1

O 4^{o} Paradigma e a Computação Intensiva em Dados

Julio César Santos dos Anjos - jcsanjos@inf.ufrgs.br [1]
Iván Marcelo Carrera Izurieta - ivan.carrera@inf.ufrgs.br [2]
André Luis Tibola - altibola@inf.ufrgs.br [3]
Cláudio Fernando Resin Geyer - geyer@inf.ufrgs.br[4]

"eScience é o ponto onde "TI encontra cientistas".
Os dados atualmente são como um iceberg, as pessoas coletam
uma grande quantidade de dados e depois reduzem
tudo isto a algum número de colunas em um artigo..."
Jim Gray (NRC-CSTB, em janeiro de 2007)

[1]Doutorando em Ciência da Computação pela Universidade Federal do Rio Grande do Sul - UFRGS/RS. Mestre em Ciência da Computação, pela UFRGS/RS, em 04/2012. Graduado em Eng. Elétrica pela PUC/RS - Pontifícia Universidade Católica do Rio Grande do Sul, em 01/1991. Pesquisador nas áreas de computação em grade, sistemas distribuídos, computação intensiva em dados e MapReduce.

[2]Mestrando em Ciência da Computação pela UFRGS/RS. Graduado em Engenheira Eletrônica e Redes da Informação pela "Escuela Politécnica Nacional" Quito/Equador, em 2011. Professor substituto de Redes Wireless na Escuela Politécnica Nacional. Professor de Tecnologia em Redes e Telecomunicações na "Universidad de las Américas" em Quito/Equador. Pesquisador na área de sistemas largamente distribuídos em nuvem e MapReduce.

[3]Mestrando em Ciência da Computação pela UFRGS/RS. Graduado em Engenheira da Computação pela Universidade Federal do Rio Grande - FURG, em 2011. Pesquisador na área de tolerância a falhas em sistemas largamente distribuídos e MapReduce.

[4]Pos-Doutorado em Informática pela Universite de Grenoble I (Scientifique Et Medicale - Joseph Fourier) - França, em 1996. Doutorado em Informática pela Universite de Grenoble I, em 1991. Mestre em Ciência da Computação pela UFRGS/RS, em 1986. Graduado em Engenharia Mecânica pela UFRGS/RS, em 1978. Professor adjunto da UFRGS/RS, em Ciência da Computação. Áreas de interesse computação pervasiva (ubíqua), computação em grade, nas nuvens e voluntária, escalonamento e suporte a jogos multijogadores, e computação intensiva em dados, com ênfase na construção de middlewares.

Resumo:

Um novo paradigma da computação científica vem se formando com o crescente volume de dados de determinadas aplicações, que chegam a casa de *exabyte* de informações. Na casa de *exabytes* de informação, os sistemas e os protocolos existentes hoje não são suficientes para o tratamento destes dados. O processamento e a transferências de dados nesta escala é um grande problema em aberto [HEY 2009]. Assim, Jim Gray, em janeiro de 2007, descreveu a visão do Quarto Paradigma da ciência como um desafio interdisciplinar, em três áreas: captura, curadoria e análise. O tratamento científico de dados, por exemplo, na área da genética representa um desafio imenso, uma vez que hoje somente certas regiões do código genético de um genoma são sequenciadas. Entretanto, cada indivíduo tem um DNA único com bilhões de pares básicos. Cerca de 25 KB de dados são gerados para sequenciar algumas centenas de milhares de pares básicos de DNA para cada indivíduo, devido ao alto custo e ao tempo necessário para executar tal operação. Apenas cerca de 1% do genoma leva à proteção de proteínas, e o próximo desafio é descobrir a função dos outros 99%, que durante anos foram chamados de "DNA lixo", por não codificar para proteínas. Outras áreas como a prospecção de petróleo ou a simulação de climatologia, para a prevenção de grandes desastres, demandam grandes volumes de dados para ter-se uma resolução de pouco mais de algumas centenas de metros. Alguns experimentos como o LHC (Large Hadron Collider) do CERN, os rádios telescópios como o ASKAP da Austrália e o Pan-STARRS do Hawaii, produzem alguns petabytes de dados por dia que necessitam ser processados. Diversas técnicas vem sendo apresentadas nos últimos anos com objetivo de dar um rumo às pesquisas de sistemas para a computação de grandes volumes de dados. Este capítulo trata de questões relacionadas à sistemas e técnicas criadas recentemente para o tratamento de dados em grande escala. Estes sistemas utilizam grandes *clusters* e *grids* com dados distribuídos por centenas de milhares de computadores, onde um dos focos é levar a programação até onde o dado está. Assim, o sistema beneficia-se da localidade dos dados evitando o transporte de grandes volumes de informação pela rede em tempo de execução [HEY 2009] O leitor encontrará uma discussão sobre os paradigmas da ciência na seção 1.1. Os modelos de computação intensiva em dados existentes até o momento estão descritos na seção 1.2. Um destaque será dado ao *MapReduce* que é um *framework* de programação paralela, criado em 2004 pelo Google. O Hadoop, uma implementação *Open Source* do *framework MapReduce*, é apresentado na seção 1.3. As diferentes plataformas de implementação do *MapReduce* são apresentadas na seção 1.4. Um estudo sobre simuladores foi introduzido na seção 1.5. O objetivo é de facilitar a pesquisa em larga escala, com o uso de milhares de máquinas. Um *Hands-on* com a implementação do Hadoop é apresentada na seção 1.6. As considerações finais são apresentadas na seção 1.7.

1.1. Os Paradigmas da Ciência

No princípio a ciência era somente empírica onde havia a descrição dos fenômenos naturais, cerca de 1000 anos atrás, seguido pela ciência teórica com modelos teóricos como as Leis de Kepler, as Leis de Newton, as equações de Maxwell, teoria de Einstein, etc. Os modelos teóricos tornaram-se complexos e de difícil aplicação para diversos fenômenos de interesse. Assim, iniciam-se as simulações de fenômenos complexos em sistemas computacionais, que nortearam a ciência na última metade do último milênio [HEY 2009].

Por vezes estes sistemas computacionais geram uma grande quantidade de dados, ao ponto dos fenômenos não serem mais facilmente observados em equipamentos convencionais como telescópios ou microscópios, mas sim em instrumentos que mostram graficamente o resultado do processamento de milhares de dados. Este novo modelo captura os dados por instrumentos eletrônicos até em locais distantes *e.g.* o telescópio Web Space Telescope. Este telescópio, em uma órbita geoestacionária, coleta informações e as envia por rádio para centrais na terra onde são armazenadas para depois serem interpretadas e analisadas. As técnicas e tecnologias para esta nova ciência distingui-se da Ciência da Computação, como o quarto paradigma para a exploração científica [HEY 2009].

1.2. Modelos de Computação Intensiva em Dados

A quantidade de dados produzida pela humanidade está na casa de *Exabytes* de informações [HIL 2010]. Este grande volume de dados exige o desenvolvimento de arquiteturas e sistemas que viabilizem o uso de aplicações intensivas em dados. Diferentes soluções foram propostas para o tratamento de dados em sistemas distribuídos que resultaram no surgimento de arquiteturas como *clusters, grids, desktop grids* e *clouds*, assim como bibliotecas e linguagens de programação tais como: Java, MPI, Globus, #C, *.Net*, etc.

Porém, a computação de grandes volumes de dados associados a baixos tempos de resposta ainda é uma questão em pauta na comunidade científica [ISA 2007, WIL 2009, MIC 2009, MOR 2010]. O problema pode ser resumido de forma genérica como: Um grande volume de dados necessita, em algum momento da programação, ser transferido de um local para outro. O uso de *storages* centralizados não é uma boa abordagem para alguns casos porque cria um gargalo de I/O muito expressivo. Outro problema está relacionado à quantidade de processamento necessária para o tratamento de uma quantidade de dados determinada, neste caso, a distribuição de processos é fundamental para a execução das tarefas em um menor

tempo possível. A questão do consumo energético é outro problema que recentemente vem sendo analisado. Deseja-se produzir o processamento de dados, no menor tempo possível, mas não a qualquer custo, também deseja-se ter o menor consumo energético possível. A seguir, serão apresentados alguns dos modelos que foram desenvolvidos para o tratamento de grandes quantidades de dados, tais como *All-Pairs, Saga, Dryad, Swift, Cassandra* e *MapReduce*.

1.2.1. *All-PAIRS*

A abstração *All-Pairs* [MOR 2010] é um modelo de um produto cartesiano de um grande número de objetos com uma função de comparação personalizada. *All-Pairs* é semelhante a outras abstrações como *MapReduce*, *Dryad*, *Phoenix*, *Pig* e Swift, mas aborda uma categoria diferente de aplicativos.

O problema é expresso pela equação 1.1. *All-Pairs* compara todos os elementos de A com todos os elementos de B através de uma função $F(x)$ retornando uma matriz $M[i, j]$. Essa abstração também é conhecida como o produto cartesiano dos conjuntos A e B.

$$(A, B, F(x))|M[i, j] = F(A[i], B[j]) \tag{1.1}$$

As variações de *All-Pairs* ocorrem em muitos ramos da ciência e da engenharia, onde o objetivo é compreender o comportamento de uma função F recém-criada em conjuntos A e B, ou calcular a co-variância dos conjuntos A e B em uma norma interna do produto F. Entretanto, a função nem sempre é simétrica [MOR 2010].

Um programa sequencial é fácil de ser construído para executar esta operação em um simples computador, porém o desempenho será muito lento. Entretanto, usar um *cluster* para efetuar uma operação sequencial resulta em baixo aproveitamento dos recursos, uma vez que os dados não podem ser facilmente divididos em partições disjuntas. Como todos os dados são necessários para as comparações serem feitas, a máquina necessitará transferi-los em tempo de execução.

A proposta dos autores é sinalizar cada tarefa, composta por um par de elementos, a uma máquina e fazer as chamadas de leitura e escrita em um sistema de arquivos compartilhados sob demanda. O sistema não sabe qual é o tipo dos dados, até o momento em que a tarefa iniciar as chamadas ao sistema. Os usuários especificam os dados e a necessidade computacional, como *flops*, tempos de processamento, cpus, etc. Assim, o sistema pode particionar e distribuir os dados de acordo com a necessidade computacional.

O sistema testa a entrada dos dados para determinar o tamanho de cada elemento e o número de elementos em cada conjunto. Uma tarefa testa um pequeno conjunto de dados para determinar o tempo de execução típico para cada chamada.

A banda da rede e a latência do escalonador, para o envio de tarefas, são testadas para determinar o número de tarefas, o tamanho de cada tarefa e o número de máquinas necessárias. Os dados são entregues às máquinas via uma *spanning tree* e o fluxo de transferência se completa em um tempo logarítmico [MOR 2010].

1.2.2. Dryad

O *Dryad* é uma alternativa ao *MapReduce* do Google, proposta pela Microsoft [ISA 2007]. Ele combina vértices computacionais com canais de comunicação para formar um grafo de fluxo de dados. Os *jobs* são grafos acíclicos orientados (dígrafos), onde em cada vértice há um programa e as arestas representam canais de dados.

A proposta consiste em dar ao desenvolvedor um controle fino sobre grafos de comunicação, bem como, às sub-rotinas que exitem em seus vértices. O desenvolvedor pode especificar um grafo acíclico orientado arbitrariamente para descrever os parceiros de comunicação e expressar seus mecanismos de transporte.

