MapReduce

Felipe Brigalante 8941280
Mateus R. Mazzari 8941255

MAC 5742-0219 Introdução à Programação Concorrente, Paralela e Distribuída

1. Introdução

No início do século XXI, grandes quantidades de dados começam a ficar disponíveis e como consequência diversas técnicas e algoritmos surgiram para possibilitar manipulação e análise de tais dados. Essas técnicas usualmente acabam por distribuir o processamento em diversas máquinas a fim de concluir a execução em tempo hábil, contudo tal distribuição necessita de uma grande quantidade de código extra obscurecendo o objetivo original.

Em 2003 na Google era comum a necessidade de processamento de grandes quantidades de dados e lá surge a primeira implementação do modelo de programação *MapReduce* criado por Jeffrey Dean e Sanjay Ghemawat. Nesse modelo o programador precisa somente descrever o processamento que será feito nos dados de entrada e o framework de *MapReduce* é responsável por paralelizar o processamento, provendo tolerância a falhas. O programador precisa escrever duas operações, o *Map* e o *Reduce*, ambas baseadas nas funções presentes em linguagens funcionais. Basicamente essas funções recebem e devolvem conjuntos de pares chave/valor e o programador é responsável por descrever o problema nessa estrutura.

2. Modelo de Programação

Um programa em *MapReduce* sempre recebe e devolve um conjunto de pares chave/valor e o programador sempre precisa definir o processamento através de duas funções, o *Map* e o *Reduce*.

O *Map* recebe uma porção dos dados de entrada e mapeia-os para um conjunto de pares chave/valor intermediários, que posteriormente servirão de entrada para o método *Reduce*.

O *Reduce* recebe os dados devolvidos pelo *Map* e realiza um novo processamento devolvendo também um conjunto de pares chave/valor. Em geral a saída do *Reduce* é substancialmente menor do que a entrada e tipicamente devolve conjuntos de saída com zero ou um elementos. Vale ressaltar que a função de *Reduce* recebe os dados através de um Iterador, de modo que seja possível processar dados que eventualmente não caibam na memória.

2.1. Exemplo

Considere o problema de contar o número de ocorrências de cada palavra em uma grande coleção de documentos. O usuário poderia escrever um código similar com seguinte pseudo-código:

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

O *Map* gera um par intermediário para cada ocorrência de uma palavra associando-a ao valor 1, que indica que tal palavra aparece uma vez no texto.

O *Reduce* soma todos os valores que foram associados a uma determinada palavra, no nosso caso vai somar várias vezes o valor 1 e finalmente devolve quantas vezes cada palavra aparece.

Adicionalmente o programador precisa indicar o nome dos arquivos de entrada e de saída antes de efetivamente executar o *MapReduce*.

Apesar de conceitualmente os pares chave/valor aceitarem dados de quaisquer tipo, na implementação da Google as chaves e valores são sempre *Strings* e o programador é responsável por converter os tipos na implementação dos métodos *Map* e *Reduce*.

3. Implementação

A função de *Map* é executada em várias máquinas, onde cada uma delas irá processar uma porção dos dados de entrada que foram divididos automaticamente pelo *framework* em M divisões. Já a execução do *Reduce* é dividida a partir de uma partição dos dados intermediários em R partes. O número de partes R e a função de partição podem ser especificadas pelo programador.

A execução da implementação realizada pela Google pode ser visualizada na Figura 1[1], a partir dos números indicados na figura:

1. A entrada é dividida em M partes, normalmente de 16MB ou 64MB. Depois várias cópias do programa são iniciadas em várias máquinas.

