MapReduce - Conceitos e Aplicações

Tiago Pedroso da Cruz de Andrade[1]

[1]Laboratório de Redes de Computadores
Instituto de Computação
Universidade Estadual de Campinas

tiagoandrade@lrc.ic.unicamp.br

1. Introdução

Em 1965, Dr. Gordon Earle Moore observou que o número de transistores em circuitos integrados aproximadamente dobraria a cada dois anos. Essa observação passou a ser conhecida como 'Lei de Moore', e está diretamente ligada a diversas medidas tecnológicas, como por exemplo a velocidade dos processadores e a capacidade de memória. Mais do que uma lei natural, tornou-se uma meta mercadológica a ser alcançada por empresas que produzem processadores e outros componentes formados por transistores.

Com tais avanços na velocidade de processamento de informações, seria natural considerar que o aumento da velocidade de leitura e escrita de tais dados se daria nas mesmas proporções. Porém não foi o que aconteceu, pois atualmente, os sistemas computacionais possuem grande capacidade de processamento sobre pequenas quantidades de informações. A análise de grandes quantidades de informações, em espaços de tempo viáveis, tornou-se uma das novas demandas na área da computação.

Com a evolução dos sistemas de informação e o aumento da quantidade de serviços disponibilizados a seus usuários, cresce também o volume de dados que precisa ser processado pelos sistemas computacionais.

Para que a computação desta grande quantidade de informações seja realizada em tempo viável, cada vez mais faz-se necessária a exploração de paradigmas de programação paralela e processamento distribuído. Porém, desenvolver sofware para ambientes distribuídos é uma tarefa complexa, pois envolve uma série de conceitos e problemas que devem ser considerados pelos programadores, como concorrência, tolerância a falhas, distribuição de dados e balanceamento de carga.

Dividir uma tarefa em subtarefas e então executá-las paralelamente em diversas unidades de processamento não é algo trivial. Inclusive, se o tamanho e a divisão das subtarefas não forem bem dimensionados, isso pode comprometer totalmente o desempenho da aplicação. Além disso, o programador precisa extrair a dependência entre os dados da aplicação e determinar um algoritmo de balanceamento de carga e de escalonamento para as tarefas, garantindo a eficiência do uso dos recursos computacionais e a recuperação ou a não interrupção da execução da aplicação caso uma máquina falhe.

A fim de facilitar este processo, surge então o *MapReduce*, um modelo de programação paralela para processamento largamente distribuído de grandes volumes de dados proposto primeiramente pela empresa Google [Dean and Ghemawat 2008] em 2004.

Após esta proposta inicial, surgiram diversas iniciativas de implementações deste

modelo de programação em diferentes linguagens de programação. Dentre elas, a mais conhecida e divulgada está inserida no projeto Hadoop [White 2012].

2. Conceito Geral do MapReduce

O paradigma de programação *MapReduce* demonstrou ser adequado para trabalhar com problemas que podem ser particionados ou fragmentados em subproblemas. Isso porque podemos aplicar separadamente as funções *Map* e *Reduce* a um conjunto de dados. Se os dados forem suficientemente grandes, podem ainda ser divididos para serem executados em diversas funções *Map* ao mesmo tempo, em paralelo.

O *MapReduce* foi desenvolvido como uma forma de processar grandes volumes de dados distribuindo o processamento em muitas máquinas para que seja processado em um tempo aceitável. Esta distribuição implica em processamento paralelo dado que a mesma função é aplicada em todas as máquinas, porém em conjuntos de dados diferentes em cada um deles.

A metodologia para atingir tal objetivo consiste em aproveitar-se do paralelismo para dividir a carga de dados, ao invés de divivir as etapas de processamento. Em outras palavras, cada componente é responsável por processar completamente um pequeno grupo de dados, ao invés de processar todos os dados em uma determinada etapa da computação.

A ideia principal por tráz do *MapReduce* é mapear um conjunto de dados em uma coleção de tuplas $\langle chave, valor \rangle$, e, então, reduzir todas as tuplas com a mesma chave produzindo a saída final do processamento. Esta abordagem adota o princípio de abstrair toda a complexidade da paralelização de uma aplicação usando apenas as funções *Map* e *Reduce*. Esta ideia simples demonstrou ser um método eficaz de resolução de problemas usando programação paralela, uma vez que tanto *Map* quanto *Reduce* são funções sem estado associado e, portanto, facilmente paralelizável. Grande parte do sucesso do *MapReduce* foi alcançada pelo desenvolvimento de sistemas de arquivos distribuídos específicos para ele.

A base de uma aplicação *MapReduce* consiste em dividir e processar conjuntos de dados com o uso das funções *Map* e *Reduce*. A função *Map* utiliza os blocos dos arquivos armazenados como entrada, e estes blocos podem ser processados em paralelo em diversas máquinas do *cluster* computacional. Como saída, a função *Map* produz, tuplas $\langle chave, valor \rangle$. A função *Reduce* é responsável por fornecer o resultado final da execução de uma aplicação.

A computação é distribuída e controlada pelo framework, que utiliza o seu sistema de arquivos distribuído e os protocolos de comunicação de troca de mensagens para executar uma aplicação *MapReduce*. O processamento é dividido em três etapas, como mostrado na Figura 1: uma etapa inicial de mapeamento, onde são executadas diversas tarefas de mapeamento; uma etapa intermediária onde os dados são recolhidos das tarefas de mapeamento, agrupados e disponibilizados para as tarefas de redução; e uma etapa de redução onde são executadas diversas tarefas de redução, agrupando os valores comuns e gerando a saída da aplicação.

Os dados utilizados na fase de mapeamento, em geral, devem estar armazenados no sistema de arquivos distribuído. Dessa forma os arquivos contendo os dados serão divididos em um certo número de blocos e armazenados no sistema de arquivos distribuído.