A forma de especificar um grafo permite flexibilidade para compor as operações básicas com entradas de dados simples. As entradas de dados simples ou múltiplas geram uma saída simples. Entretanto, as aplicações exigem que o desenvolvedor deva compreender a estrutura computacional da computação e as propriedades dos recursos dos sistemas.

A Figura 1.1, adaptada de [ISA 2007], especifica a organização do sistema. Um *job* no *Dryad* é coordenado por um processo chamado gerenciador de *jobs* (JM). O gerenciador de *jobs* contém os códigos e bibliotecas específicas das aplicações para construir a comunicação entre as tarefas. Quando a comunicação estiver definida serão escalonados os *jobs* através dos recursos disponíveis.

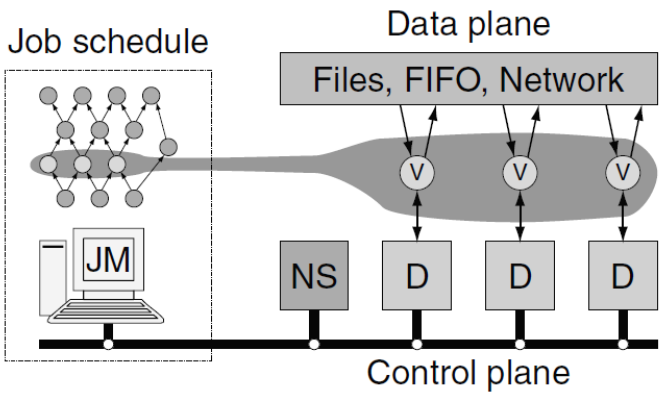

Figura 1.1: Arquitetura do *Dryad*.

Todo dado é enviado diretamente para os vértices. O gerenciador de *jobs* é responsável somente pelo controle de decisões e não será um gargalo para qualquer transferência de dados. Um servidor de nomes (NS) fornece o suporte ao descobrimento de computadores disponíveis. Um *daemon* (D) é responsável por criar os processos em favor do gerenciador de *jobs* em cada computador no cluster. A primeira vez que um vértice (V) é executado em um computador seus binários são enviados do gerenciador de *jobs* para o *daemon*. O *daemon* atua como um *proxy* para o gerenciador de *jobs* poder comunicar o estado da computação e a quantidade de dados lidos ou escritos nos canais.

Os grafos são construídos pela combinação de modelos simples de um conjunto de operações. Todas as operações preservam as suas propriedades e resultam em um grafo acíclico. O objeto básico do grafo é dado pela equação 1.2, G onde contém uma sequência de vértices V_G, um conjunto de arestas E_G, e dois direcionadores $I_G \subseteq V_G$ e $O_G \subseteq V_G$ que indicam se o vértice é de entrada ou saída respectivamente. Nenhum grafo pode conter uma aresta inserindo uma entrada em I_G ou saída em O_G, estas indicações são utilizadas somente para as composições de operações.

$$G = \langle V_G, E_G, I_G, O_G \rangle \tag{1.2}$$

Através dos direcionadores, a entrada e saída de arestas de um vértice são orientadas e uma aresta conecta a portas específicas em um par de vértices. Um par de vértices também pode estar conectado a múltiplas arestas. Cada canal de comunicação tem um protocolo associado. O canal é implementado usando um arquivo temporário: um produtor escreve no disco local (normalmente) e um consumidor lê deste disco o arquivo [ISA 2007].

Os protocolos de comunicação dos canais podem ser: arquivos (preservado até a conclusão da tarefa), TCP *pipe* (que não requer discos, mas deve ser escalonado fim-a-fim entre os vértices ao mesmo tempo) ou memória compartilhada FIFO (que tem baixo custo de comunicação, mas os vértices devem rodar no mesmo processo).

Os vértices do *Dryad* contém puramente código sequencial com suporte à programação baseada em eventos, usando um *pool* de *threads*. O programa e as interfaces dos canais têm forma assíncrona, embora seja mais simples utilizar interfaces assíncronas que escrever códigos sequenciais usando interfaces síncronas [ISA 2007].

1.2.3. SWIFT

O trabalho de [WIL 2009] apresenta um conceito diferente para o processamento em larga escala. A proposta consiste em explorar um modelo de programação paralela orientada ao fluxo de dados, que trata as aplicações como funções e o conjunto de dados como objetos estruturados. A técnica cria múltiplas cópias de um programa sequencial para executá-las ao mesmo tempo em um *script* dentro de um ambiente paralelo.

O *Swift* combina a sintaxe da linguagem de programação C com características de programação funcional. A linguagem é projetada para expor as oportunidades de execução paralela, evitar a introdução de falta de determinismo, simplificar o desenvolvimento de programas que operem em sistemas de arquivos e permitir uma execução eficiente em computadores paralelos de memória distribuída. A arquitetura do *Swift* é apresentada na Figura 1.2, adaptada de [WIL 2009]. A arquitetura de software é constituída de quatro camadas que são:

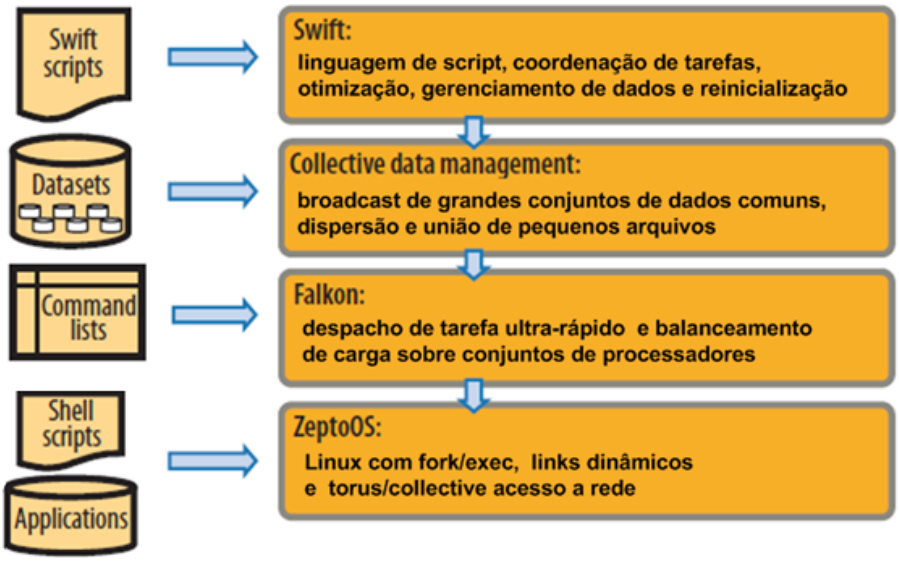

Figura 1.2: Arquitetura do *Swift*.

1 O *Swift*: é uma linguagem de *script* que coordena as tarefas, otimizações, gerenciamento de dados e reinicialização;

2 CDM (*Collective data management*): é responsável pela transmissão de grandes quantidades de dados de entradas comuns por *broadcast*, dispersão e união de pequenos arquivos;

3 *Falkon*: é um *dispatcher* de tarefas que usa a combinação de escalonamento *multi-thread* com uma arquitetura hierárquica, promovendo o balanceamento de carga;

4 *ZeptoOS*: é um sistema operacional Linux que fornece acesso para as primitivas *fork*() e *exec*() utilizadas no lançamento de novos programas.

Um programa produz os dados em um arquivo que, imediatamente, são consumidos por um segundo programa. O *Swift* assegura que para a variável compartilhada (representada pelo arquivo) não é atribuído nenhum valor até o primeiro programa completar sua execução. A quantidade de códigos necessários para expressar aplicações dessa forma é menor do que *scripts ad hoc* ou *shell scripts* e menos expressivos como anotações de grafos acíclicos orientados (dígrafos) da proposta do Dryad [WIL 2009].

1.2.4. CASSANDRA

O Cassandra é um sistema de armazenamento descentralizado, utilizado para gerenciar grandes quantidades de dados distribuídos entre diversos servidores. O Cassandra é um serviço de alta disponibilidade sem ponto único de falha, proposto pelo Facebook em [LAK 2010], para atingir os níveis de escalabilidade e confiabilidade que um serviço dessa escala requer. O sistema executa sobre uma infraestrutura de centenas de máquinas, possivelmente distribuídas em vários *data centers*. Assim, nesta escala, os componentes falham continuamente de maneira aleatória. A maneira com que o Cassandra gerencia o estado persistente frente a essas falhas influi na confiabilidade e escalabilidade dos sistemas de software que confiam nesse serviço.

Por diversos aspectos, o Cassandra parece com um banco de dados e compartilha estratégias de *design* e implementação de um banco de dados, mas não suporta um modelo totalmente relacional. Provê aos clientes um modelo simples que suporta o controle dinâmico sobre a estrutura e formato dos dados. O sistema Cassandra foi idealizado para gerenciar grandes taxas de transferência de dados e executar sobre um *hardware* de prateleira. O Cassandra tem demonstrado ser um sistema de armazenamento que provê escalabilidade, alto rendimento e uma grande aplicabilidade.

1.2.4.1. Modelo de Dados

O modelo de dados do Cassandra especifica que uma tabela seja um mapa de múltiplas dimensões que são indexadas por uma chave. Dentro da sua arquitetura,

o Cassandra possui módulos para: divisão, replicação, cadastro, gerenciamento de falhas e escalonamento. Estes módulos trabalham em sincronia para dar conta das requisições de leitura/escrita.

Para obter capacidade de escalonamento, os dados são divididos entre as máquinas do *cluster*. O Cassandra utiliza uma função de *hashing* consistente para identificar as máquinas dentro de um anel, cada uma das máquinas é assinalada com um valor aleatório que representa sua posição dentro do anel. Cada dado é identificado por uma chave e atribuído a uma máquina aplicando uma função de *hash* na chave para indicar a qual máquina pertence. A aplicação especifica sua chave e o Cassandra a usa para encaminhar as requisições. Assim, cada máquina é responsável somente pelos dados atribuídos a ela.

O *hashing* pode causar maior sobrecarga carga a uma máquina que outras. Para evitar isto, analisa-se a carga no anel e as máquinas com maior carga mudam de posição dentro do anel. Cassandra utiliza replicação para conseguir alta disponibilidade e durabilidade dos dados. Cada um dos dados é replicado em N máquinas (a argumento N é configurado como fator de replicação). Cada chave k é designada a uma máquina que fará a coordenação (aquele com o menor identificador). O Cassandra tem vários tipos de políticas de replicação de dados: *Rack Aware, Rack Unaware e Datacenter Aware*.

A detecção de falhas no Cassandra é um mecanismo no qual uma máquina pode determinar se outra máquina dentro do sistema está *up* ou *down*. As máquinas que estão *down* devem ser substituídas por aqueles que tem os dados replicados. No Cassandra utiliza-se a detecção de falhas para evitar tentativas de comunicação com máquina não acessíveis. Para isto, o Cassandra utiliza uma modificação do parâmetro ϕ, chamado de *Accrual Failure Detector*, para ajustar o nível de detecção de falhas da rede [LAK 2010].

Quando uma nova máquina é adicionada ao sistema, um *token* é lançado no anel com o objetivo de aliviar a carga das máquinas que formam o anel. Isto faz com que a nova máquina divida a informação com as outras, distribuindo a carga entre elas.

O sistema Cassandra depende do sistema de arquivos local para a persistência de dados. Cada um dos dados é representado no disco utilizando um formato que permite uma eficiente recuperação de dados. Uma operação típica de escrita envolve um registro para a recuperação e durabilidade dos dados, e as atualizações de dados em memória. Uma operação típica de leitura faz primeiro uma consulta na estrutura dos dados dentro da memória e, após, os arquivos são lidos na ordem do mais novo ao mais velho.