[1]Retirada de *MapReduce: Simplified Data Processing on Large Clusters*

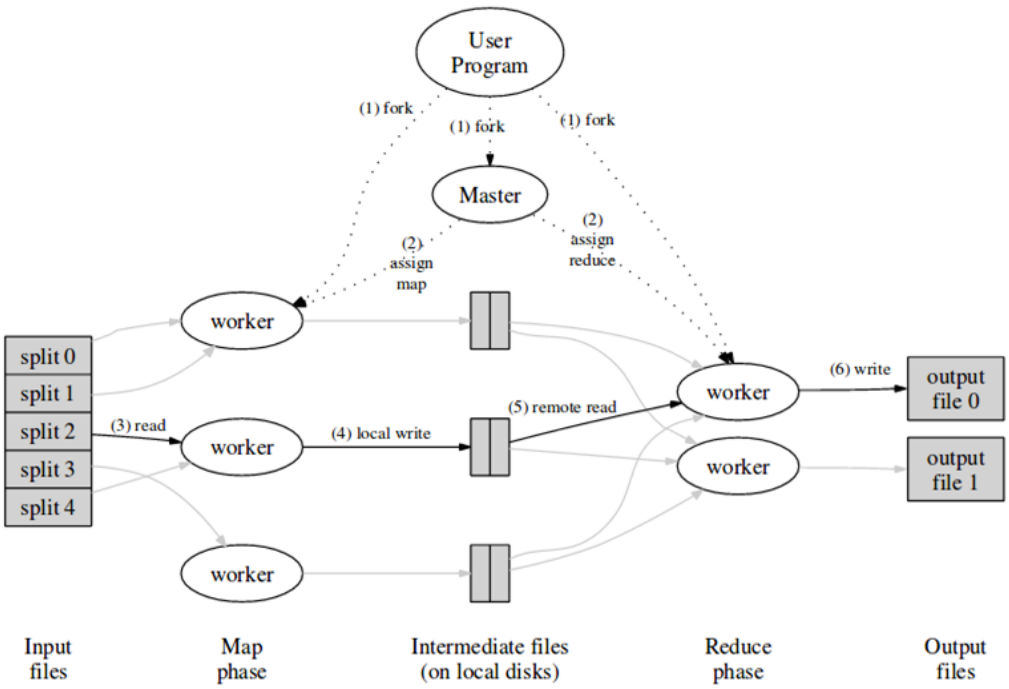

Figura 1: Implementação

2. Uma dessas cópias é a *master*, as outras são chamadas de *workers* que tem suas tarefas definidas pela *master*.
3. Uma *worker*, na qual uma tarefa *map* foi atribuída, lê a parte da entrada correspondente e a converte para pares chave/valor que são passados para a função *Map*. Os pares intermediários gerados são armazenados na memória.
4. Periodicamente, os pares armazenados na memória são escritos no disco local, particionados em R regiões pela função de particionamento. A localização desses pares são enviados para a *master* que fica responsável por encaminhar essa informação para as *workers* que realizarão o *Reduce*.
5. As *workers* de *Reduce* recebem a notificação dessa localização e leem os dados por meio de procedimentos remotos. Quando a *worker* lê todos os dados intermediários, os pares são ordenados de forma que as ocorrências de determinada chave ficam juntas.
6. A *worker* itera sobre os dados ordenados e para cada chave intermediária, passa a chave e seu conjunto de valores para a função *Reduce*. O resultado é armazenado num arquivo correspondente àquela partição.

Após o final da execução, a saída será R arquivos (um para cada tarefa de

reduce). Esses arquivos podem ser agrupados num único arquivo ou podem ser utilizados para outras chamadas de *MapReduce*.

3.1. Estruturas de Dados do Master

A *master* precisa manter algumas estruturas de dados. Para cada tarefa de *Map* ou *Reduce* ele armazena seu estado atual e em qual máquina tal tarefa está sendo executada. Além disso ele também precisa saber a localização dos arquivos intermediários gerados pelo *Map* uma vez que que essas localizações precisam ser propagadas para as máquinas que irão realizar tarefas de *Reduce*. Essas localização são armazenadas e propagadas incrementalmente a medida que as tarefas de *Map* são finalizadas.

3.2. Tolerância a falhas

Como o *MapReduce* processa grande quantidades de dados usando inúmeras máquinas, tolerância a falhas é muito importante.

Para tratar falhas nas *workers*, a *master* verifica periodicamente se há resposta de cada *worker*, caso contrário, é considerado que a *worker* falhou. Quando isso acontece, as tarefas de *Map* já realizadas por essa *worker* são desconsideradas e seu estado na *master* atualizado, o mesmo acontece com a tarefas de *map* ou *reduce* em execução, dessa forma essas tarefas podem ser re-executadas.

