми обставинами, але з цих результатів можна побачити,

що синхронизація між декількома воркерами дешевша, аніж синхронізація декількох потоків у воркерах(цей патерн можна побачити у всіх результатах). Це досить легко пояснити – при обробці K зображень, необхідно синхронізувати роботу *K* потоків. Якщо ж синхронізувати на рівні воркеру, необхідно синхронізувати *K * (min(wthreads, 8))* потоків.

6. Висновки

У ході лабораторної роботи було розроблено прототип інджестору та воркерів для обчислення dhash. Також було спроектовано архытектуру майбутньої системи, яку буде побудовано протягом наступних лабораторних робіт. Для отриманого алгоритму було розраховоно відсоток послідовної обробки за законом Амдала та проведено заміри швидкодії.