1.2.5. MapReduce

O *MapReduce* é um modelo de computação intensiva em dados proposto em 2004, por Jeffrey Dean e Sanjay Ghemawat, para simplificar a construção de índices reversos na análise de pesquisas na *web* [DEA 2004]. Os algoritmos utilizados no *MapReduce* permitem o tratamento de dados intensivos em um sistema de arquivos distribuído, a abstração do paralelismo de tarefas e o controle de falhas.

O funcionamento do *MapReduce*, em uma abstração em alto nível, pode ser comparado a uma linguagem de consulta SQL. A Figura 1.3 [CHE 2008], exemplifica uma aplicação *word count* para a contagem de palavras de uma determinada entrada de dados. Os dados da entrada são selecionados por uma tarefa *Map* que gera um conjunto de *tuplas* (chave_in,valor_in). As *tuplas* são então ordenadas para, na saída, serem agrupadas em *tuplas* (chave_out, valor_out) por uma tarefa *Reduce*.

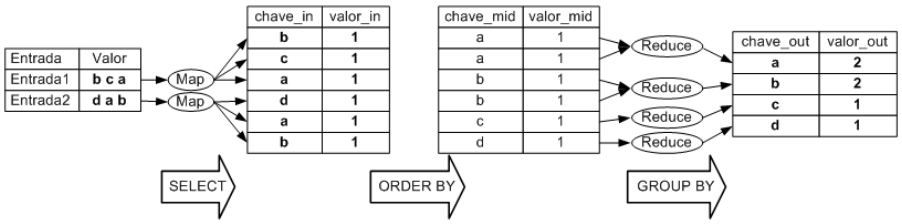

Figura 1.3: Estrutura do modelo de programação do *MapReduce*.

No exemplo, a contagem dos valores das ocorrências das chaves iguais é somada na tarefa de *Reduce*. Assim abstraindo-se as funções de *Map* e *Reduce*, pode-se comparar a execução do processamento como um fluxo de uma consulta na linguagem SQL [WHI 2009]. A representação da consulta ao banco de dados é exemplificada através das funções representadas nos comandos a seguir:

SELECT *chave_in* **FROM** *entrada* **ORDER BY** *chave_in* **CREC**

SELECT COUNT(*), *chave_mid* **AS** *chave_out* **FROM** *chave_mid* **GROUP BY** *chave_mid* **CREC**

Entretanto, diferentemente de uma consulta em banco de dados, o *MapReduce* executa esta busca nos dados em centenas ou milhares de máquinas simultaneamente, abstraindo o paralelismo das tarefas, através da declaração de duas funções, como apresentado no exemplo para a contagem de palavras no pseudocódigo 1 do (*Word Count*), adaptado de [DEA 2010].

O objetivo do *Word Count* é contar a quantidade de ocorrências de cada palavra em um documento. Cada chamada da função *Map* recebe como *valor* uma linha de texto do documento e como *chave* o número desta linha. Para cada palavra encontrada na linha recebida a função emite um par (chave,valor), onde a chave é

a palavra em si e o valor é a constante 1 (um). A função *Reduce*, então, recebe como entrada uma palavra (*chave*) e um iterador para todos os valores emitidos pela função *Map*, associados com a palavra em questão. No final é emitido um par (chave,valor), contendo em cada palavra o total de sua ocorrência [DEA 2010].

Pseudocódigo 1 *Word Count*

```
map (line_number, text):
word_list[] = split (text)
Para each word in word_list: Faça
   emit (word, 1)
Fim Para
reduce (word, values[]):
word_count = 0
Para each v in values: Faça
   word_count+ = v
Fim Para
emit (word, word_count)
```

A Figura 1.4 exemplifica o funcionamento do *MapReduce* para uma operação *WordCount* (contagem de palavras). As funções de *Map* e *Reduce* são aplicadas sobre os dados, *e.g.* 1 TB de dados. Os dados contêm as vendas de carros de uma concessionária. O objetivo é encontrar o número de carros existentes no estoque.

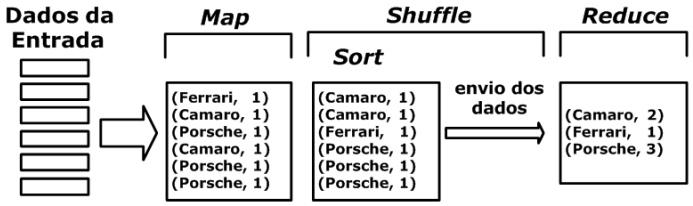

Figura 1.4: Exemplo do fluxo de dados do *Word Count*.

Após um *job* ter sido submetido por um programador, uma tarefa *Map* processa as linhas da entrada de dados e emite um valor 1 a cada nome de carro não vendido. Na fase de *Shuffle* os dados intermediários produzidos no *Map* são ordenados por um *sort* e após, as chaves intermediárias são enviadas para serem consumidas pelas tarefas *Reduce* na próxima fase. Finalmente, após as máquinas receberem todos os dados intermediários executa-se um *merge* para preparar os dados para a execução da tarefa *Reduce*. A tarefa *Reduce* soma os valores das chaves semelhantes e emite um resultado .

O *MapReduce* relaciona-se com um banco de dados pela lógica de consulta dos dados. Porém diferentemente de uma consulta SQL, o *MapReduce* tem uma

escalabilidade linear. Isto ocorre porque podem ser feitas diversas consultas simultaneamente distribuídas entre várias tarefas. Os dados podem não ter qualquer estrutura interna ou serem estruturados ou semi-estruturados. Arquivos longos de todos os tipos são bem aceitos para serem analisados neste contexto, sendo desmembrados em tamanhos menores e manipulados por diversas máquinas [WHI 2009].

1.2.5.1. Fluxo de Dados do *MapReduce*

O *MapReduce* abstrai a complexidade do paralelismo das aplicações. O conceito computacional simples da manipulação de dados, através de uma função de *tuplas* (chave,valor), esconde do programador a grande complexidade da distribuição e do gerenciamento de dados. A complexidade se deve aos dados serem de grandes volumes, por estarem espalhados através de centenas ou milhares de máquinas e pela necessidade da computação ser realizada com intervalos de tempos cada vez menores [DEA 2010].

A arquitetura do *MapReduce* é constituída de uma máquina mestre que gerencia as demais máquinas escravas. A estrutura do *MapReduce* é composta de diversos módulos como: um escalonador, mecanismos de *split* de dados da entrada, sistema de arquivos distribuídos, *sort*/*merge* de dados e sistemas de monitoramento e gerenciamento. O mecanismo de comunicação é baseado no envio de mensagens de *heartbeat* que informa o *status* das máquinas escrevas ao mestre.

A entrada de dados é dividida em pedaços menores chamados de *chunks*. O tamanho é um parâmetro fornecido pelo programador, normalmente de 64 MB, embora o valor dependa da tecnologia dos discos e da taxa de transferência de dados da memória para o disco [WHI 2009]. Os dados são divididos e após distribuídos em um sistema de arquivos que mantém um mecanismo de replicação dos dados.

A Figura 1.5, adaptada de [WHI 2009], apresenta o modelo do fluxo de dados do *MapReduce* com três fases distintas. As fases de *Map* e *Reduce* são acessíveis ao programador. A terceira, chamada de *Shuffle*, é criada pelo sistema durante sua execução. A tarefa *Map*, quando executa uma função de mapeamento, gera dados intermediários que são armazenados na máquina local. A tarefa *Reduce* recebe estes dados intermediários e executa uma função de redução que agrupa as chaves iguais. O *Shuffle* é constituído de dois processos: um na máquina que processa a tarefa *Map* onde executa-se um *sort* (ordenação de chaves) e serialização dos dados. O segundo, ocorre após os dados intermediários serem enviados para a máquina que executará a tarefa de *Reduce*, nesta máquina é aplicado um *merge* nos dados recebidos das tarefas *Map* com o objetivo de agrupar adequadamente as chaves intermediárias antes de se executar a função *Reduce* [WHI 2009].

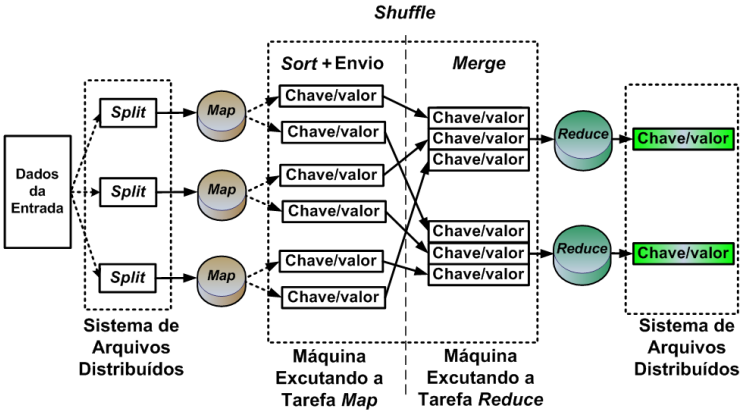

Figura 1.5: Fluxo de dados do *MapReduce*.

A implementação do *MapReduce* particiona cada tarefa em um conjunto de sub-tarefas menores de tamanhos iguais, com características *Bag-of-Tasks* (tarefas sem dependências). Um cliente submete um *job* com as definições das funções *Map* e *Reduce* que, então, será dividido em várias tarefas ou *tasks* (*Map* e *Reduce*) no mestre. O mestre designa as tarefas às máquinas que irão executar cada etapa do processamento. Os dados intermediários produzidos no *Map* são armazenados temporariamente no disco local. O modelo de execução cria uma barreira computacional, que permite sincronizar a execução de tarefas entre produtor e consumidor. Uma tarefa *Reduce* não inicia seu processamento enquanto todas as tarefas *Map* não terminarem primeiro.

Uma função *Hash* é aplicada sobre os dados intermediários produzidos no *Map*, para determinar quais das chaves irão compor as tarefas de *Reduce* a serem executadas. As chaves iguais de todas as tarefas *Map* são transferidas, no processo de *Shuffle*, para uma mesma máquina processar a tarefa de *Reduce*. Após uma função de redução ser aplicada nestes dados intermediários é emitido um novo resultado. O resultado em formato de *tuplas* é então armazenado no sistema de arquivos distribuído para ser disponibilizado ao cliente que submeteu o *job*.

1.2.6. SAGA

O SAGA (***S**imple **A**PI for **G**rid **A**pplications*) apresenta uma interface de programação de alto nível que oferece a capacidade de criar aplicações distribuídas em *grids* e *clouds* [MIC 2009].

O modelo da arquitetura do SAGA é representado na Figura 1.6, adaptado de [MIC 2009]. A implementação SAGA-*MapReduce* é relativamente mais complexa

e naturalmente mais lenta que o *MapReduce* original. A aplicação consiste de 2 processos o *master* e o *worker*.

O *master* é responsável por gerenciar os dados, controlar réplicas, manipular dados intermediários e lançar as tarefas para os *workers*. Os *workers* unicamente têm a função de executar as tarefas.