As tarefas de *map* executadas pela *worker* que falharam precisam ser re-executadas pois a saída intermediária fica armazenada localmente, portanto estão inacessíveis. Tarefas de *reduce*, ao contrário, são armazenadas num arquivo global.

Como só existe uma *master*, sua falha é bastante incomum e quando uma falha ocorre toda execução do *MapReduce* é abortada. Um contorno para essa situação seria armazenar periodicamente o estado das estruturas de dados da *master*, e caso ocorra alguma falha recomeçar a partir desse *checkpoint*.

3.3. Localidade

Os arquivos de entrada estão replicados em várias máquinas, para conservar banda de rede, a tarefa de *map* é executada preferencialmente numa máquina que contém a parte da entrada correspondente, se isso não é possível, tenta-se executar em uma máquina na mesma rede local da que contém a entrada. Dessa forma, a maioria dos dados de entrada é lido localmente e o consumo de banda é bastante reduzido.

3.4. Granularidade das tarefas

Como dito anteriormente, a fase de *map* é dividida em M tarefas e a fase de *reduce* em R. Para garantir uam distribuição mais equilibrada e facilitar recuperação de falhas, M e R devem ser bem maiores que o número de *workers* disponíveis.

Há limitações de quão grandes M e R podem ser, a *master* precisa fazer $O(M+R)$ decisões de agendamento e manter $O(M \times R)$ estados na memória. Além disso, R é frequentemente limitado pelo usuário já que cada tarefa gera um arquivo separado de saída. Na prática, M é determinado pelo tamanho da entrada para já que cada tarefa é responsável por um tamanho fixo da entrada.

3.5. Tarefas de Backup

Um dos casos mais comums que deixam o tempo total de execução do *MapReduce* mais longo é quando uma máquina demora para completar as últimas tarefas de *map* ou *reduce*, o motivo dessa demora pode ser competição por CPU ou memória, problemas com a banda de rede disponível, problemas de hardware.

Para prevenir que isso ocorra, quando a operação de *MapReduce* está perto do fim, a *master* agenda uma execução reserva das tarefas quem estão em progresso. Esta tarefa é marcada como completada quando ou a primeira execução ou a reserva terminarem.

3.6. Distributed File System

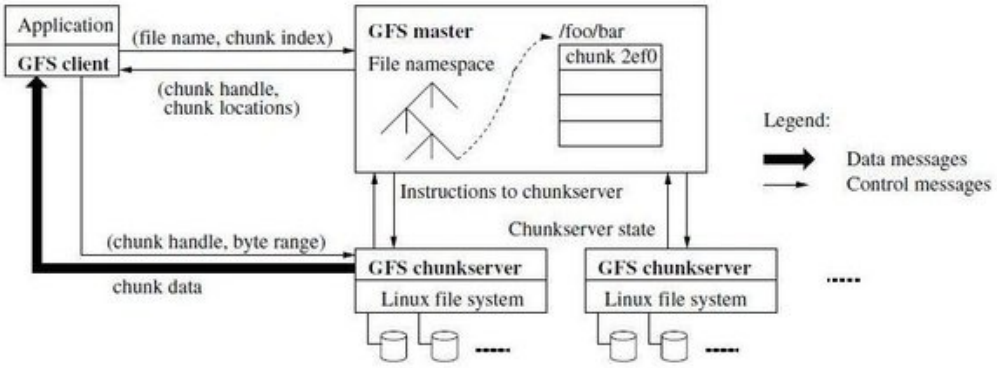

Figure 1: GFS Architecture

Figura 2: GFS

Em geral as implementações do *MapReduce* dependem de um sistema de arquivos distribuído no qual as máquinas dos *clusters* possam compartilhar os arquivos intermediários. O funcionamento do *GFS (Google File System)* e do *HDFS (Hadoop File System)* pode ser observado na Figura 2[2]. Basicamente existe um nó *master* que é responsável por controlar onde cada parte de cada arquivo fica armazenada e vários nós responsáveis por armazenar partes dos arquivos e devolvê-los para os clientes quando necessário. Vale ressaltar que esse sistema de arquivos funciona melhor quando há poucos arquivos grandes e quando não é necessário editar esses arquivos com muita frequência.

