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

Clase pentru thread-urile Mapper/Reducer

Cele doua tipuri de thread-uri vor avea cate o clasa, iar instantele lor vor avea referinte la variabilele partajate intre thread-uri (mai putin ID-urile thread-urilor, acestea sunt unice si nu se vor partaja intre thread-uri).

Totusi, avand in vedere semnatura (header-ul) functiei `pthread_create`, metoda furnizata ca argument lui `pthread_create` nu are voie sa apartina vreunei instante. Eu am implementat metoda pe care thread-urile o executa ca fiind metoda **statica** (`static void* routine(void *arg);`) in cadrul acestor clase. Metodei `routine` i se va da ca argument o referinta la o istanta din cele doua clase (**MapperThread**/**ReducerThread**). In interiorul functiei `routine`, argumentul se va dereferentia la un tip de date concret.

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 pentru 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 o instanta** a clasei **MapperThread** (clasa contine referinte la toate variabilele partajate de care are thread-ul nevoie)
- 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 vida**), 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 carui **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).

Un thread Reducer:

- Primeste ca argument **o referinta la o instanta** a clasei **ReducerThread** (clasa contine referinte la toate variabilele partajate de care are thread-ul nevoie)
- Pune **lock** pe **mutex**-ul ce protejeaza variabila pentru numarul de Mapperi finalizati
- Verifica intr-un **if** daca numerul de Mapperi finalizati este egal cu numarul total de Mapperi
    - Daca egalitatea nu este indeplinita, thread-ul inca mai asteapta semnalul de **broadcast** de la **variabila conditionala** (pentru a-si putea incepe)
- Da **unlock** la **mutex**-ul mentionat anterior

> Practic, ce se intampla aici e ca un singur thread Reducer (primul care ajunge sa puna **lock** pe **mutex**) va astepta semnalul de **broadcast** de la variabila conditionala. Restul thread-urilor Reducer, nici nu vor primi acest semnal. Restul vor astepta sa ia si ei **lock**-ul pentru **mutex**-ul variabilei care le spune cati Mapperi au fost finalizati, iar in momentul in care ceilalti Reduceri iau **lock**-ul pentru **mutex**, numarul Maperilor completati == numarul total de Maperi, drept pentru care nu vor mai astepta **broadcast**-ul de la variabila conditionala.
>
> **TL;DR**:
>
> - Un singur thread Reducer primeste **broadcast** de la variabila conditionala
> - Restul, vor vedea ca "numarul Maperilor completati == numarul total de Maperi" si vor dea imediat **unlock** pe **mutex**

Apoi, un thread Reducer mai are de facut urmatoarele lucruri:

- Intr-o bucla infinita
    - Va pune **lock** pe **mutex-ul cozii indicilor rezultatelor Mapperilor**
    - Daca coada nu mai contine niciun element, inseamna ca toate rezultatele Mapper-ilor au fost concatenate in dictionarele WordList-urilor
    - Altfel, extrage din coada indexul unui rezultat produs de un Mapper si imediat da **unlock** pe **mutex**-ul care protjeaza **coaada indicilor rezultatelor Mapperilor**
    - > Pana aici, logica este identica ca mai inainte ca la Mapperi: avem o coada cu indecsi, variabila partajata pe care o protejam printr-un **mutex**, iar de fiecare data cand vrem sa extragem un index din coada, punem **lock**, extragem si dam **unlock**

    - Acum ca am extras indexul unui rezultat produs de un Mapper, iterez toate perechile de forma **word - set de fileIDs** din acel rezultat, iar in functie de prima litera a cuvantului respectiv (**word[0]**), inserez in dictionarul asociat acelei litere:
        - Daca cuvantul se afla deja in dictionar, adaug set-ul de fileID-uri ale cuvantului la set-ul mapat de catre cuvant in dictionar (reuniunea celor doua seturi)
        - Daca cuvantul nu se afla in dictionar, il adaug, avand va valoarea setul de fileID-uri
        - Lista de dictionare pentru fiecare litera din alfabet este si aceasta o variabila partajata intre thread-uri, dar, de vreme operatiile de inserare se fac in paralel, uneori la indecsi diferiti, este mai eficient si reduce cu mult mai mult overhead-ul sa creez cate un mutex pentru fiecare dictionar (26), decat sa creez un mutex pentru tot vector Astfel, voi avea 26 de **mutex**-uri, cate unul pentru fiecare litera din alfabet. Iarasi, ma folosesc de prima litera a cavantului, pentru a pune **lock** pe **mutex**-ul asociat dictionarului ei din WordList, mai apoi inserez in dictionarul aferent literei, iar la final, dau **unlock** la **mutex**. In acest mod, ma asigur ca 2+ thread-uri pot scrie simultan la indici (litere) diferiti in dictionarele din WordList.

        - > Asa...daca as fi avut un singur mutex pentru tot WordList-ul, toate thread-urile ar fi trebuit sa-l astepte pe unul sa-si faca operatiile de inserare...si practic nu ar fi paralelizat nimic.

        - > In plus, aceasta sectiune g**rupeaza intrarile** din WordList **in functie de litera cu care cuvintele incep**.

- Cand a iesit din bucla (coada indicilor rezultatelor Mapper este vida)
    - Asta inseamna ca thread-ul curent nu mai are niciun rezultat Mapper pentru care sa faca ceva
    - Va astepta la **bariera**
        - Bariera va astepta toate thread-urile Reducer
        - Cand toate thread-urile Reducer au ajuns la bariera inseamna ca toate rezultatele Mapper-ilor au fost concatenate in WordList (dictionarele pentru fiecare litera din alfabet).
        - De abia cand toate thread-urile Reducer au ajuns la bariera, bariera le va lasa sa treaca mai departe

        - > Am nevoie de bariera, pentru a impune ca toate dictionarele sa fie construite in totalitate inainte sa fie sortate si scrise in fisiere.

- Mai repet inca o data logica cu alta coada, pentru a converti dictioanerele la vectori, a sorta vectorii si a-i scrie in fisierele de iesire:
    - Intr-o bucla infinita:
        - Thread-ul pune **lock** pe mutex-ul care protejeaza **coada cu indexul literelor din alfabet**
            - Extrage un index (al unei litere) si da imediat **unlock**
            - Daca **coada** este vida, iese din bucla
        - Pentru indexul (literei) extras din coada, thread-ul curent face urmatoarele lucruri:
            - Converteste dictionarul din WordList aferent literei la un vector
            - Sorteaza vectorul construit mai inaite:
                - Crescator dupa numarul de fisiere (lungimea set-ului fileID-urilor) in care cuvantul apare
                - Alfabetic dupa cuvant

        - Scrie elementele vectorului in fisierul text de iesire asociat literei respective

Elemente de sincronizare folosite

- Multe mutex-uri (array-uri chiar, acolo unde este cazul) - pentru a proteja accesul la variabilele partajate
- O variabila conditionala
    - Semnaleaza unui thread Reducer ca toate thread-urile Mapper si-au finalizat task-urile
    - Actioneaza ca un fel de *"bariera"* intre thread-urile Mapper si Reducer
- O bariera (in WordList) - impune ca toate dictionarele fiecarei litere sa fie pe deplin construite inainte de a trece mai departe (cu convertirea la vector, sortarea si scrierea in fisiere)