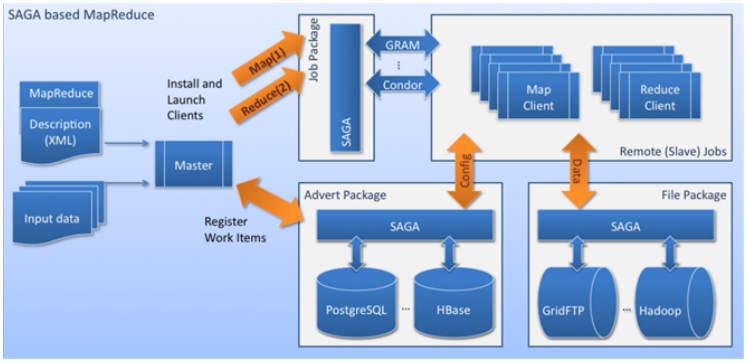

Figura 1.6: Arquitetura do SAGA usando *MapReduce*.

No SAGA a implementação cria uma interface para o usuário. O código lógico recebe instruções explícitas para onde devem ser escalonadas as tarefas. A aplicação não é obrigada a executar em um sistema que fornece a semântica originalmente exigida pelo *MapReduce*, assim a aplicação pode ser portável para uma ampla gama de sistemas genéricos.

O SAGA pode lançar *jobs* e utilizar tanto a infra-estrutura do Globus/GRAM como a do Condor para execução das tarefas. A abstração do *MapReduce* é feita sobre a implementação do All-Pairs. Os arquivos podem estar em qualquer tipo de sistema de aquivos distribuídos como por exemplo HDFS, Globus/GridFTP, KFS ou nos sistemas de arquivos locais das máquinas. O *master* cria partições de processos análogos aos *chunks* no *MapReduce*. A arquitetura é mestre/escravo e a comunicação é feita através do SAGA *advert package*. Uma das desvantagens do SAGA-*MapReduce* é não ser *multi-thread*.

1.3. Hadoop

O Hadoop é uma implantação *Open Source* do *MapReduce* desenvolvida pela Apache Software Foundation (http://hadoop.apache.org/), e é utilizada como base para o desenvolvimento de aplicações de *Cloud Computing* na indústria e para

a pesquisa de computação intensiva em dados nas universidades. O seu funcionamento é idêntico ao *MapReduce* e seus algoritmos são descritos nesta seção. Grandes empresas como, por exemplo, Yahoo, Facebook, Amazon e IBM utilizam o modelo do *MapReduce* como ferramenta para aplicações de *Cloud Computing*, através desta implementação.

O *MapReduce* é constituído de vários algoritmos, entre eles o algoritmo de distribuição de dados e de tarefas. O algoritmo de distribuição de dados executa a divisão e a distribuição dos dados sobre um sistema de arquivos distribuídos. O algoritmo de distribuição de tarefas, recebe o *job* e cria tarefas *Map* e *Reduce* para serem distribuídas às máquinas que executarão o processamento. Os algoritmos de distribuição de dados e de tarefas são a base para a computação paralela no *MapReduce*, que é implementado no Hadoop.

1.3.1. Descrição Geral dos Algoritmos

O algoritmo de distribuição de tarefas possui duas funções, um para distribuir as tarefas *Map* e outro para as tarefas *Reduce*. Tarefas *backup* são criadas através de réplicas destas tarefas, conforme um parâmetro definido pelo programador, e o escalonador cria as tarefas e as réplicas associadas. Porém, ele dispara uma tarefa de cada réplica por vez. As demais tarefas somente serão lançadas para outra máquina livre no caso de se constatar alguma falha (tarefa *backup*) ou para aumento de desempenho, através da análise do tempo de progresso de uma tarefa (tarefa especulativa).

Uma tarefa *Map* é associada a cada *chunk* no momento em que ocorre a divisão dos dados no sistema de arquivos distribuídos durante o processo de *split*. Então, as tarefas são colocadas no mestre em uma fila FIFO para serem processadas pelas máquinas. As máquinas escravas enviam uma mensagem de *heartbeat* solicitando uma tarefa ao mestre. A tarefa de *Map* é enviada para a máquina que contenha uma réplica do *chunk*. Caso o *chunk* não esteja na máquina local, o escalonador verifica qual máquina contém uma réplica do *chunk* e envia a tarefa a ela [DEA 2010].

Um algoritmo de distribuição de dados faz o *split* dos dados da entrada para criar os *chunks*. Um mecanismo de replicação faz uma cópia dos dados. Quando as tarefas forem executadas, os dados normalmente estarão na máquina local, portanto, sem perdas associadas com a transferência de dados, em tempo de execução. Neste tipo de aplicação, a política sobre os dados da entrada força a escrita do dado uma única vez, executando após diversas leituras e manipulações sobre os mesmos dados, política também conhecida como (*Write Once, Read Many Times*) [WHI 2009].

O lançamento de tarefas remotas ocorre quando uma máquina encontra-se livre mas não tem uma réplica dos dados para processar no disco local. Assim a

máquina faz uma cópia dos dados em tempo de execução de uma outra máquina que contenha a réplica do *chunk*. O lançamento de tarefas especulativas ocorre após não existirem tarefas pendentes para serem lançadas. A tarefa especulativa é uma tarefa criada a partir da constatação de uma possível falha, originada da análise do tempo de progresso de uma tarefa [DEA 2010].

Um processo denominado $Job\_inProgress$ controla o progresso de execução das tarefas. Se uma máquina está executando uma tarefa por mais de 60 segundos e seu progresso é igual ou inferior à 20% do tempo médio total das execuções das tarefas de cada fase, a máquina é marcada como *straggler* (máquina lenta). Assim, as máquinas são classificadas de acordo com seu progresso de execução em máquinas normais, máquinas que falham e máquinas *stragglers*.

Após não haverem mais tarefas pendentes, o escalonador lança uma tarefa especulativa para uma máquina livre que contenha uma réplica dos dados, com o objetivo de aumentar o desempenho das execuções das máquinas lentas. Quando qualquer uma das máquinas (a *straggler* ou aquela com a tarefa especulativa), enviar uma mensagem de *heartbeat* informando a conclusão da tarefa, a outra tarefa será encerrada [WHI 2009, DEA 2010].

Na fase de *Map* se uma máquina tornar-se livre, as tarefas são escolhidas na seguinte ordem: tarefa que falhou, tarefa pendente com *chunk* local, tarefa pendente com *chunk* que está em outra máquina e, finalmente, tarefa especulativa. Uma tarefa lançada para uma máquina livre que não possua os dados locais é denominada de tarefa remota. Em uma tarefa remota, a máquina necessitará copiar os dados de outra máquina antes de executar essa tarefa [WHI 2009].

Na fase de *Reduce* as tarefas são sinalizadas às máquinas através de uma função *Hash* como no *MapReduce*. A função de *Hash* divide as chaves intermediárias pelo número de *Reduces* definido pelo programador. As máquinas que irão processar as tarefas de *Reduce* iniciam a cópia dos dados intermediários quando 5% das tarefas *Map* já estiverem concluídas, num processo denominado de *prefetch*. Porém, o processamento das tarefas *Reduces* somente iniciará após todas as tarefas *Map* estarem concluídas. Na fase de *Reduce*, quando uma máquina tornar-se livre, as tarefas são escolhidas na seguinte ordem: tarefa que falhou, tarefa pendente e tarefa especulativa. [WHI 2009]

1.3.2. Algoritmo para a Divisão de Dados

O algoritmo 2 descreve o processo de divisão dos dados (*data split*), executado pelo HDFS quando é lançado um *job* por um usuário. A entrada de dados é dividida em tamanhos iguais (Tam_chunk) que podem ser de 16, 32, 64 ou 128 MB (o padrão é 64 MB). Cada bloco de dados recebe o nome de *chunk*.

Algoritmo 2 HDFS: *Split* dos dados

1: $Dados \leftarrow$ entrada de dados no HDFS
2: **Se** $(Dados\ \%\ Tam\_chunk) \neq 0$ **then**
3: $\quad Num\_chunks \leftarrow (\frac{dados}{Tam\_chunk}) + 1$
4: **Então**
5: $\quad Num\_chunks \leftarrow \frac{dados}{Tam\_chunk}$
6: **Fim Se**
7: **Para** $j \leftarrow 1$ to Num_hosts **Faça**
8: $\quad HD(j) \leftarrow$ verifica tamHD do $Host(j)$
9: **Fim Para**
10: **Para** $i \leftarrow 1$ to Num_chunks **Faça**
11: $\quad Chunk(i) \leftarrow Dados[i]$
12: $\quad$ **Se** $HD(j) \geqslant Tam\_chunk$ **then**
13: $\quad\quad Host(j) \leftarrow Chunk(i)$
14: $\quad\quad HD(j) \leftarrow HD(j) - Tam\_chunk$
15: $\quad$ **Então**
16: $\quad\quad j \leftarrow (j\ \%\ Num\_hosts) + 1$ { Vai para a próxima iteração }
17: $\quad\quad i \leftarrow i - 1$
18: $\quad$ **Fim Se**
19: $\quad$ **Para** $r \leftarrow 1$ to $Num\_replicas$ **Faça**
20: $\quad\quad Host(j + r) \leftarrow Chunk(i)$
21: $\quad\quad j \leftarrow (j\%Num\_hosts) + 1$
22: $\quad$ **Fim Para**
23: **Fim Para**

Uma vez definido o início e o fim de cada *chunk*, os dados são preparados e enviados às máquinas. No processo de envio, o HDFS testa o espaço disponível no *storage* de cada máquina e distribui os dados de forma balanceada.

Um sistema de tolerância a falhas replica os *chunks* para evitar a redistribuição de dados em tempo de execução, caso ocorra a queda de uma máquina. O número de réplicas ($Num\_replicas$) é um parâmetro definido pelo programador (o padrão é 3). As réplicas são transferidas para as máquinas considerando também o balanceamento de carga [WHI 2009].

1.3.3. Algoritmos do *TaskRunner* e *JobTracker*

Cada uma das tarefas *Map* tem um *buffer* de memória circular associado de 100 MB para escrever a saída de dados. Quando o *buffer* atinge 80% de sua capacidade inicia-se uma *thread* para executar um processo *spill* que descarrega seu conteúdo para o disco. As saídas do *Map* irão continuar a ser escritas no *buffer* enquanto o processo *spill* ocorrer. O processo segue um algoritmo *round-robin*, escrevendo para o *buffer* de memória e do *buffer* para o disco de forma circular até a conclusão da tarefa [WHI 2009].

Uma máquina recebe uma ou mais tarefas conforme sua quantidade de *slots* livres. Um *slot* define quantas tarefas podem ser processadas simultaneamente. O número é um parâmetro de configuração global definido pelo programador. As informações do progresso de uma tarefa são enviadas para o mestre através de um *heartbeat*. O progresso das tarefas é monitorado através de um processo chamado *TaskRunner* [WHI 2009].

A cada tarefa são associados uma prioridade e um tempo estimado para o término de sua execução, conforme é apresentado no algoritmo 3 para o *Map* e no algoritmo 4 para o *Reduce*. Caso o tempo de progresso da tarefa seja maior que o tempo médio das execuções das tarefas no *cluster* ($Average\_taskMap$) a máquina será marcada como *straggler*.

Algoritmo 3 Hadoop: *TaskRunner-Map*

1: $Receive() = Heartbeat[StatusHost()]$
2: **Se** Job não terminou **then**
3: $New\_taskId() \leftarrow f(Map[Chunk_t])$
4: **Se** $Time\_taskExec > 60s$ e $Job\_inProgress \leqslant 0,2 * Average\_taskMap$ **then**
5: $Host(j) \leftarrow Straggler$
6: Troca prioridade da tarefa
7: Lança tarefa especulativa para o $Host_{j+1}$ que tenha $Chunk_t$ {Tarefa especulativa}
8: **Fim Se**
9: **Fim Se**
10: Aguarda tarefa completar

Ao término das tarefas pendentes uma nova tarefa *backup* é lançada pelo *JobTracker* para uma máquina que contenha uma réplica dos dados. O processo de lançamento de tarefas *backup* é chamado de execução especulativa. O *JobTracker* fica aguardando a conclusão da tarefa e a máquina que mandar primeiro uma mensagem informando da conclusão da tarefa (tarefa normal ou especulativa) força o cancelamento da outra.