Cada um desses blocos é atribuído a uma tarefa de mapeamento e a distribuição das tarefas de mapeamento é feita por um escalonador que escolhe quais máquinas executarão as tarefas. Isso permite utilizar praticamente todos os nós do *cluster* computacional para realizar o processamento. Ao criar uma função *Map* o usuário deve declarar quais dados contidos nas entradas serão utilizados como chaves e valores.

Na etapa inicial, cada tarefa de mapeamento processa as tuplas $\langle chave, valor \rangle$ que recebeu como entrada, e, após o processamento, produz um conjunto intermediário de tuplas $\langle chave, valor \rangle$, onde para cada tupla $\langle chave, valor \rangle$, a tarefa de mapeamento invoca um processamento definido pelo usuário, que transforma a entrada em uma tupla $\langle chave, valor \rangle$ diferente.

Na etapa intermediária, os dados intermediários são agrupados pela chave produzindo um conjunto de tuplas $\langle chave, valor \rangle$. Assim todos os valores associados a uma determinada chave serão agrupados em uma lista. Após essa fase intermediária, os conjuntos de tuplas são divididos e replicados para as tarefas de redução que serão executadas. Essa etapa é a que mais realiza troca de dados, pois os dados de diversos nós são transferidos entre si para a realização das tarefas de redução.

Na etapa final, cada tarefa de redução consome o conjunto de tuplas atribuído a ela. Para cada tupla, uma função definida pelo usuário é chamada, transformando-a em uma saída formada por uma lista de tuplas $\langle chave, valor \rangle$.

Existe uma grande quantidade de implementações de *MapReduce* atualmente, em diversas linguagens de programação. As principais implementações conhecidas são Google MapReduce (C++), Hadoop MapReduce (Java), Greenplum (Python), GridGain (Java), Phoenix (C), Skynet (Ruby), MapSharp (C#), Disco (Erlang), Holumbus (Haskell), BachReduce (Bash Script) entre diversas outras implementações criadas para atender demandas internas às empresas.

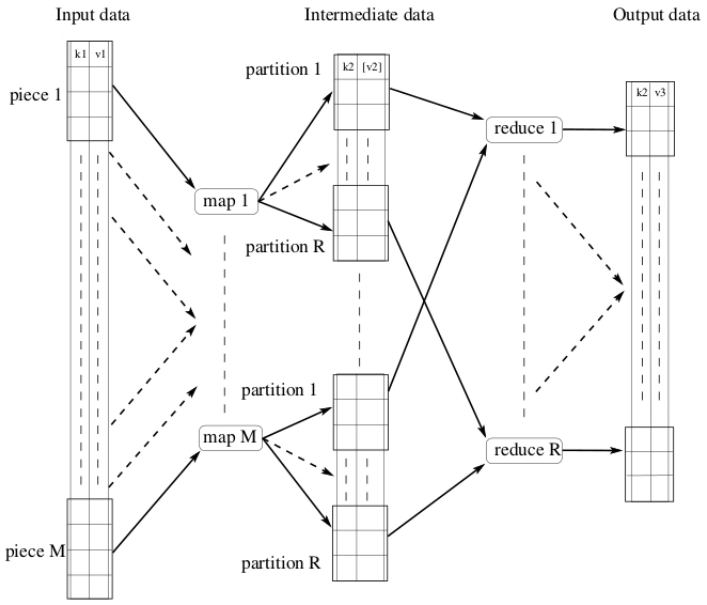

Figura 1. Fluxo geral do MapReduce

2.1. Modelo de Programação

O modelo de programação *MapReduce* consiste na construção de um programa formado por duas funções básicas: *Map* (mapeamento) e *Reduce* (redução). Durante a execução, estas funções recebem e emitem dados no formato de tuplas $\langle chave, valor \rangle$.

A função *Map* é definida pelo usuário e recebe uma tupla $\langle chave, valor \rangle$ gerando um conjunto intermediário de dados, também no formato $\langle chave, valor \rangle$. A função *Reduce*, que também é definida pelo usuário, é executada para cada chave intermediária gerada, com todos os conjuntos de valores intermediários associados àquela chave combinados. Em geral a tarefa de mapeamento é usada para encontrar algo, e a tarefa de redução é usada para fazer a sumarização do resultado.

Para exemplificar o funcionamento do *MapReduce*, considere que tenhamos uma grande coleção de ebooks (alguns terabytes de arquivos) e queremos saber o número de ocorrências de uma palavra existente nessa coleção.

Nos pseudocódigos 1 e 2, cada chamada da função *Map* recebe como chave o nome de um dos documento da coleção e como valor o conteúdo deste documento. Para cada palavra encontrada no conteúdo do documento recebido, a função emite uma tupla $\langle chave, valor \rangle$, onde a chave é a palavra em si, e o valor é a constante um. A função *Reduce*, por sua vez, recebe como chave uma palavra e como valor um iterador para todos os valores emitidos pela função *Map*, associados com a palavra em questão. Todos os valores são então somados e uma tupla $\langle chave, valor \rangle$, contendo a palavra e seu total de ocorrências é emitido.

Algoritmo 1 Map

```
Require: String key: nome do documento
         String value: conteúdo do documento
  for each word w in value do
    emitIntermediate(w, "1");
  end for
```

Algoritmo 2 Reduce

```
Require: String key: uma palavra
         Iterator value: uma lista de contadores
  int result = 0;
  for each v in value do
    result += parseInt(v);
  end for
  emit(key, asString(result));
```

As etapas realizadas no processo de *MapReduce* podem ser vistas na Figura 2 e são descritas logo abaixo.

