Relatório *MapReduce*

Eduardo Marinho, Rafael Rosendo e Samuel Morais

Abril 2024

1 Introdução

O objetivo desse relátorio é explicar como foi implementado e como utilizar o projeto *MapReduce*, desenvolvido na linguagem Elixir e baseado no modelo de programação concorrente do Google.

Neste link se encontra o repositório com o código fonte

2 Interface Pública

A interface pública desse programa é composta apenas pela função "main", porém, essa função possui três definições: main/4, na qual o primeiro argumento deve ser uma lista com os dados, o segundo a função para o *Map*, o terceiro a função para o *Reduce* e o quarto argumento o elemento neutro para a função reduce; essa é a implementação geral do *MapReduce*, já que aceita listas com qualquer tipo de dados pretendidos pelo cliente, contando que as funções de *map* e *reduce* estejam na forma adequada. A segunda definição da main é similar a primeira, com a a exceção de que o primeiro argumento da função deve ser uma string contendo o caminho para um arquivo de texto (.txt), essa implementação, assim como a seguinte, foi feita pensando no contexto de operações com palavras de um arquivo de texto, tais operações são aquelas performadas pelas funções fornecidas pelo cliente.A terceira definição funciona de maneira equivalente a segunda, porém sem aceitar argumentos e sim usando valores padrão para facilitar os testes, essa chamada realiza a contagem de palavras do arquivo de texto *test.txt*, como visto em sala de aula. Em relação ao retorno da função main, decidiu-se por trabalhar internamente, e retornar ao usuário, os dados na forma de Maps contendo pares de valores "id" e "value".

3 Função *Master*

'A função privada principal deste projeto é a função *Master*. Esta função tem o papel de, ao receber a lista de entrada, as funções de map e reduce e o elemento neutro para a operação de reduce: **particionar a lista em sublistas, em função do número de processadores utilizados**; **realizar o *mapping* das**

sublistas de forma concorrente e receber os retornos das chamadas; **realizar a ordenação e partição dos *maps* gerados em função do valor "id"** e, por fim, **a aplicação da função *reduce* de forma concorrente e retornar os resultados ao cliente, como uma lista de *maps***.

4 Map

Na parte da função Map do programa MapReduce, temos a função *map* que é implementada pelo usuário, assim como a estrutura de organização, também proposta pelo usuário. Dentro da parte do *map*, temos algumas funções auxiliares que seguem uma estrutura de mestre e escravo: a função *map_manager* é usada para gerenciar as threads e autorizar a entrada e saída para a função *master* principal.

A função *concurrent_map*, que recebe uma lista na qual o *map* será aplicado - assim como uma função *map* fornecida pelo usuário - e o PID (identificador de processos Elixir) do processo da função *master*. Esta função desempenha o papel de chamar a função *recebe_map* em uma outra thread e retornar seu valor ao processo mestre através da função *send*, que envia algum valor, o retorno de *recebe_map*, ao PID inserido como argumento. Quanto à função *recebe_map*, esta apenas aplica a função *map* do cliente sobre os elementos da lista de entrada.

5 Shuffle

Ao adentrar na parte do Shuffle, temos uma função que ordena o conjunto de dados com base em um critério estabelecido. Na nossa implementação, foi utilizado o critério da biblioteca Enum do próprio Elixir. O segundo critério seria como implementar a organização com base no tipo de dados, seja numérico ou literal. A princípio, temos como entrada uma estrutura do tipo map, e dentro da map, temos um critério, seja id, chave ou valor. Nisso, vamos ordenar usando o Enum para cada elemento do map. E com isso a função shuffle recebe a coluna do map que precisa ser ordenada

6 Reduce

Para a implementação do Reduce, usamos uma arquitetura bastante parecida com a arquitetura implementada pelo Map, uma vez que a função master faz o shuffle e a base de dados fica ordenada, temos a entrada da função reduce, no qual foi dividida em duas funções, função reduceManager que recebe um contador e uma função definida pelo cliente, também temos a função concurrentReduce que recebe uma lista, a função reduce do cliente, um contador e o pid que representa o processo do elixir, assim aplicamos a função reduce ao dataset retornando o pedido pela função reduce do cliente

7 Outras Funções Auxiliares

Além das funções principais vistas acima, foram implementadas outras funções com papéis menores na execução do programa. Faremos uma breve explicação dessas funções: a função *receber_msgs* performa o papel de receber os valores retornados dos outros processos, tanto da etapa do *map* quanto a do *reduce*. A função *split_lista* é chamada para fazer a partição da lista de entrada antes da realização do *map*, ela recebe uma lista de dados e a retorna de forma particionada, sendo agora uma lista de listas; Essa função chama recursivamente a função *formar_sublistas*. As funções: *dividir_dataset*, *map_words* e *reduce_words* são utlizadas para o caso de contagem de palavras. *map_words* e *reduce_words* são, respectivamente, funcões para o map e reduce de contagem e a função *dividir_dataset* faz a fragmentação do texto em uma lista de palavras.

8 Conclusão

Em resumo, o projeto MapReduce em Elixir mostrou como é possível usar programação concorrente para processar grandes volumes de dados de forma eficiente. Com uma interface simples, a função "main" permite diferentes tipos de entrada, como listas e arquivos de texto, criando uma maneira simples de fazer o processamento de dados de maneira concorrente.

Tabela de Participação	
Nome	**Participação (de 0 a 10)**
Eduardo Marinho	**10**
Rafael Rosendo	**10**
Samuel Morais	**10**