No lançamento de tarefas *Reduce*, como no algoritmo 4, as chaves intermediárias residem na máquina que executa a tarefa de *Map*. Os dados são copiados das máquinas que executaram as *Map* para aquelas que executarão as tarefas *Reduce*, durante um processo de cópia na fase de *Shuffle*. Quando uma tarefa especulativa é lançada as chaves intermediárias $Map_n(key[n], value[n])$ devem ser copiadas antes da execução da tarefa.

Algoritmo 4 Hadoop: *TaskRunner-Reduce*

1: $Receive() = heartbeat[StatusHost()]$
2: **Se** Job não terminou **then**
3: $New\_taskId() \leftarrow f(Reduce(Map(key[n], value[n]))$
4: **Se** $Time\_taskExec > 60s$ e $job\_inProgress \leqslant 0,2 * Average\_taskReduce$ **then**
5: $Host(j) \leftarrow straggler$
6: Troca prioridade da tarefa
7: $New\_taskId_{esp}() \leftarrow f(Reduce(Map(key[n], value[n]))$ {Cria tarefa especulativa}
8: $Send(taskReduce, Host_{j+1})$
9: $Send(locais(P_{interm}(key[n], value[n]), Host_{j+1})$ { $Host_{j+1}$ Copia pares}
10: **Fim Se**
11: **Fim Se**
12: Aguarda tarefa completar

O lançamento de tarefas executado no mestre é apresentado no algoritmo 5. O processo *JobTracker*, executado no mestre, faz o controle e o escalonamento das tarefas do *MapReduce*. Quando o usuário submeter um *job*, o *JobTracker* verifica o número de máquinas participantes do *cluster* e cria um *Job_Id* para cada entrada de dados do usuário. Um processo de *split* divide os dados no HDFS, define as funções *Map* associadas a cada *chunk* e a localização dos dados.

Algoritmo 5 Hadoop: *JobTracker*

1: **Para** $i \leftarrow 1$ to Num_chunks **Faça**
2: $Submit\_task(x) \rightarrow Create\ New\_taskMap(i)$
3: **Enquanto** $t < Num\_hosts$ **Faça**
4: set $Job\_priority(x) \rightarrow taskMap(i)$
5: $Send(taskMap, Host_j)$
6: **Fim Enquanto**
7: **Fim Para**
8: Executa Hadoop: *TaskRunner-Map*
9: **Se** $Job\_inProgressTask[Map(finish)] > 5\%$ **then**
10: **Para** $j \leftarrow 1$ to $Num\_reduces$ **Faça**
11: Divide chaves intermediárias com uma função de *Hash*
12: $Submit\_task(x) \rightarrow Create\ New\_taskReduce(i)$
13: set $Job\_priority(x) \rightarrow taskReduce(i)$
14: $Send(taskReduce, Host_j)$
15: $Send(locais(P_{interm}(key[n], value[n]), Host_j)$ { $Host_j$ Copia pares}
16: **Fim Para**
17: **Fim Se**
18: Executa Hadoop: *TaskRunner-Reduce*

O *JobTracker*, após receber uma chamada para criar uma nova tarefa de *Map* ou *Reduce*, cria a respectiva tarefa e coloca-a em uma fila FIFO. As propriedades das tarefas são definidas nas configurações do *job* que recebem o nome de *JobConf*.

As propriedades do *JobConf* são copiadas para a tarefa e enviadas junto com um número de identificação dessa tarefa para a máquina processar.

As tarefas recebem uma prioridade que pode ser definida como: muito baixa, baixa, normal, alta e muito alta. Por padrão as tarefas são configuradas com prioridade normal e são então lançadas para execução. A prioridade das tarefas muda no $Job\_inProgress$ conforme o progresso da tarefa.

Uma função de *Hash* é aplicada sobre os pares intermediários da fase de *Map* para dividir os dados de forma balanceada para as máquinas. As tarefas *Reduce* são criadas conforme o número de *Reduces* ($Num\_reduces$), definido pelo programador. Quando uma máquina é notificada pelo mestre da localização dos pares intermediários, ela lê os dados intermediários do disco local de cada máquina. Após ler os dados intermediários, aplica-se um *sort* sobre as chaves intermediárias na fase de *Shuffle* para, então, serem agrupadas conforme sua ocorrência. O *sort* é necessário porque tipicamente muitas chaves intermediárias diferentes são produzidas pelo *Map* para uma mesma tarefa *Reduce* [DEA 2010].

1.4. *MapReduce* em Outras Plataformas

O *framework* do *MapReduce* pode ser implementado em outras plataformas diferentemente de um *cluster* homogêneo. Entretanto, poderão existir algumas restrições ou características impostas pela própria estrutura da plataforma utilizada. Nesta seção serão apresentadas as demais plataformas em que o *MapReduce* pode ser implementado.

1.4.1. Plataformas *Desktop Grid*

A computação em *grid* utiliza diversos computadores distribuídos através de várias organizações virtuais. Os recursos computacionais são heterogêneos e aplicam troca de mensagens para a computação de alto desempenho. A abordagem funciona bem para a computação intensiva no domínio de uma rede local, porém pode tornar-se um problema se for necessário o acesso a grandes volumes de dados através das demais organizações.

As tarefas de *Map* ou *Reduce* são do tipo *Bag-of-tasks*, ou seja, sem dependências entre si. Tarefas independentes facilitam o processamento das informações em *grid*. Assim os programadores não precisam se preocupar com a ordem da execução das tarefas. Eventuais falhas podem ser recuperadas facilmente com o escalonamento de uma nova tarefa para as máquinas que estejam livres.

Por mais de uma década, plataformas de Computação Voluntária têm sido

um dos maiores e mais poderosos sistemas de computação distribuída do planeta, oferecendo um alto retorno sobre o investimento para aplicações de uma ampla variedade de domínios científicos (incluindo a biologia computacional, a previsão do clima e física das altas energias). As *grids* de computação voluntária, como *desktop grids*, são formadas por máquinas *desktops* que utilizam ciclos ociosos de máquina para o processamento de tarefas [AND 2007]. Em *grids* formadas por máquinas dispersas em vários locais a heterogeneidade é muito maior do que aquelas formadas por organizações virtuais.

Várias implementações *desktop grid* utilizam recursos dispersos como, por exemplo, o BOINC (*Berkeley Open Infrastructure for Network Computing*) que é um *middleware* para o uso de recursos computacionais de forma voluntária. Dentre os seus diversos projetos, o SETI@home funciona com cerca de 3 milhões de computadores, com capacidade de processamento de mais de 109,5 Zettaflops . As *desktop grids* caracterizam-se por serem largamente distribuídas, heterogêneas e de alta volatilidade. Entretanto, a arquitetura de *desktop grid* não é voltada para aplicações intensivas em dados. O modelo do BOINC, por exemplo, é puramente mestre-escravo. No *MapReduce* o gerenciamento é mestre-escravo mas os nós utilizam comunicação direta para a execução das tarefas.

1.4.2. Memória Compartilhada

O *MapReduce* foi proposto para uso em ambientes de memória distribuída, como *clusters*. Porém, rapidamente foram apresentadas novas implementações do *framework* sobre memória compartilhada. O objetivo é de melhorar o desempenho do *MapReduce* em pequenas implementações. Ainda, mais recentemente, foi apresentado um modelo híbrido que utiliza ambas implementações de memória compartilhada e distribuída.

1.4.2.1. MARS

O MARS é uma implementação do *MapReduce* sobre GPUs, para fornecer aos desenvolvedores um *framework* sobre esta plataforma [FAN 2011]. A implementação pode ser utilizada para máquina de memória compartilhada ou de memória distribuída.

O projeto é dividido em três estágios: *Map*, Grupo e *Reduce*. O estágio Grupo é utilizado para produzir os dados intermediários na saída do *Map* e para calcular o tamanho dos dados intermediários. O objetivo do cálculo é determinar a área de *cache* necessária para criar cada *thread* na GPU. O fluxo da execução do MARS é apresentado na Figura 1.7, adaptada de [FAN 2011].

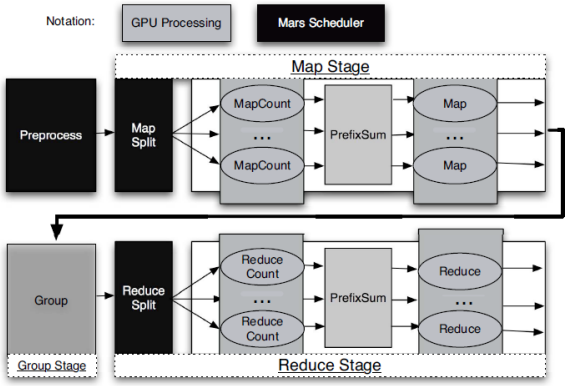

Figura 1.7: Fluxo de execução do MARS .

As fases de *Map Count, Reduce Count* e *PrefixSum* são responsáveis por calcular o tamanho dos dados para alocar a memória necessária dentro das GPUs em função de que nesta estrutura a alocação dinâmica de memória não é permitida.

O estágio de *Map* faz o *split* despachando a entrada gravada para *threads* nas GPUs. Cada *thread* executa uma função *Map* definida pelo usuário. Na fase de Grupo, um *short* e um *hash* são executados para preparar a próxima fase. O estágio *Reduce* separa os dados do Grupo e os envia às GPUs para executar a função *Reduce* e obter o resultado em um *buffer* único.

A estrutura dos dados no MARS afeta o fluxo de dados, o acesso à memória e a concepção do sistema. O MARS processa em CPU a entrada dos discos, transformando a entrada em pares chave/valor na memória principal, para ser transferida para a GPU. A implementação em memória distribuída utiliza o Hadoop para distribuir o processamento em CPU/GPU para outras máquinas. Nesta implementação o sistema de arquivos distribuído, a divisão dos dados, o escalonamento de tarefas e os mecanismos de tolerância a falhas são funções executadas pelo Hadoop.

1.4.2.2. *Phoenix*

O trabalho de [RAN 2007] apresenta uma implementação do *MapReduce* em sistemas de memória compartilhada, incluindo uma API de programação e um sistema de *runtime*. O *Phoenix* gerencia automaticamente a criação de *threads*, o escalonamento dinâmico de tarefas, a partição de dados e o processo de tolerância a falhas.

A implementação do *Phoenix* é baseada nos mesmos princípios do *MapReduce*. Porém, objetivando o uso de sistemas em memória compartilhada com processadores *multi-core* e simétricos. Utiliza *threads* para criar tarefas paralelas de

Map e *Reduce*, com *buffers* em memória compartilhada para facilitar a comunicação sem a cópia excessiva de dados.

O *runtime* escalona tarefas automaticamente através dos processadores disponíveis com o objetivo de proporcionar balanceamento de carga e maximizar o *throughput* de tarefas. Assim, o *runtime* gerencia a granularidade e a sinalização de tarefas paralelas, e recupera-se de transientes ou falhas durante a execução [RAN 2007].