1. O primeiro passo do *MapReduce* é dividir os dados de entrada e iniciar uma série de cópias do programa nas máquinas do *cluster* computacional.

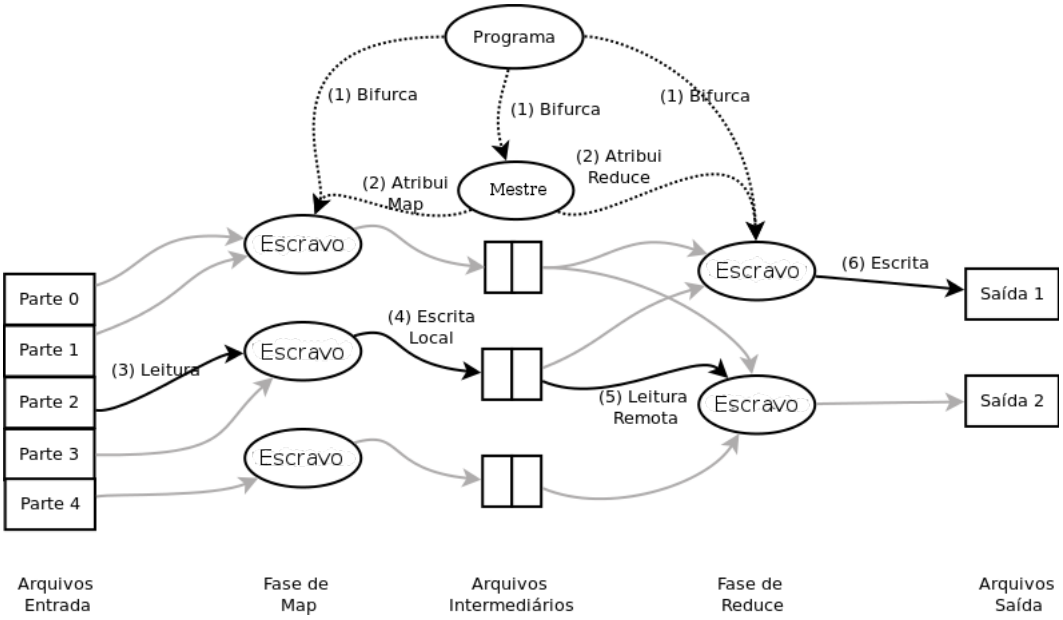

Figura 2. Fluxo geral do MapReduce

2. Uma das cópias do programa é o Mestre e as outras são todas Escravos. O trabalho consiste em realizar X tarefas de mapeamento e Y tarefas de redução, sendo o Mestre o responsável pela atribuição dessas tarefas aos Escravos.

3. O Escravo para o qual foi atribuído uma tarefa de mapeamento deve ler o conteúdo correspondente a entrada, separar todas as tuplas $\langle chave, valor \rangle$ e enviar para a função de mapeamento. As tuplas $\langle chave, valor \rangle$ produzidas pela função de mapeamento são armazenadas em memória.

4. Periodicamente, as tuplas $\langle chave, valor \rangle$ armazenadas em memória são escritas em disco, particionadas em regiões pela função de particionamento. A localização dessas regiões são passadas para o Mestre o qual é responsável por passar esta informação aos Escravos que realizam a redução.

5. Quando um Escravo que está realizando a redução é notificado pelo Mestre desta localização, ele faz um RPC (Remote Procedure Call) para realizar a leitura dos dados dos Escravos que realizaram o mapeamento. Depois de ler todos os dados, este é ordenado pelas chaves intermediárias tal que todas as ocorrências das chaves fiquem agrupadas. Se o dado não couber em memória, então é realizado uma ordenação externa.

6. O Escravo para o qual foi atribuído uma tarefa de redução deve pegar todos os valores de uma determinada chave, que foi produzido pelas tarefas de mapeamento, e enviar para a função de redução.

7. Quando todas tarefas de mapeamento e redução forem concluídas o Mestre acorda o programa do usuário e retorna o controle para ele.

   O Mestre mantêm armazenadas muitas estruturas de dados, pois para cada tarefa

de mapeamento e redução, ele precisa armazenar o estado (*idle*, *in-progress* ou *complete*) e o identificador de cada Escravo.

O Mestre é o canal através do qual a localização das regiões dos arquivos intermediários é propagado das tarefas de mapeamento para as tarefas de redução. Portanto, para cada tarefa de mapeamento completa, o Mestre armazena a localização e os tamanhos das regiões do arquivo intermediário produzido pela tarefa de mapeamento.

2.1.1. Tolerância a Falhas

- **Quando ocorrem falhas nos nós Escravos:** Para verificar se algum nó Escravo falhou, o nó Mestre testa a comunicação com todos os nós Escravos utilizando o camando *ping* periodicamente. Se nenhuma resposta for recebida do nó Escravo dentro de um certo intervalo de tempo, o nó Mestre marca esse nó como falho. Todas as tarefas de mapeamento realizadas por nós Escravos falhos são resetadas para o estado *idle*, tornando-se elegíveis para ser escalonadas por outros nós Escravos. Quando ocorre uma falha, as tarefas de mapeamento são reexecutadas, mesmo as tarefas já completadas. Isso se deve ao fato que os dados de saída do mapeamento são armazenados localmente, ou seja, os dados estão inacessíveis. Já as tarefas de redução não precisam ser reexecutadas desde que seus dados de saída estejam armazenados em um sistema de arquivos global.

- **Quando ocorrem falhas no nó Mestre:** No caso do nó Mestre falhar, é necessário um controle mais complexo, pois esse nó é o elo entre a execução das tarefas de mapeamento e redução. O nó Mestre deve executar *checkpoints* periódicos de suas estruturas de dados. Em caso de falha, uma nova instância pode ser levantada, recuperando a partir do último estado que foi salvo. O *MapReduce* assume que só existirá um único nó Mestre, portanto, falhas neste processo são indesejáveis.