4. Refinamentos

Além das funcionalidade básicas provenientes das funções de *Map* e *Reduce*, algumas extensões úteis foram implementadas.

4.1. Função de partição

O próprio usuário pode escolher como que as chaves intermediárias serão particionadas entre as R tarefas de *reduce*. A forma padrão de partição é utilizando uma função de *hashing*, garantindo que as partições estão bem equilibradas. Porém, funções mais sofisticadas podem ser definidas pelo usuário.

4.2. Garantia de Ordem

Ao executar a função de *reduce*, é garantido que as chaves intermediárias são processadas de forma ordenada o que facilita a geração de uma saída também ordenada.

4.3. Função de combinação

Em alguns casos, há uma grande repetição de chaves intermediárias produzidas por uma tarefa de *map* e a função *Reduce* é comutativa e associativa. Nesses casos, para economizar banda de rede, é utilizada uma função *Combiner* que ao invés de enviar vários pares com a mesma chave, os pares com a mesma chave intermediária são agrupados num único par e após isso é enviado para a máquina que realizará a função de *Reduce*. Geralmente, a função *Combiner* é parecida com a função *Reduce*, a diferença é de como as saídas são tratadas, enquanto a *Reduce* armazena a saída num arquivo global, a *Combiner* é armazenada num arquivo intermediário para ser enviado para a tarefa de *reduce*.

[2]Retirada de *https://www.quora.com/What-is-the-difference-between-the-Hadoop-file-distributed-system-and-the-Google-file-system*

4.4. Tipos de entrada e saída

A implementação de *MapReduce* trabalha com entrada e saída de pares chave/valor, dessa forma a conversão dos tipos de entrada mais comuns é feita automaticamente. Para entradas mais sofisticadas, a conversão pode ser implementada facilmente pelo usuário com uma função *reader*. O mesmo acontece com a saída do *MapReduce*.

4.5. Ignorando registros ruins

As vezes podem ocorrer erros que impossibilitem que as funções de *Map* e *Reduce* sejam completadas. Casos esses erros não possam ser corrigidos, como a utilização de uma biblioteca externa, ou não precisam ser corrigidos, como análises estatísticas de uma imensa quantidade de dados, é possível ignorar esses registros.

A *master* recebe avisos de quando essas falhas ocorrem e caso a opção de ignorar registros ruins esteja ligada, na reocorrência dessa falha, a *master* indicará que este registro precisa ser ignorado.

4.6. Execução local

Para facilitar o *debug* da execução do *MapReduce* é possível executar toda operação localmente de forma sequencial, possibilitando a utilização de ferramentas para *debug* já conhecidas.

4.7. Informação de status

A *master* executa um servidor HTTP interno que proporciona uma página que contém os status da execução do *MapReduce*. Nessa página é possível verificar alguns valores da operação, como os estados das tarefas, falhas que ocorreram, taxas de processamento, entre outros. Essas informações são muito úteis para prever o tempo de execução ou consertar *bugs*.

5. Performance

As medições de performance do *MapReduce* foram feitas sobre um grande *cluster* com o objetivo de ordenar um terabyte de dados. O *cluster* consistia aproximadamente 1800 máquinas, cada uma com 2GHz Intel Xeon com Hyper-Threading, 4GB de memória, dois discos de 160GB e 1 gigabit de Ethernet.

O algoritmo *Sort* utiliza a garantia de ordenação descrita anteriormente, a função *Map* gera o par com chave igual aos dez primeiros bits da entrada e com valor igual à entrada original, a função *Reduce* funciona como uma função identidade. A parte mais importante da execução consiste na função de partição que já sabe previamente a distribuição de chaves, garantindo uma distribuição equilibrada.