Os fatores alocados pelo *runtime* são o tamanho das unidades, o número de máquinas envolvidas, as unidades que são atribuídas às máquinas dinamicamente e o espaço dos *buffers*. A decisão de qual fator alocar pode ser totalmente automática ou guiada pelo programador da aplicação.

As decisões tomadas permitem a execução eficiente da programação ao longo de uma grande variedade de máquinas e cenários sem modificações no código fonte. A API do *Phoenix* provém dois conjuntos de funções, um para inicializar o sistema e emitir pares de saídas e outro para definições do programador.

A Figura 1.8, adaptada de [RAN 2007], apresenta a estrutura do fluxo de dados do *runtime*. O *runtime* é controlado pelo escalonador, o qual é inicializado pelo código do usuário. O *worker* executa as tarefas de *Map* e *Reduce*.

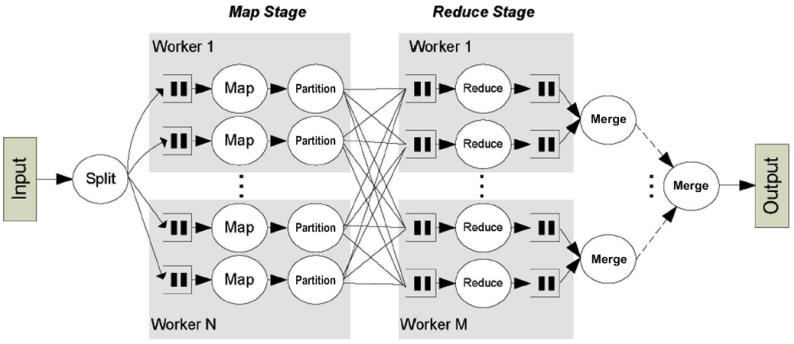

Figura 1.8: Controle do fluxo de dados do *runtime* no *Phoenix*.

O dados intermediários produzidos no *Map* são divididos em partições, chamadas de partição de dados intermediários. As funções garantem que dentro de uma partição de dados intermediários, os pares serão processados na ordem da chave. Isto permite a produção de uma saída ordenada porém, não existem garantias da ordem de processamento da entrada do *Map*.

Após a inicialização o escalonador determina o número de núcleos para uso de sua computação e cria uma *thread worker* em cada núcleo. Ao iniciar o estágio de *Map* os dados são divididos através do *Splitter* em unidades de dados de tamanhos iguais para serem processados.

Então, o *Splitter* é chamado novamente pela tarefa *Map* e retorna um ponteiro para os dados da tarefa a ser processada. A tarefa *Map* aloca o número de *workers* necessários para a execução da função *Map*, então cada *worker* processa os dados e emite um par de chaves intermediárias.

A função *Partition* separa os pares intermediários em unidades para a execução das tarefas *Reduce*. A função assegura que todos os valores da mesma chave estarão na mesma unidade. Dentro de cada *buffer* os valores são ordenados por chave através de um *sort*. O escalonador aguarda todas as tarefas *Map* encerrarem para depois lançar as tarefas *Reduce*.

A tarefa *Reduce* aloca os *workers* necessários para processarem as tarefas de maneira dinâmica. Uma mesma chave é agrupada em uma única tarefa para ser repassada pelo *worker Reduce*. Nesta fase pode-se ter grande desbalanceamento de carga. Assim, o escalonamento dinâmico é muito importante para evitar este problema.

Quando as saídas *Reduce* estão ordenadas por chave é executado um *merge* de todas as tarefas em um único *buffer*. O *merge* gasta $\log_2(\frac{numWorkers}{2})$ passos para completar, onde $numWorkers$ é o número de *workers* usados [RAN 2007].

1.4.3. *Cloud Computing*

Cloud Computing é atualmente uma maneira de facilitar a implementação de aplicações de alto desempenho como por exemplo o *MapReduce*. Como se falou nas seções anteriores, o *MapReduce* pode ser implementado em plataformas heterogêneas, ao custo de poder apresentar baixo desempenho. *Cloud Computing* introduz uma camada adicional que isola a heterogeneidade do hardware e permite uma implementação de um *cluster* homogêneo de máquinas virtuais. Uma *grid* pode utilizar máquinas heterogêneas e rodar uma infraestrutura de *Cloud*. Dentro desta *Cloud*, pode-se instanciar um *cluster* de máquinas virtuais que seria similar a uma infraestrutura homogênea para *MapReduce*.

O NIST [MEL 2011] define três modelos de serviço para *Cloud Computing* que dependem da capacidade e do controle oferecidos para o usuário. Uma implementação de *MapReduce* em *Cloud Computing* oferece em várias opções relacionadas com os modelos de serviço definidos pelo NIST. Executar o *MapReduce* em *Cloud Computing* se diferencia pela maneira de se incorporar o software do *MapReduce* dentro da nuvem e por um grau de controle que o usuário deve ter.

Em todos estes modelos, o usuário deve fornecer à implementação os dados que vão ser analisados. A transferência destes dados pode incorrer em custos para o usuário. O NIST também define quatro modelos de implementação para *Cloud Computing*, estes modelos se definem por suas características, o que também modifica a maneira em que os usuários interagem com o *MapReduce*.

Em todos os casos, uma implementação do *MapReduce* em nuvem privada requer que a organização possua um *data center* onde executa um *middleware* de *Cloud*. Esta implementação pode ser mais custosa para a organização, mas oferece maior controle para o administrador da infraestrutura. Utilizar uma nuvem pública significa utilizar um serviço a partir de um fornecedor, isto significa, que o usuário terá um menor controle da aplicação, embora não seja necessário uma infraestrutura física para implementar o serviço.

1.4.3.1. Infraestrutura como Serviço IaaS

Uma das maneiras mais fáceis de implementar um serviço na nuvem é a Infraestrutura como um serviço, o IaaS (*Infrastructure as a Service*). A implementação de *MapReduce* em IaaS segue a lógica de uma implementação em hardware dedicado físico: Instanciar um grupo de máquinas virtuais, instalar o software desejado e configurar como se fosse uma implementação em hardware.

As vantagens principais deste tipo de implementação são: o controle do usuário sobre o software instalado, a maneira de configurar o *cluster* e o número de máquinas virtuais que podem ser alocadas. A desvantagem é que o usuário será encarregado da instalação do software, Assim, é necessário um grande conhecimento sobre uma implementação do *MapReduce* por parte do usuário.

Uma implementação de *MapReduce* em IaaS pode ser realizada em qualquer dos fornecedores de *Cloud* de infraestrutura, como: Amazon Elastic Compute Cloud, Windows Azure, Google Compute Engine, ou pode ser realizada dentro de uma nuvem privada com o *middlewares* adequados *e.g.* Eucalyptus.

1.4.3.2. Software como Serviço SaaS

O modelo de Software como serviço na nuvem implica na disponibilidade de um *software* para o usuário diretamente a partir de um *browser*, sem ter que programá-lo. As vantagens principais desde tipo de implementação são: O menor tempo de implementação para o usuário, isto significa dizer que o usuário pode utilizar diretamente o software, e o usuário não precisa gerenciar a infraestrutura sobre a qual roda o serviço, o único controle que o usuário possui é da configuração de sua aplicação. No contexto de *MapReduce* como SaaS em uma nuvem privada, o usuário do *cluster* não tem acesso à infraestrutura, mas a implementação não tem diferença com a de uma implementação em IaaS.

Existem opções para utilizar *MapReduce* como serviço diretamente desde um provedor. O Windows Azure, da Microsoft, com seu produto Hadoop on Azure é uma implementação disponível atualmente para os usuários que inclui software e

dados de exemplo. Outras opções em nuvem pública são Amazon Elastic MapReduce, IBM InfoSphere BioInsights, e Cloudera CDH.

1.4.3.3. *MapReduce* em Plataforma como Serviço

O modelo da plataforma como serviço PaaS se refere à opção do usuário desenvolver um software que será executado na nuvem. O provedor da plataforma como serviço deve fornecer um *framework* para que o usuário desenvolva sua aplicação. Qualquer fornecedor de serviços PaaS pode-se converter em um fornecedor do *framework MapReduce* para uma implementação na nuvem. A principal vantagem desde tipo de implementação é que o usuário pode desenvolver sua aplicação com as características desejadas, sem depender de outros desenvolvedores. Porém, a desvantagem é que o usuário pode não ter controle sobre as características e até sobre o número de *workers* utilizados para a execução do seu programa.

Para um usuário de serviço PaaS, uma implementação de *MapReduce* implica em programar a aplicação utilizando a plataforma oferecida pelo fornecedor para depois converter seu programa em um serviço de SaaS, o que significa que o modelo de PaaS estritamente se reduz à programação do aplicativo *MapReduce* utilizando um serviço na nuvem. A utilização e a escolha dos serviços na nuvem são de responsabilidade dos usuários que devem considerar qual modelo se ajusta mais com a natureza da sua organização e dos seus dados.

1.5. *MapReduce* em Simuladores

As implementações de ambientes em larga escala com milhares de máquinas são muito complexas. Os poucos recursos disponibilizados para simulações científicas, como a *Grid 5000*, possuem ambientes de *clusters* homogêneos onde a criação de ambientes heterogêneos para a pesquisa em larga escala é muito difícil. Assim, o estudo dos simuladores existentes para o *MapReduce* justifica-se para identificar qual ferramenta possa ser utilizada para o estudo de modificações dos algoritmos sobre diferentes ambientes.

1.5.1. Simuladores Existentes para MapReduce

O estudo de algoritmos de escalonamento e distribuição de tarefas em ambientes intensivos em dados, como o *MapReduce*, exige uma grande quantidade de máquinas para ter-se um tratamento adequado de grandes volumes de dados em larga escala. A construção de experimentos com centenas ou até milhares de máquinas em computação voluntária, como em *desktop grids*, não é uma tarefa trivial.

Outro fator é não ter-se esta quantidade de equipamentos disponível para pesquisa. Portanto, a utilização de um simulador se faz necessária para este estudo.

Cardona *et al.* apresentaram um simulador para *grids* baseado nos simuladores *GridSim* e *SimJava* com foco na simulação de funções *Map* e *Reduce* em um sistema de arquivos semelhante ao HDFS [CAR 2007]. O simulador, entretanto tem muitas simplificações sobre o ambiente do *MapReduce* como, por exemplo, não considerar replicações, execuções especulativas e *split* de dados.

Ao nosso conhecimento, existem atualmente ao menos outros 7 simuladores MapReduce: *HSim, MRPerf, MRSG, MRSim, Mumak, SimMapReduce* e *SimMR*. Sendo que apenas o *MRPerf*, MRSG, *MRSim* e *Mumak* estão disponíveis publicamente e com código fonte aberto.

O *Mumak* [TAN 2009] simula o ambiente *MapReduce* a partir de um log de execução Hadoop, com o intuito de permitir análise de diferentes estratégias de escalonamento. Entretanto sua grande limitação é justamente não permitir a simulação de um *job* nunca executado. Ele foi desenvolvido para simular o ambiente de aplicações do *MapReduce*. O objetivo é ser uma ferramenta para planejar a implantação de ambientes de larga escala. O simulador é aplicado somente a ambiente homogêneo, não desempenha simulações de entrada/saída e computação, assim como, não detecta operações de comunicação como *heartbeat*. Assim não é possível simular funções como execuções especulativas.