Ao final da execução do programa, algumas máquinas, apesar de ainda responderem, podem apresentar um tempo de resposta muito inferior a média das outras máquinas. Para evitar que estes processos atrasem a execução do programa, quando o programa está perto de terminar, algumas cópias das tarefas restantes são iniciadas. A tarefa será marcada como completada assim que, ou a tarefa primária ou uma tarefa de backup, responder.

2.1.2. Sistema de Arquivos Distribuídos

A manipulação dos arquivos em um computador é realizada pelo sistema operacional instalado na máquina por meio de um sistema gerenciador de arquivos. Esse sistema possui um conjunto de funcionalidades, tais como: armazenamento, organização, nomeação, recuperação, compartilhamento, proteção e permissão de acesso aos arquivos. Todo o gerenciamento deve ocorrer da forma mais transparente possível aos usuários, ou seja, o sistema de arquivos deve omitir toda a complexidade arquitetural da sua estrutura não exigindo do seu usuário muito conhecimento para operá-lo.

Quando um conjunto de dados aumenta acima da capacidade de aramazenamento de uma única máquina, torna-se necessário particionar os dados entre diversas máquinas.

O sistema de arquivos que gerencia o armazenamento em máquinas em uma rede é chamado de sistema de arquivos distribuídos.

Um sistema de arquivos distribuídos possui as mesmas características que um sistema de arquivos convencional, entretanto, deve permitir o armazenamento e o compartilhamento desses arquivos em diversos hardwares diferentes, que normalmente estão interconectados por meio de uma rede de computadores. O sistema de arquivos distribuído também deve prover os mesmos requisitos de transparência a seus usuários, inclusive permitindo uma manipulação remota dos arquivos como eles estivessem localmente em suas máquinas. Além disso, deve oferecer um desempenho similar ao de um sistema tradicional e ainda prover escalabilidade.

Assim, existem algumas estruturas de controle exclusivas, ou mais complexas, que devem ser implementadas em um sistema de arquivos distribuídos. Entre essas, podemos citar algumas imprescindíveis para o seu bom funcionamento:

- **Segurança:** no armazenamento e no tráfego das informações, garantindo que o arquivo não seja danificado no momento de sua transferência, no acesso às informações, criando mecanismos de controle de privacidade e gerenciamento de permissões de acesso.
- **Tolerância a falhas:** deve possuir mecanismos que não interrompam o sistema em casos de falhas em algum nó escravo.
- **Integridade:** o sistema deverá controlar as modificações realizadas no arquivo, como por exemplo, alterações de escrita e remoção, permitindo que esse seja modificado somente se o usuário tiver permissão para tal.
- **Consistência:** todos os usuários devem ter a mesma visão do arquivo.
- **Desempenho:** embora possua mais controles a serem tratados que o sistema de arquivos convencional, o desempenho do sistema de arquivos destribuído deve ser alto, uma vez que provavelmente deverá ser acessado por uma maior gama de usuários.

2.1.3. Algumas aplicações do MapReduce

- **Agrupamento de Dados**
  Problema: É necessário guardar todos os itens que têm o mesmo valor da função em um arquivo ou executar algum outro processamento que requer que todos os itens sejam processados como um grupo. O exemplo mais típico é a construção de índices reversos.
  Solução: As tarefas de mapeamento calculam uma dada função para cada item e emite o valor da função como uma chave e o próprio item como valor. As tarefas de redução obtém todos os itens agrupados pelo valor da função e os processa ou salva. No caso de índices reversos, os itens são termos (palavras) e a função é um identificador do documento onde o termo foi encontrado.
  Aplicações: Indice reverso, ETL.

- **Filtro, Análise e Validação**
  Problema: Há um conjunto de registros onde é necessário recolher todos os registros que satisfazem uma condição ou transformar cada registro (independentemente de outros registros) em uma outra representação.

Solução: As tarefas de mapeamento obtêm os registros, um por um, e emite os itens aceitos ou suas versões transformadas.
Aplicações: Análise de log, consulta de dados, ETL, Validação de Dados.

- **Execução de tarefas distribuídas**
  Problema: Existe um grande problema computacional o qual pode ser dividido em múltiplas partes e os resultados de todas as partes podem ser combinadas para obter um resultado final.
  Solução: Descrição do problema é dividida em um conjunto de especificações e essas especificações são armazenadas como dados de entrada para as tarefas de mapeamento. Cada tarefa de mapeamento obtém uma especificação, executa os cálculos correspondentes e emite resultados. As tarefas de redução combinam todas as partes emitidas para o resultado final.
  Aplicações: Simulações Físicas e Engenharia, Análise Numérica, testes de desempenho.

3. Frameworks MapReduce

3.1. Google

O *MapReduce* criado pela Google [Dean and Ghemawat 2008] foi o primeiro modelo de programação paralela para processamento largamente distribuído de grandes volumes de dados desenvolvido e seu objetivo é facilitar a programação de aplicativos distribuídos com este perfil. Neste framework, a tarefa principal do programador é implementar as duas funções principais, *Map* e *Reduce*, indicando como o mapeamento e a redução dos dados serão realizados. Todo o trabalho de distribuição do sistema - incluindo problemas de comunicação, tolerância a falhas, concorrência, etc. - é abstraído, e fica a cargo do próprio framework.

Esse framework foi desenvolvido para grandes *clusters* computacionais de 'máquinas de prateleira'[1] interligadas por uma rede to tipo *Switched Ethernet* e é constituído por basicamente dois tipos de nós: um nó *Master* e um ou mais nós *Workers*.