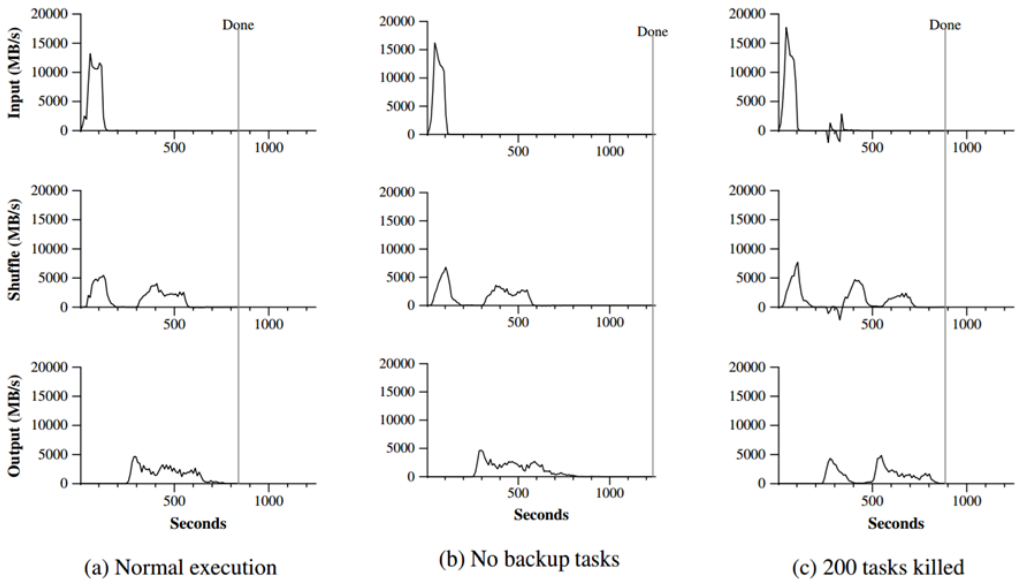

Figura 3: Performance do algoritmo *Sort*

5.1. Execução normal

A Figura 3 [3] mostra os valores obtidos na medição desse algoritmo. A coluna (a) mostra o progresso de uma execução normal, o primeiro gráfico mostra que a velocidade de leitura no *Map* é bem alta, isso se deve ao tratamento de localidade descrito anteriormente.

O segundo gráfico mostra a velocidade com a qual os valores intermediários são enviados para as tarefas de *reduce*, nota-se que o primeiro par é enviado quando a primeira tarefa de *map* é concluída. Os dois picos no gráfico representam os momentos que aproximadamente cada metade das tarefas de *Reduce* começa.

O último gráfico mostra a velocidade de escrita no arquivo final da operação, a pequena diferença entre o fim da fase anterior e o começo dessa é por causa do tempo necessário para ordenar as chaves intermediárias. Assim como na fase anterior, a velocidade é mais baixa que na leitura da entrada, pois é limitada pela largura de banda de rede.

5.2. Sem tarefas de backup

Na coluna (b) da Figura 3 percebe-se a diferença no tempo total de execução quando não são utilizadas tarefas reservas, depois de 960 segundos,

[3]Retirada de *MapReduce: Simplified Data Processing on Large Clusters*

faltava apenas cinco tarefas de *Reduce* que demoraram mais de 300 segundos para terminar, resultando num aumento de mais de 40% no tempo total.

5.3. Falhas

Na coluna (c), 200 máquinas foram intencionalmente abortadas e com isso percebe-se que rapidamente o algoritmo re-executa as tarefas que falharam, isso é representado nos picos negativos que são seguidos por picos positivos. Com isso, o tempo total de execução é minimamente maior do que o tempo de execução normal.

6. Críticas

Em Janeiro de 2008 David DeWitt e Michael Stonebraker, dois respeitados cientistas da computação na área e banco de dados publicaram o artigo *MapReduce: A major step backwards* com uma série de críticas ao *MapReduce*. Basicamente o artigo cita várias funcionalidades presentes em sistemas de bancos de dados modernos e ausentes no *MapReduce*, como por exemplo:

1. Não há um separação clara entre o que são dados e o que é lógica do programa e não possível explicitar o esquema dos dados evitando que "lixo"fique nos dados que devem ser processador.
2. É necessário usar linguagens de baixo nível para usar o *framework*, diferentemente dos sistemas de bancos de dados que em geral SQL.
3. O acesso é realizado com *força bruta*, não é possível criar índices para acessar apenas alguma porção dos dados de forma eficiente
4. Os resultados dos processamentos do *Map* são *materializados* no disco ao invés de serem armazenados na memória e eventualmente enviados por rede para as máquinas que irão processar o *Reduce*, causando um grande *overhead* de leitura e escrita de disco.
5. De fato as operações de *MapReduce* pode ser expressas com uma consulta de agrupamento presente nos bancos na linguagem SQL e ao contrário do que a comunidade de *MapReduce* afirma não há uma grande mudança de paradigma com a chegada do *MapReduce*.