Wang [WAN 2009] propôs o uso de um simulador chamado *MRPerf*, como um meio de planejar a configuração de parâmetros de otimização para a implantação do *MapReduce* em ambientes de larga escala. Este simulador utiliza o *ns-2* para a simulação da rede e o *DiskSim* para simular os discos. A implementação atual limita-se à modelagem de um único dispositivo de armazenamento por máquina, com suporte a apenas uma réplica para cada bloco de dados de saída. Destaca-se ao modelar aspectos do Hadoop como *spill* e pela modelagem direta dos custos da fase de *suffle*, além de permitir simulações básicas de falhas; mas não aplica algumas das técnicas de otimização do *MapReduce*, como a execução especulativa, além de não permitir a simulação de variações no poder de processamento ou na vazão dos enlaces.

O simulador *MRSim* foi desenvolvido baseado no *GridSim* e *SimJava*, e simula aplicações do *MapReduce*. As características do simulador incluem o *split* de dados e a replicação de dados no nível do *rack* local [HAM 2010]. Entretanto ambientes de execução real tem centenas ou milhares de máquinas distribuídas sobre diversos *racks*. Neste caso, os dados são replicados em diferentes *racks*. Porém, nenhuma interface é fornecida para modificar os algoritmos como, por exemplo, o escalonamento de tarefas e a distribuição de dados. Os autores advocam maior precisão que o MRPerf em função da detalhada especificação e modelagem dos custos para processamento de registros, de *spill* e de *merge*; entretanto valida-se apenas

uma aplicação, e esse grande número de parâmetros influencia essencialmente o desempenho individual dos workers não sendo útil para o estudo, por exemplo, do escalonamento.

1.5.2. MRSG

O simulador MRSG foi construído sobre o SimGrid para simular o ambiente de execução do *MapReduce*. O SimGrid foi escolhido pela sua escalabilidade e funcionalidades para simular diferentes tipos de aplicações distribuídas [DON 2010]. O objetivo do MRSG é de facilitar a pesquisa do comportamento do *MapReduce* em diferentes plataformas, possibilitando o estudo de novos modelos de algoritmos de modo simplificado e a construção de plataformas em larga escala, sem a necessidade de uma implementação de uma infra-estrutura de *hardware* necessária.

A Figura 1.9, ilustra os módulos necessários para a simulação do *MapReduce*. A plataforma de *hardware* e o comportamento das máquinas são simulados pelo SimGrid. O módulo DFS simula o sistema de arquivos distribuído, onde foram criados os novos algoritmos de distribuição dos dados e o novo mecanismo de agrupamentos. O projeto foi desenvolvido pelo subgrupo de pesquisas do MapReduce do GPPD da UFRGS, desde 2009, denominado GPPD/MR e encontra-se disponível para download no site https://github.com/MRSG/MRSG.

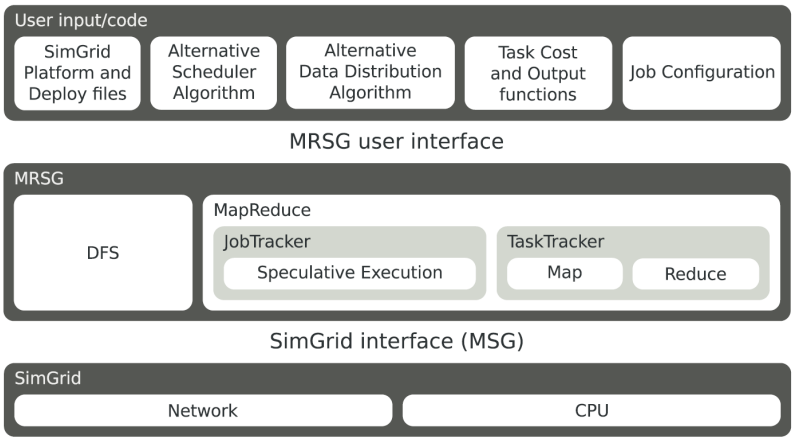

Figura 1.9: Módulos do MRSG.

As informações sobre a rede e as definições de *hardware* são feitas sobre arquivos de plataforma especificados pelo SimGrid. A submissão do *job* é definida por um arquivo de configuração do MRSG onde se especifica a quantidade de tarefas, o custo de cada tarefa, o número de *slots* disponíveis e a entrada de dados.

O gerenciamento das execuções do *MapReduce* e o escalonamento de tarefas são feitos no módulos do MRSG. A computação das máquinas e a simulação da *grid* são feitas pelo Simgrid.

Uma função *data skew* permite simular a distribuição dos dados conforme uma função matemática. Esta função pode expressar o modelo desejado de distribuição dos dados, assim facilitando o estudo de novos modelos e o comportamento de modelos de dados. Dois arquivos, escritos em formato XML, definem a arquitetura de *clusters* ou *grid* e das máquinas. Um arquivo de plataforma determina as características das máquinas e da rede, como é apresentado no *script* 1. Um segundo arquivo de *deployment* descreve a função de cada uma das máquinas na *grid*.

```
<!DOCTYPE platform SYSTEM "http://simgrid.gforge.inria.fr/simgrid.dtd">
<platform version="3">
  <AS id="AS0" routing="Full">
        <host id="Host 0" power="3500000000.0" core="2" />
        <host id="Host 1" power="3500000000.0" core="2" />
        ......
        <link id="l1" bandwidth="125000000.0" latency="1e-4" />
        <link id="l2" bandwidth="125000000.0" latency="1e-4" />
        .......
<route src="Host 0" dst="Host 1">
              <link_ctn id="l1"/>
        </route>
        ......
        <route src="Host 1" dst="Host 2">
               <link_ctn id="l1"/>
               <link_ctn id="l2"/>
             </route>
         .......
```

Outro arquivo necessário é o que descreve cada uma das máquinas na *grid*, chamado de arquivo de *deployment* apresentada no *script* 2.

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "http://simgrid.gforge.inria.fr/simgrid.dtd">
<platform version="3">
        <process host="Host 0" function="master"/>
        <process host="Host 1" function="worker"/>
        ..........
</platform>
```

Outro aquivo utilizado pelo MRSG é o que define as especificações do *job*, chamado de *mrsg.conf*. As especificações são o total de tarefas *Reduce*, o tamanho do *chunk*, o tamanho da entrada de dados, o número de réplicas, o percentual de saída de dados da função *Map*, o custo de cada tarefa *Map* e *Reduce* e os números de *slots* de *Map* e *Reduce*.

O MRSG usa uma descrição simplificada da entrada de trabalho e dos custos da tarefa. Para a entrada, o usuário deve informar apenas seu tamanho. Portanto, não há dados reais para armazenar e processar na máquina de simulação. Esta

abordagem está diretamente ligada ao modo como o *SimGrid* simula a computação de tarefas. No *SimGrid*, o poder de computação das máquinas simuladas é medido em *flops* por segundo. Esta descrição simplificada permite aos usuários alterar o tamanho da tarefa, redefinindo os seus custos.

1.6. *Hands-on*

Nesta seção será apresentado um roteiro para a instalação prática do Hadoop, assim como, as configurações mais usuais para o desenvolvimento de pesquisas em computação intensiva em dados. Também será apresentado o processo de submissão de *jobs MapReduce* no Hadoop.

1.6.1. Instalação do Hadoop

A instalação que segue é baseada nas instruções do site oficial do Apache Hadoop em:

`http://hadoop.apache.org/docs/r0.20.2/quickstart.html`

e tem como objetivo ajudar os usuários a experimentar o Hadoop e seu sistema de arquivos distribuído em aplicações de computação intensiva em dados.

A estrutura é formada por um *cluster* com um *front-end* que é acessível aos usuários e programadores e, portanto, esta máquina terá duas placas de rede, uma para o acesso externo e outra para a conexão com as máquinas escravas (*slaves*) que irão fazer o processamento dos *jobs*.

Para instalar um *Cluster* Hadoop deve-se começar com a definição do hardware que vai ser utilizado para o *cluster* físico. Deve-se definir um nó *Master*, que vai gerenciar o desempenho do *cluster*, e os nós escravos que vão ser os encarregados da análise dos dados. Segundo a documentação, o Hadoop é suportado em plataformas Linux e Windows. Porém o Windows é somente suportado para desenvolvimento, e não para execução. Neste caso utiliza-se a distribuição Ubuntu Server 12.04 em todos os nós do *cluster*. Em cada nó deve-se instalar e configurar um serviço SSH ativo e instalar o Java 1.6 da SUN, conforme recomendação da documentação do Hadoop.

Torne a partição /opt a maior partição do disco local ou até mesmo coloque-a em um disco específico, pois nela serão armazenados os arquivos do sistema de arquivos distribuído do Hadoop o HDF.

1. **O serviço SSH ativo**. Para instalar o serviço SSH se digita:

```
$ sudo apt-get install ssh
```

Deve-se também configurar o serviço SSH para que os nós possam ser acessados sem usar senha, utilizando certificados digitais. Dentro do usuário que será utilizado para rodar Hadoop deve-se efetuar o seguinte comando para gerar a chave SSH.

2. **Para gerar as chaves pública e privada do ssh**

```
$ ssh-keygen -t dsa
```

3. **Após enviar a chave para o nó de destino**

```
cat HOME$/.ssh/id_dsa.pub | ssh user@ip_destino 'cat » .ssh/authorized_keys'
```

Onde user = usuário de login e ip_destino = ip do nó. O processo anterior se repete para cada um dos nós, e finalmente as chaves se distribuem para que todos os nós sejam visíveis mediante uma conexão SSH sem requerer senhas.

4. **Java**. A versão recomendada de Java é 1.6, e pode ser instalada com o seguinte comando:

```
$ sudo apt-get install sun-java6-jdk
```

Importante: Não utilize o Java da distribuição Linux, pois o mesmo não é totalmente compatível com o Hadoop. Deve-se conferir, para futura referência, que o diretório de instalação do Java, neste caso é:

```
/usr/lib/jvm/java-6-sun
```

5. **Hadoop**

Os arquivos para uma instalação do Hadoop podem ser descarregados desde a página oficial:

`http://hadoop.apache.org/releases.html#Download`

O arquivo descarregado deve ser descomprimido. Neste exemplo o diretório utilizado para descomprimir os arquivos e onde será configurada a implementação do Hadoop é:

```
\opt\hadoop
```

Após a instalação deve-se configurar o *cluster* e os arquivos do *master* e dos *slaves*. A estrutura destes arquivos formam a estrutura do *cluster*. Após descompactar os arquivos, você terá uma estrutura de diretórios semelhante ao que segue.