O nó *Master* tem como função atender as requisições de execuções efetuadas pelos usuários e gerenciá-las, criando várias tarefas e delegando-as aos nós *Workers*. Os nós *Workers* são encarregados de executar as tarefas delegada a eles, aplicando, de acordo com seu tipo, as funções de mapeamento e redução definidas pelo usuário.

Além das etapas de mapeamento e redução, que consistem na execução de fato das funções *Map* e *Reduce* criadas pelo programador, três outras etapas são características durante a execução do framework, Figura 3. Elas são conhecidas como *Split*, *Shuffle* e *Combine*.

A etapa *Split* consiste na leitura e divisão dos dados de entrada. Após essa divisão, cada tarefa de mapeamento lê o conteúdo de seu *split* e gera tuplas $\langle chave, valor \rangle$ que serão passadas a várias chamadas da função de mapeamento definida pelo usuário. Vale ressaltar que cada *split* recebido por uma tarefa de mapeamento pode gerar várias chamadas da função de mapeamento criada pelo usuário.

[1] Computadores de baixo a médio custo, sem hardware tolerante a falhas

A etapa *Shuffle* é realizada a partir do momento em que cada tarefa de mapeamento passa a produzir tuplas intermediárias e é dividida em dois passos: particionamento e ordenação. No primeiro, as tuplas $\langle chave, valor \rangle$ são divididas em partições de acordo com a tarefa de redução de destino de cada chave. Já no passo de ordenação, as chaves pertencentes a uma mesma partição são ordenadas para facilitar o processamento posterior. É importante destacar que cada tarefa de redução pode processar várias chaves, porém uma única chamada à função de redução do usuário será feita para cada chave, isto é, uma função de redução deve processar todos os valores associados a uma determinada chave.

Como previamente comentado, uma tarefa de redução deve processar todas as partições a ela destinada. Estes dados, contudo, estão espalhados pelos nós da rede, pois o mapeamento normalmente é realizado em mais de um *Worker*. Portanto, os *Workers* devem copiar e fundir as partições a eles destinado, e que foram geradas durante a etapa *Shuffle*.

A etapa *Combine*, por sua vez, é definida por uma função do usuário, assim como as fases de mapeamento e redução, porém não é obrigatória. Ela consiste em um pré-processamento da redução e provê um significativo ganho de desempenho para certas classes de aplicações. Esta etapa ocorre em cada *Worker* que executa uma tarefa de mapeamento, após o passo de ordenação realizado durante a etapa *Shuffle*.

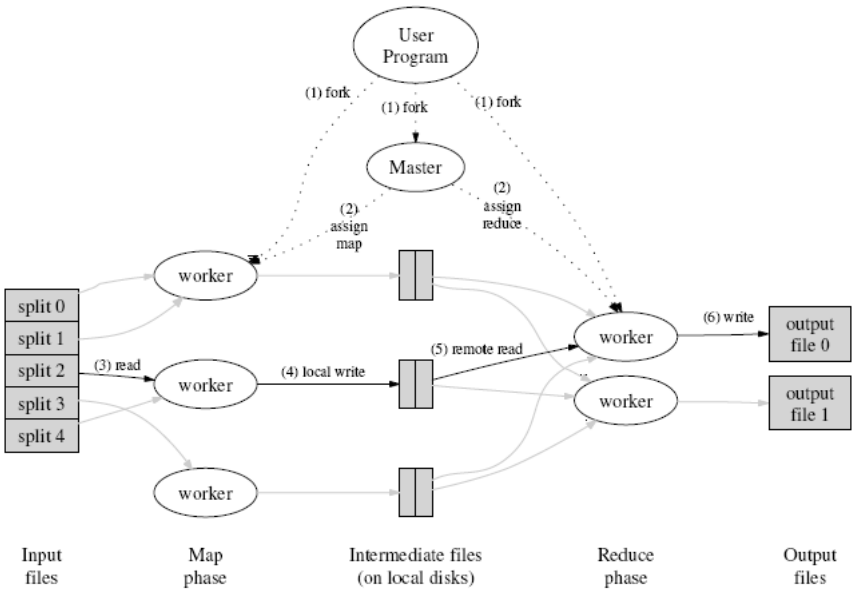

Figura 3. Fluxo do Google MapReduce

Uma das causas mais comum que aumenta o tempo total que uma operação de *MapReduce* leva é um '*straggler*'[2]. Neste caso, as tarefas executadas nessas máquinas atrasam o processamento como um todo, especialmente quando a lentidão das tarefas ocorre no final da operação.

Para contornar este problema, foram introduzidas as *Backup*

[2]Uma máquina que demora um tempo maior que o normal para completar sua tarefa

Tasks [Dean and Ghemawat 2008] no framework. Quando uma operação *MapReduce* está próxima de ser concluída, o nó *Master* escalona execuções de backup das tarefas restantes em progresso. Dessa maneira, a tarefa é marcada como concluída sempre que a execução primária ou a de backup é concluída.

À primeira vista, o processo de execução do *MapReduce* parece transferir um grande volume de dados pela rede para que os *Workers* possam processar as informações. Este problema, no entanto, não apresenta o impacto esperado devido a alguns fatores importantes. Como visto, a função *Combine* é um deles, realizando 'reduções parciais' sobre as tuplas intermediárias. É possível, ainda, observar que o *Worker* responsável por um mapeamento pode vir a realizar a redução dos dados computados durante essa fase, assim evitando a cópia destes dados para outras máquinas. Um outro importante recurso utilizado pelo framework, é a utilização de um sistema de arquivos distribuído, o qual influência fortemente a implementação do *MapReduce*.