Em resposta Greg Jorgensen publicou um artigo explicando que o *MapReduce* jamais foi planejado para ser um banco de dados e portanto as críticas feitas eram inválidas.

Em 2014 a Google abertamente disse que havia abandonado o *MapReduce* pois a tecnologia não era capaz de tratar a quantidade de dados que ela necessitava, substituindo o *MapReduce* pelo Cloud Dataflow na maior parte das aplicações.

7. Aplicações

Talvez a aplicação mais icônica seja a construção do índice de buscas da Google. Até 2010 os índices eram construídos a partir de uma sequência de operações em lotes nas quais as páginas da web eram processadas e com base em seus conteúdo, os índices invertidos de busca eram atualizados. Entre outras coisas o algoritmos era capaz de calcular o PageRank das páginas assim como o relacionamento elas. Os índices eram divididos em camadas, algumas eram atualizadas mais rapidamente que outras. Para atualizar uma camada de índice, era necessário analisar todas páginas disponíveis e isso podia demorar semanas.

Em 2010, segundo a própria Google, os índices não eram atualizados tão rápidos quanto eles gostariam por esse motivo o *MapReduce* foi substituído pelo *Caffeine*. Ao contrário do seu antecessor, o *Caffeine* atualiza pequenas porções do índice de modo contínuo, ou seja, novas páginas são adicionadas rapidamente nos índices de busca. Isso se deve ao fato de o *Caffeine* ser um modelo de programação de banco de dados baseado em *BigTable*, permitindo que pequenas parte dos índices sejam modificados sem a necessidade de refazer todo o processo. Com isso, o *GFS* foi substituído por uma nova infraestrutura conhecida como *Colossus* que foi especificamente projetado para *BigTable*.

Para outras aplicações, a Google substituiu o *MapReduce* pelo *Cloud Dataflow*[4].

7.1. Outras aplicações

1. Busca baseada em padrões
2. Ordenação distribuída
3. Geração de estatísticas em arquivos de log
4. Clusterização de documentos
5. Machine Learning
6. Extração de dados para geração de relatórios

8. Outras plataformas

8.1. Apache Hadoop

O Apache Hadoop é uma implementação *open source* de MapReduce, as diferenças entre as implementações são:

[4] `https://cloud.google.com/dataflow/`

- Hadoop é implementado em Java enquanto o Google implementou em C++
- Hadoop utiliza HDFS enquanto o Google utiliza GFS

8.2. Apache Stark

Apache Stark é um *framework* também baseado nas funções *Map* e *Reduce*, mas que ao contrário do MapReduce, utiliza armazenamento em memória com o auxílio de estruturas de dados chamadas Resilient Distributed Datasets (RDDs) que ficam armazenadas em memória e são utilizados para armazenar os dados dos nós do *cluster* em cache. Como estão armazenados na memória, os RDDs podem ser acessados várias vezes de modo eficiente.

Outra diferença entre os dois *framework* é que na fase de agrupar os pares com a mesma chave antes da função *Reduce*, ao invés de utilizar um algoritmo de ordenação, o Spark utiliza uma função de *hash*. Com essas duas principais modificações, o Apache Stark mostra resultados que chegam a ser cinco vezes mais rápidos que o MapReduce, isso acontece principalmente em operações iterativas.

A limitações do Spark são as mesmas do MapReduce, a função de *Map* é limitada pela velocidade da CPU e a função de *Reduce* dependente da velocidade da rede.

9. Referências

- MapReduce: Simplified Data Processing on Large Clusters, acessado em 18/06/2017 às 18:05
- MapReduce: A major step backwards, acessado em 18/06/2017 às 18:05
- Um comparativo entre MapReduce e Spark para analise de Big Data, acessado em 18/06/2017 às 18:05
- HDFS vs GFS, acessado em 18/06/2017 às 18:05
- Google search index splits with MapReduce, acessado em 18/06/2017 às 19:20
- Our new search index: Caffeine , acessado em 18/06/2017 às 19:20