Paradigma Map-Reduce (Programare Paralela)

Structuri de Date

Rezultatul Mapper-ilor

Fiecare thread Mapper produce un rezultat, acesta fiind un vector de perechi **word - fileID**. De vreme ce acelasi Mapper poate colecta acelasi cuvant din fisiere diferite, **MapperResult** nu se poate implementa drept un dictionar **word - fileID**, ci ca un vector, in care aceste perechi sa fie unice.

WordList-ul

WordList-ul implementat de mine foloseste doua structuri de date (intermediare):

- Un vector de dictionare
- Un vector de liste

> Cate un dictionar si cate o lista pentru fiecare litera din alfabet.

Variabile Partajate

In main, se va crea o instanta a unei clase **SharedVariables** care retine toate valorile partajate intre thread-uri. La crearea unui thread, **SharedVariables** de constructia argumentului thread-ului, fiecare thread primind referinte la aceste variabile partajate. In functie de tipul thread-ului creat, se vor referentia variabile diferite (thread-urilor Mapper/Reducer li se vor da doar referintele de care au nevoie, nu mai mult).

Variabilele partajate intre thread-uri contin atat datele de interes (numele fisierele de intrare, rezultatele Mapper-ilor ...), cat si primitivele de sincronizare din **pthread.h** (mutex-uri, o variabila conditionala si o bariera).

Aceasta clasa (**SharedVariables**) permite crearea cu usurinta a argumentelor thread-urilor.

In plus, tot **SharedVariables** se ocupa si de:

- Alocarea si dezalocarea memoriei
- Initializarea variabilelor partajate

Programarea Paralela

Problema Producatori - Consumatori

Paradigma Map-Reduce rezolva problemea producator-consumator, in ipostaza mai multi producatori - mai multi consumatori:

- N-M: mai multi producatori (fisiere de intrare), mai multi consumatori (Mapperi)
- N-M: mai multi producatori (rezultatele Mapper-ilor), mai multi consumatori (cate un dictionar din WordList pentru fiecare litera din alfabet)
- N-M: mai multi producatori (cate un dictionar din WordList pnetru fiecare litera din alfabet), mai multi consumatorii (cele 26 de sectiuni din WordList - implicit si fisierele de iesire corespondente - pentru

fiecare litera mica din alfabetul englez)

> NOTA: Totusi, daca as fi implementat WordList-ul drept un dictionar/lista mare, si nu l-ar fi impartit pentru litera din alfabet, ultimele doua puncte din lista ar fi fost **many to one**, respectiv **one to many**.

Thread-urile Mapper

Mapperi rezolva se rezolva problema producator-consumator, in ipostaza mai multi producatori (fisiere de intrare), mai multi consumatori (rezultatele mapper-ilor)

In main, thread-urile cu indicii in intervalul [0, argv[1]) vor reprezenta thread-urile Mapper.

Un thread Mapper:

- Va primi ca argument o referinta la clasa **MapperThread**
- Intr-o bucla infinita
    - Va pune lock pe mutex-ul cozii indicilor fisierelor de intrare
        - Daca coada nu mai contine niciun element, inseamna ca toate fisierele au fost parcurse si thread-ul curent nu mai are nimic de facut , astfel, da unlock la mutex si iese din bucla
        - Altfel, extrage un elemnent din coada, reprezentand indexul unui fisier de intrare si da imediat unlock la mutex-ul cozii. Astfel, niciun alt thread nu va avea acces la acel fisier afara de thread-ul curent, drept pentru care Mapper-ul isi poate incepe munca:
            - Parcurge fisierul, citind rand cu rand, caracter cu caracter, folosind o varibila auxiliara (**word**) pentru constructia cuvintelor. Atunci cand intalnm o litera (fie ea litera mare sau mica), transformam litera in litera mica si adaugam in **word**. In momentul in care intalnim un spatiu sau ajungem la capat de rand, adaugam **word**-ul construit intr-un **set** (**set**-ul va contine toate **cuvintele unice din fisier**)
            - Adaugam fiecare **cuvant** din **set**-ul predecent construit, alaturi de **ID**-ul fisierul parcurs, la rezultatul mapper-ului cu indicele egal cu ID-ul thread-ului mapper. De vreme ce inserararea in rezultatele mapper-ilor se face in paralel la indici diferiti (fiecare thread mapper insereaza doar in bucata da rezultat), nu mai este nevoie sa protejam aceasta variabila printr-un mutex
- Cand a iesit din bucla (coada indicilor fisierelor de intrare este vid), asta inseamna ca thread-ul curent nu mai are nimic de facut, adica si-a terminat cu succes munca, drept pentru care incrementeaza o variabila care numara cati Mapperi si-au indeplinit rolul (variabila partajata atat intre toate thread-urile, atat Mapper, cat si Reducer, variabila la al cerui access este protejat printr-un mutex). In momentul in care valoarea aceastei variabile pentru numarul de Mapperi finalizati este egala cu numarul total de Mapper, trimite thread-urilor Reduce un semnal de **broadcast** printr-o variabila conditionala (**pthread_cond_t**), **broadcast** care anunta ca toti Mapperi s-au terminat, iar Reducerii pot incepe

Thread-urile Reducer

Reduceri rezolva problema producator-consumator, in ipostaza **mai multi producator - mai multi consumatori**:

- Mai intai: mai multi producatori (rezultatele Mapper-ilor) si mai multi consumatori (cate un dictionar in WordList pentru fiecare litera din alfabet)

- Apoi: mai multi producatori (cate un dictionar pentru fiecare litera din alfabet) si mai multi consumatori (pentru fiecare dictionar: cate un vector si un fisier de iesire).