O Google File System (GFS) [Ghemawat et al. 2003], Figura 4, é um sistema de arquivos distribuídos para aplicações com processamento intensivo de dados. Sua arquitetura consiste em um nó *Master*, e diversos nós *ChunkServers*. A arquitetura, considera, ainda, máquinas clientes que acessam o sistema de arquivos concorrentemente.

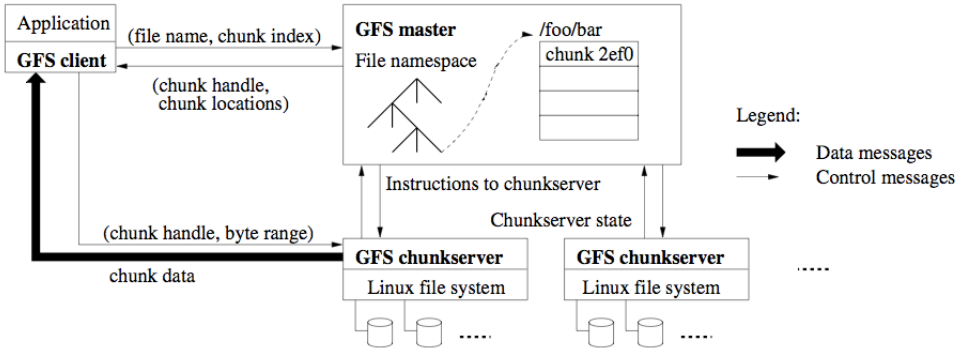

Figura 4. Arquitetura do GFS

O nó *Master* é encarregado de mantêr e controlar todos os metadados do sistema de arquivos, incluindo espaço de nomes, mapeamento de arquivos, localização dos *chunks*, dentre outros. O nó *Master* centraliza, também, diversas atividades como balanceamento de carga dos *ChunkServers*, '*garbage collection*' e atentimento a requisições dos clientes, por exemplo. Os *ChunkServers* possuem a tarefa de armazenar os dados e enviá-los diretamente aos clientes que requisitaram. Esta característica é fundamental para o bom desempenho do sistema.

Os *Workers* são também *ChunkServers*, portanto, quando o escalonador do *MapReduce* atribui uma tarefa de mapeamento a um Worker, ele tenta fazê-lo para um nó que possua, em sua unidade de armazenamento local, uma cópia dos chunks, que devem ser processados, a fim de evitar a transferência de informações pela rede. Quando esta atribuição não pode ser realizada, o escalonador tenta passar a tarefa à máquina que estiver mais próxima ao ChunkServer que possua os dados.

Assim, quando executa-se grandes operações *MapReduce*, envolvendo uma significante fração de nós do *cluster* computacional, a maioria dos dados são lidos localmente

e o consumo de banda é praticamente nulo.

3.2. Hadoop MapReduce

O projeto Apache Hadoop [White 2012] é um framework para o processamento de grandes quantidades de dados em *clusters* computacionais homogêneos. A ideia de promover soluções para os desafios dos sistemas distribuídos em um único framework é o ponto central desse projeto.

Nesse framework, problemas como integridade dos dados, disponibilidade dos nós, escalabilidade da aplicação e recuperação de falhas ocorrem de forma transparente ao usuário. Além disto, seu modelo de programação e sistema de armazenamento dos dados promovem um rápido processamento, muito superior às outras tecnologias similares. Atualmente, além de estar consolidado no mundo empresarial, o framework também tem obtido crescente apoio da comunidade acadêmica, proporcionando, assim, estudos científicos e práticos.

O Hadoop é um framework de código aberto, implementado em Java e utilizado para o processamento e armazenamento em larga escala para alta demanda de dados, utilizando máquinas comuns. Os elementos chave do Hadoop são o modelo de programação *MapReduce* , Figura 5, e o sistema de arquivos distribuído HDFS (Hadoop Distributed File System), Figura 7.

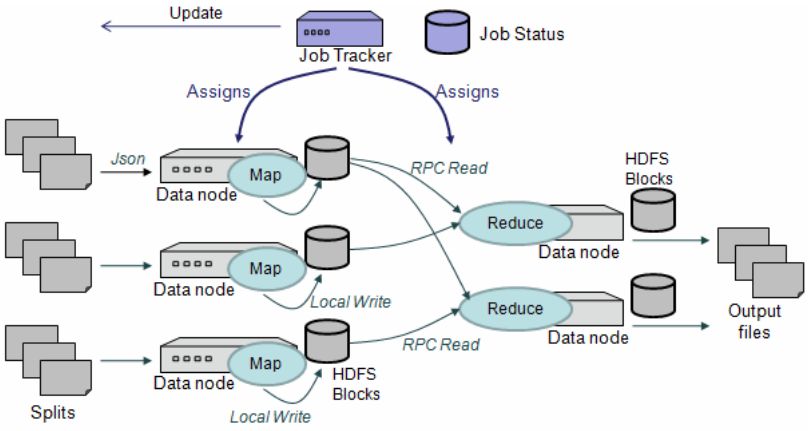

Figura 5. Fluxo do Hadoop MapReduce

Uma execução típica de uma aplicação Hadoop em um *cluster* computacional utiliza cinco processos diferentes: *NameNode*, *DataNode*, *SecondaryNameNode*, *JobTracker* e *TaskTracker*. Os três primeiros são integrantes do modelo de programação *MapReduce* e os dois últimos do sistema de arquivo HDFS. Os componentes *NameNode*, *JobTracker* e *SecondaryNameNode* são únicos para toda a aplicação, enquanto que o *DataNode* e *JobTracker* são instanciados para cada máquina.

- **NameNode:** tem como responsabilidade gerenciar os arquivos armazenados no HDFS. Suas funções incluem mapear a localização, realizar a divisão dos arquivos em blocos, encaminhar os blocos aos nós Escravos, obter os metadados dos arquivos e controlar a localização de suas réplicas. Como o *NameNode* é constantemente acessado, por questões de desempenho, ele mantém todas as suas

informações em memória. Ele integra o sistema HDFS e fica localizado no nó Mestre da aplicação.