```
drwxr-xr-x  2 user group    4096 Feb 19  2013 bin
-rw-r-r-    1 user group   74035 Feb 19  2013 build.xml
drwxr-xr-x  4 user group    4096 Feb 19  2013 c++
-rw-r-r-    1 user group  348624 Feb 19  2013 CHANGES.txt
drwxr-xr-x  2 user group    4096 Feb 19  2013 conf
drwxr-xr-x 13 user group    4096 Feb 19  2013 contrib
drwxr-xr-x  7 user group    4096 Feb 19  2013 docs
-rw-r-r-    1 user group    6839 Feb 19  2013 hadoop-0.20.2-ant.jar
-rw-r-r-    1 user group 2689741 Feb 19  2013 hadoop-0.20.2-core.jar
-rw-r-r-    1 user group  142466 Feb 19  2013 hadoop-0.20.2-examples.jar
-rw-r-r-    1 user group 1563859 Feb 19  2013 hadoop-0.20.2-test.jar
-rw-r-r-    1 user group   69940 Feb 19  2013 hadoop-0.20.2-tools.jar
drwxr-xr-x  2 user group    4096 Feb 19  2013 ivy
-rw-r-r-    1 user group    8852 Feb 19  2013 ivy.xml
drwxr-xr-x  5 user group    4096 Feb 19  2013 lib
drwxr-xr-x  2 user group    4096 Feb 19  2013 librecordio
-rw-r-r-    1 user group   13366 Feb 19  2013 LICENSE.txt
-rw-r-r-    1 user group     101 Feb 19  2013 NOTICE.txt
-rw-r-r-    1 user group    1366 Feb 19  2013 README.txt
drwxr-xr-x 15 user group    4096 Feb 19  2013 src
drwxr-xr-x  8 user group    4096 Feb 19  2013 webapps
```

Descrição do conteúdo dos diretórios mais importantes do Hadoop:
\bin → executáveis do Hadoop.
\conf → arquivos de configuração.
\contrib → binários de terceiros.
\lib → bibliotecas do hadoop.
\src → códigos fontes.
\webapps → serviços web.

Crie um diretório em /etc onde serão armazenadas as configurações do Hadoop. A sugestão é criar **hadoop/conf** e após faça um link simbólico para os arquivos XML que estão em /opt/hadoop/conf.

1.6.2. Configurações em Arquivos XML

Os arquivos abaixo especificam as características do *cluster* e devem ser distribuídos para todas as máquinas.

/etc/hadoop/conf/core-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
        <property>
              <name>fs.default.name</name>
              <value>hdfs://ip_master_node:8900</value>
</configuration>
```

/etc/hadoop/conf/hdfs-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
     <property>
         <name>dfs.name.dir</name>
         <value>caminho-do-DFS</value>
     </property>
     <property>
         <name>dfs.data.dir</name>
         <value>caminho-do-DFS</value>
     </property>
     <property>
               <name>dfs.block.size</name>
               <value>tamanho-do-bloco</value>
        </property>
</configuration>
```

dfs.name.dir = local no sistema de arquivos no NameNode onde se armazena os metadados do HDFS, aplicado entre `<value>caminho-do-DFS</value>` .

dfs.data.dir = local no sistema de arquivos no DataNode onde se armazena os dados do HDFS, aplicado entre `<value>caminho-do-DFS</value>` .

dfs.block.size = tamanho em Byte do bloco 64MB = 67108864, aplicado entre `<value>tamanho-do-bloco</value>`.

/etc/hadoop/conf/mapred-site.xml

O arquivo mapred-site.xml segue o mesmo formato dos arquivos anteriores e as variáveis a serem ajustadas são:

mapred.job.tracker = informa-se o IP do *master*.

mapred.system.dir = informa-se o caminho no HDFS onde o MapReduce armazena os arquivos de sistema.

mapred.local.dir = informa-se o caminho no sistema de arquivos local onde os dados temporários são escritos, aplicado .

mapred.tasktracker.{map|reduce}.tasks.maximum = o número de tarefas de *map* ou *reduce* que rodam simultaneamente na máquina, definida conforme as *threads* dos processadores.

Ainda são necessários informar qual é o *master* e quem são os *slaves* isto é feito nos arquivos: **etc/hadoop/conf/master** para informar o *hostname* ou o endereço IP do nó *master* e **/etc/hadoop/conf/slaves** para informar o *hostname* ou o endereço IP dos nós escravos.

Após configurados os arquivos em uma máquina, a distribuição pode ser feita com o seguinte comando:

```
rsync -avHK /etc/hadoop/conf user@ip:/etc/hadoop
```

1.6.3. Um primeiro programa *MapReduce*

Antes de iniciar o *cluster* Hadoop, deve-se iniciar o sistema de arquivos distribuído no HDFS. Primeiro, deve-se formatar o sistema de arquivos, executando o seguinte comando:

```
$ bin/hadoop namenode -format
```

Depois iniciar o sistema de arquivos no NameNode:

```
$ bin/start-dfs.sh
```

O *script* `start-dfs.sh` também inicia o *daemon* do DataNode nos escravos indicados no arquivo /etc/hadoop/conf/slaves. Depois se inicia o *cluster* com o seguinte comando no JobTracker:

```
$ bin/start-mapred.sh
```

O *script* inicia também o *daemon TaskTracker* em todos os nós escravos indicados no arquivo *slaves* do diretório de configuração. Para rodar uma aplicação no *cluster* Hadoop, se copia os arquivos da entrada de dados para o HDFS:

```
$ bin/hadoop dfs -copyFromLocal <nome_do_arquivo> <nome_do_arquivo_no_hdfs>
```

Para submeter um *job* deve-se fazer o seguinte procedimento:

```
$ bin/hadoop jar /usr/share/hadoop/hadoop-examples-1.0.3.jar <nome_da_aplicação> <parametros_da_aplicação>
```

Dependendo do tipo de programa, os arquivos dos resultados da analise com Hadoop devem encontrar-se no sistema de arquivos distribuído:

```
$ bin/hadoop fs -cat output/*
```

Para suspender o *cluster* Hadoop e o sistema de arquivos, deve-se parar os serviços na ordem inversa na qual foram iniciados, assim:

```
$ bin/stop-dfs.sh
$ bin/stop-mapred.sh
```

1.7. Conclusão

Dentro do desenvolvimento científico, existiram diferentes maneiras de gerar conhecimento ao longo da história. No começo foi a ciência empírica, depois veio a formulação teórica, logo depois a simulação de fenômenos e, atualmente, o quarto paradigma tem a ver com o tratamento de grandes quantidades de dados, obtidas por experimentos e instrumentos científicos. Neste capítulo mostrou-se algumas técnicas que atualmente permitem formular o conhecimento a partir do processamento de uma grande quantidade de dados produzida.

A grande quantidade de dados exige novas arquiteturas e sistemas em hardware e software para seu tratamento. Dentro das arquiteturas que ajudam neste novo quarto paradigma estão as de computação paralela e distribuída, assim como aplicações e paradigmas de programação específicos para esse tipo de arquiteturas. Com a computação intensiva em dados três problemas principais se apresentam: a transferência dos dados entre os componentes do sistema distribuído, a quantidade de processamento para tratar os dados e que deve ser distribuída entre os componentes, e finalmente a questão do consumo energético do sistema.

O modelo MapReduce, proposto pela Google, permite tratar grandes quantidades de dados e também abstrair dos programadores a parte de paralelização e distribuição das tarefas entre os componentes do *cluster*. As tarefas *Map* e *Reduce* são altamente paralelizáveis e tem uma alta capacidade de escalonamento.

O *framework* do *MapReduce* foi pensado inicialmente para ambientes homogêneos de memória distribuída, mas pode ser implementado também em outras plataformas diferentes de um *cluster* homogêneo. Assim, o *MapReduce* mostra uma capacidade de trabalhar em diferentes plataformas e arquiteturas de hardware e software, que lhe permitem solucionar os problemas de transferência dos dados no sistema distribuído, e a quantidade de processamento; isto converte o MapReduce em uma ferramenta muito útil para o desenvolvimento do quarto paradigma da ciência.

1.8. Bibliografia

[AND 2007] ANDERSON, D. P.; MCLEOD, J. Local Scheduling for Volunteer Computing. **Parallel and Distributed Processing Symposium, International**, Los Alamitos, CA, USA, v.0, p.477, 2007.

[CAR 2007] CARDONA, K. et al. **A Grid Based System for Data Mining Using MapReduce**. [S.l.]: The AMALTHEA REU Program, 2007.

Acesso em Março 2011. (Technical Report TR-2007-02).

[CHE 2008] CHEN, S.; SCHLOSSER, S. W. **Map-Reduce Meets Wider Varieties of Applications**. [S.l.]: Intel Research Pittsburgh, 2008. (IRP-TR-08-05).

[DEA 2004] DEAN, J.; GHEMAWAT, S. MapReduce - Simplified Data Processing on Large Clusters. In: OSDI, 2004. **Anais...** [S.l.: s.n.], 2004. p.137–150.

[DEA 2010] DEAN, J.; GHEMAWAT, S. MapReduce - A Flexible Data Processing Tool. **Communications of the ACM**, New York, NY, USA, v.53, n.1, p.72–77, 2010.

[DON 2010] DONASSOLO, B. et al. Fast and Scalable Simulation of Volunteer Computing Systems Using SimGrid. In: Workshop on Large-Scale System and Application Performance (LSAP), 2010. **Anais...** [S.l.: s.n.], 2010.

[FAN 2011] FANG, W. et al. Mars: accelerating mapreduce with graphics processors. **IEEE Transactions on Parallel and Distributed Systems**, Los Alamitos, CA, USA, v.22, p.608–620, 2011.

[HAM 2010] HAMMOUD, S. et al. MRSim: a discrete event based mapreduce simulator. In: FSKD, 2010. **Anais...** IEEE, 2010. p.2993–2997.

[HEY 2009] HEY, T.; TANSLEY, S.; TOLLE, K. M. (Eds.). **The Fourth Paradigm: data-intensive scientific discovery**. [S.l.]: Microsoft Research, 2009. 263–289p.

[HIL 2010] HILBERT, M.; LOPEZ, P.; VASQUEZ, C. Information Societies: measuring the digital information-processing capacity of a society in bits and bytes. **The Information Society**, v.26, p.157–178, May 2010.

[ISA 2007] ISARD, M. et al. Dryad - Distributed Data-parallel Programs from Sequential Building Blocks. In: EuroSys '07 - Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, 2007, New York, NY, USA. **Anais...** ACM, 2007. p.59–72.

[LAK 2010] LAKSHMAN, A.; MALIK, P. Cassandra: a decentralized structured storage system. **SIGOPS Oper. Syst. Rev.**, New York, NY, USA, v.44, n.2, p.35–40, Apr. 2010.

[MEL 2011] MELL, P.; GRANCE, T. **The NIST Definition of Cloud Computing**. [S.l.: s.n.], 2011. Available http://csrc.nist.gov/publications/PubsSPs.html.

[MIC 2009] MICELI, C. et al. Programming Abstractions for Data Intensive Computing on Clouds and Grids. **9th IEEE/ACM International Symposium on Cluster Computing and the Grid, 2009. CCGRID '09.**, v.1, n.9, p.478–483, May 2009.

[MOR 2010] MORETTI, C. et al. All-Pairs - An Abstraction for Data-Intensive Computing on Campus Grids. **IEEE Trans. Parallel Distrib. Syst.**, Piscataway, NJ, USA, v.21, n.1, p.33–46, 2010.

[RAN 2007] RANGER, C. et al. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In: HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture, 2007, Washington, DC, USA. **Anais...** IEEE Computer Society, 2007. p.13–24.

[TAN 2009] TANG, H. **Mumak: map-reduce simulator**. [S.l.]: Apache Software Foundation, 2009. [ONLINE], Disponível em https://issues.apache.org/jira/browse/MAPREDUCE-728. (Technical Report MAPREDUCE-728).

[WAN 2009] WANG, G. et al. Using Realistic Simulation for Performance Analysis of MapReduce Setups. In: LSAP '09: Proceedings of the 1st ACM workshop on Large-Scale System and Application Performance, 2009, New York, NY, USA. **Anais...** ACM, 2009. p.19–26.

[WHI 2009] WHITE, T. **Hadoop - The Definitive Guide**. 1.ed. [S.l.]: OReilly Media, Inc., 2009. v.1.

[WIL 2009] WILDE, M. et al. Parallel Scripting for Applications at the Petascale and Beyond. **Computer**, Los Alamitos, CA, USA, v.42, p.50–60, 2009.