- **DataNode:** realizam o armazenamento dos dados. Como o HDFS é um sistema de arquivos distribuído, é comum a existência de diversas instâncias do *DataNode* em uma aplicação Hadoop, para que eles possam distribuir os blocos de arquivos em diversas máquinas. Um *DataNode* poderá armazenar múltiplos blocos, inclusive de diferentes arquivos. Além de armazenar, eles precisam se reportar constantemente ao *NameNode*, informando quais blocos estão armazenados bem como todas as alterações realizadas localmente nesses blocos.
- **JobTracker:** também possui uma função de gerenciamento, porém, nesse caso, o controle é realizado sobre o plano de execução das tarefas a serem processadas pelo *MapReduce*. Sua função então é designar diferentes nós para processar as tarefas de uma aplicação e monitorá-las enquanto estiverem em execução. Um dos objetivos do monitoramento é, em caso de falha, identificar e reiniciar uma tarefa no mesmo nó ou, em caso de necessidade, em um nó diferente.
- **TaskTracker:** processo responsável pela execução de tarefas *MapReduce*. Assim como os *DataNodes*, uma aplicação Hadoop é composta por diversas instâncias de *TaskTrackers*, cada uma em um nó Escravo. Um *TaskTracker* executa uma tarefa *Map* ou uma tarefa *Reduce* designada a ele. Como os *TaskTrackers* rodam sobre máquinas virtuais, é possível criar várias máquinas virtuais em uma mesma máquina física, de forma a explorar melhor os recursos computacionais.
- **SecondaryNameNode:** utilizado para auxiliar o *NameNode* a manter seu serviço, e ser uma alternativa de recuperação no caso de uma falha do *NameNode*. Sua única função é realizar *checkpoints* do *NameNode* em intervalos pré-definidos, de modo a garantir a sua recuperação e atenuar o seu tempo de reinicialização.

Na Figura 6 pode-se observar de forma mais clara como os processos da arquitetura do Hadoop estão interligados. Nota-se uma separação dos processos entre os nós Mestre e Escravos. O primeiro contém o *NameNode*, o *JobTracker* e possivelmente o *SecondaryNameNode*. Já o segundo, comporta em cada uma de suas instâncias um *TaskTracker* e um *DataNode*, vinculados respectivamente ao *JobTracker* e ao *NameNode* do nó mestre.

Um cliente de uma aplicação se conecta ao nó Mestre e solicita a sua execução. Nesse momento, o *JobTracker* cria um plano de execução e determina quais, quando e quantas vezes os nós Escravos processarão os dados da aplicação. Enquanto isso, o *NameNode*, baseado em parâmetros já definidos, fica encarregado de armazenar e gerenciar as informações dos arquivos que estão sendo processados. Do lado Escravo, o *TaskTracker* executa as tarefas a ele atribuídas, que podem ser *Map* ou *Reduce*, e o *DataNode* armazena um ou mais blocos de arquivos. Durante a execução, o nó Escravo também precisa se comunicar com o nó Mestre, enviando informações de sua situação local.

Paralelamente a toda essa execução, o *SecondaryNameNode* registra os *checkpoints* dos arquivos de log do *NameNode*, para a necessidade de uma possível substituição no caso do *NameNode* falhar.

As execução das tarefas pode tornar-se mais lenta devido a vários fatores como a degradação do hardware e a configuração errada do software, porém, as causa são difícieis de se detectar porque as tarefas ainda serão completadas com sucesso embora levando

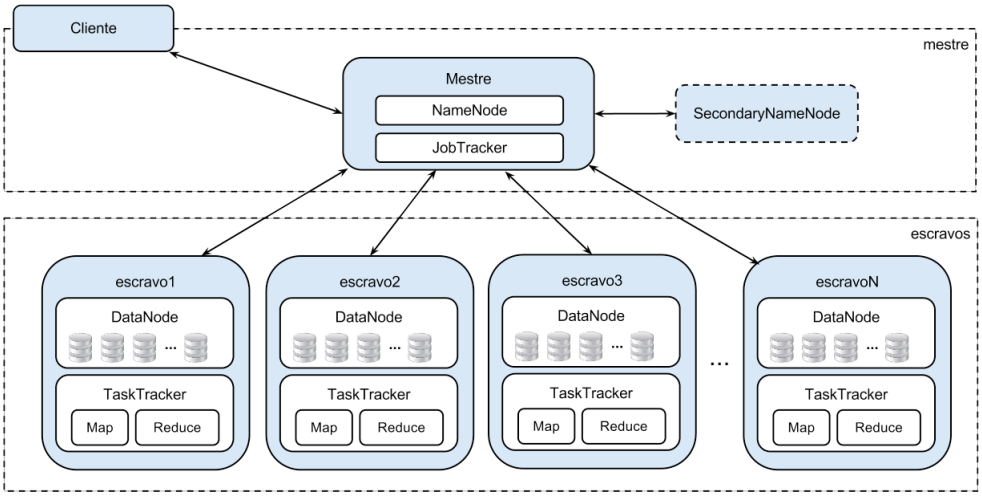

Figura 6. Interligação dos processos da arquitetura do Hadoop

mais tempo que o normal. O Hadoop não tenta diagnosticar e corrigir o problemas das tarefas lentas, em vez disso, ele tenta detectar quando uma tarefa esta mais lenta do que o esperado para executar outra tarefa equivalente como um backup. Isto é denominado *speculative execution* [White 2012].

É importante entender que uma *speculative execution* não funciona como a execução de duas tarefas duplicadas ao mesmo. Em vez disso, uma *speculative execution* é executada somente depois que todas as tarefas estão em execução e somente para as tarefas cujo o progresso está demorando mais que a média. Depois que uma tarefa é completada com sucesso, todas suas tarefas duplicadas são encerradas.

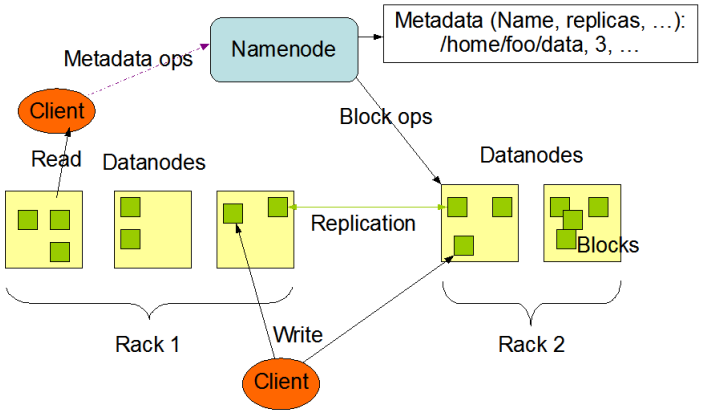

Figura 7. Arquitetura do HDF

4. Trabalhos Relacionados

Muitos sistemas têm proporcionado modelos de programação restritivos e utilizado as restrições para paralelizar a computação automaticamente. Por exemplo, uma função associativa pode ser computada sobre todos os prefixos de um array de N elementos em tempo $logN$ sobre N processadores usando computação paralela de prefixos [Blelloch 1989, Gorlatch 1996, Ladner and Fischer 1980]. *MapReduce* pode ser

considerado uma simplificação e destilação de alguns destes modelos baseados em experiências no mundo real. Mais significantemente, o *MapReduce* proporciona uma implementação tolerânte a falhas que escala para milhares de processadores. Em contraste, muitos dos sistemas de processamento paralelo tem sido implementado somente em pequena escala deixando os detalhes da manipulação das falhas das máquinas a cargo do programador.

Em [Zaharia et al. 2008] é realizado um estudo que mostra a degradação de desempenho gerada pelo escalonador de tarefas do Hadoop quando executado sobre um ambiente heterogêneo. Os testes foram conduzidos sobre o ambiente virtualizado Elastic Compute Cloud (EC2) da Amazon. É apresentado, ainda, um novo algoritmo de escalonamento, o Longest Approximate Time to End (LATE), que é altamente robusto para ambientes heterogêneos, e está relacionado com mecanismo de textitspeculative execution. De acordo com a pesquisa, este novo algoritmo conseguiu reduzir o tempo de resposta do Hadoop pela metade em determinadas situações.

Uma visão diferente de heterogeneidade é abordada em [Tian et al. 2009], onde tal propriedade é observada na natureza das tarefas (CPU-bound e I/O-bound) e não no poder computacional dos nós, mas que, apesar disso, provê uma melhora de aproximadamente 30% no throughput do Hadoop através da paralelização desses tipos de tarefas.

Em [Ranger et al. 2007] foi avaliado a adequação do *MapReduce* como um ambiente de programação para sistemas de memória compartilhada. Os autores descrevem o Phoenix, uma implementação *MapReduce* que usa memória compartilhada de modo a miniminizar os overheads das tarefas e da comunicação de dados. Com o Phoenix, o programador fornece uma expressão funcional simples do algoritmo e deixa a paralelização e escalonamento para os sistema. Foi mostrado que Phoenix leva a um desempenho escalável para os chips multi-core e multiprocessadores simétricos convencionais. Ele trata automaticamente decisões de escalonamento durante a execução paralela e também pode recuperar de erros transitórios e permanentes nas tarefas de *Map* e *Reduce*.

5. Conclusão

O modelo de programação *MapReduce* tem sido usado com sucesso para muitos propósitos diferentes. Isso se deve a muitas razões. Primeiro, o modelo é fácil de usar, mesmo para programadores sem experiência em sistemas paralelos e distribuídos, pois o framework esconde os detalhes da paralelização, tolerância a falhas, distribuição de dados e balanceamento de cargas. Segundo, uma grande variedade de problemas são facilmente expressados com o *MapReduce*. Por exemplo, o *MapReduce* é usado para geração de dados no serviço de pesquisa da Google, para ordenação de grandes quantidades de dados, para mineração de dados, para aprendizado de máquinas e muitos outros sistemas. Terceiro, o MapReduce é facilmente escalável para *clusters* computacionais compreendendo milhares de máquinas. A implementação torna o uso dos recursos da máquinas eficiente e portanto é adequado para ser usado em muitos problemas computacionais onde é necessário o processamento de grandes quantidades de dados.

Referências

Blelloch, G. E. (1989). Scans as primitive parallel operations. *IEEE Trans. Comput.*, 38(11):1526–1538.

Dean, J. and Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113.

Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43.

Gorlatch, S. (1996). Systematic efficient parallelization of scan and other list homomorphisms. In *Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II*, Euro-Par '96, pages 401–408, London, UK, UK. Springer-Verlag.

Ladner, R. E. and Fischer, M. J. (1980). Parallel prefix computation. *J. ACM*, 27(4):831–838.

Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., and Kozyrakis, C. (2007). Evaluating mapreduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 13 –24.

Tian, C., Zhou, H., He, Y., and Zha, L. (2009). A dynamic mapreduce scheduler for heterogeneous workloads. In *Grid and Cooperative Computing, 2009. GCC '09. Eighth International Conference on*, pages 218 –224.

White, T. (2012). *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 3th edition.

Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R., and Stoica, I. (2008). Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA. USENIX Association.