UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

WAGNER KOLBERG

Simulação e Estudo da Plataforma Hadoop MapReduce em Ambientes Heterogêneos

Trabalho de Conclusão

Prof. Dr. Cláudio Fernando Resin Geyer
Orientador

Porto Alegre, Dezembro de 2010

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
CPU	Central Processing Unit
DTD	Document Type Definition
DFS	Distributed File System
GFS	Google File System
GPU	Graphics Processing Unit
HDFS	Hadoop Distributed File System
LATE	Longest Approximate Time to End
MRSG	MapReduce-SimGrid
XML	Extensible Markup Language

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

MapReduce é um modelo de programação voltado à computação paralela em larga escala, e ao processamento de grandes volumes de dados. A implementação do modelo, e as suposições feitas em relação ao ambiente sobre o qual será executado, influenciam fortemente no tempo de computação dos *jobs* submetidos.

O Hadoop, uma das implementações mais populares do MapReduce, e que será estudada neste trabalho, supõe que o ambiente de execução é homogêneo, prejudicando o desempenho do *framework* quando a grade apresenta um certo nível de heterogeneidade no que toca a capacidade de processamento das máquinas que a constituem.

Como ferramenta de análise para as adaptações propostas, é desenvolvido um simulador para o MapReduce — tendo como base o simulador de grades SimGrid — com o objetivo de facilitar a implementação e avaliação de novos algoritmos de escalonamento de tarefas e distribuição de dados, dentre outros. Dentre as vantagens proporcionadas pelo uso do simulador é possível citar: a facilidade na implementação de algoritmos teóricos; a agilidade em testes para uma grande variedade de configurações; e a possibilidade de avaliar rapidamente a escalabilidade de algoritmos sem custos de infraestrutura.

Em relação ao simulador, é ainda apresentada uma validação de seu comportamento em relação ao Hadoop MapReduce, comparando execuções do sistema em uma grade, com simulações que emulam as configurações reais. Uma vez validado o simulador, o mesmo é utilizado para avaliar as adaptações do Hadoop a ambientes heterogêneos.

Os resultados obtidos, tanto com a validação do simulador, quanto com a implementação das adaptações propostas, apresentaram resultados positivos, demostrando que é viável utilizar simulação para estudar e avaliar diferentes implementações para o modelo MapReduce.

Este trabalho, portanto, consiste em um estudo do funcionamento interno do Hadoop MapReduce, seu comportamento em ambientes heterogêneos, e também propõe um novo simulador, com os recursos necessários para avaliar adaptações em implementações do MapReduce.

Palavras-chave: Grade, MapReduce, Hadoop, ambientes heterogêneos, escalonamento, framework, programação paralela, simulação.

Simulation and Study of the Hadoop MapReduce Platform on Heterogeneous Environments

ABSTRACT

MapReduce is a programming model for large-scale parallel computing, and for processing large data sets. The model implementation, and the assumptions made about the running environment, strongly affect the job execution time.

Hadoop, one of the most popular implementations of the MapReduce model, that will be studied in this work, assumes that the execution environment is homogeneous, deprecating its performance when the grid presents a certain level of heterogeneity, concerning the computation power of its nodes.

As an analysis tool, a MapReduce simulator — having the SimGrid simulator as its base system — is developed to easily implement and evaluate new task scheduling and data distribution algorithms. As advantages that a simulator provides, it is possible to name: the simplified development of theoretical algorithms; the agility to test a great variety of configurations; and the possibility to quickly evaluate algorithm's scalability without infrastructure costs.

The simulator has its behavior validated against the Hadoop MapReduce, through comparisons of real executions of the framework on a real grid environment, and simulations that emulate the configurations used on the real execution. Once the simulator is validated, it is used to evaluate the modifications to the MapReduce algorithms on heterogeneous environments.

The results of the simulator validation, and the evaluation of the proposed modifications, were positive, showing that it is possible to use simulation to study and evaluate different MapReduce implementations.

Therefore, this work consists of a study about the Hadoop MapReduce, its behavior on heterogeneous environments, and it also proposes modifications in this framework to improve its performance on this kind of environment. To evaluate the proposed adaptations, a MapReduce simulator was developed, and it will also be presented in this study.

Keywords: Grid, MapReduce, Hadoop, heterogeneous environments, scheduling, framework, parallel programming, simulation.

1 INTRODUÇÃO

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. Como exemplos de computação intensiva de dados, (WHITE, 2009) aponta: a bolsa de valores de Nova Iorque, que gera um terabyte de dados por dia; o Facebook, que hospeda 10 bilhões de fotos (totalizando um petabyte de armazenamento); e o Large Hadron Collider (LHC), que deve produzir cerca de 15 petabytes de dados por ano. Empresas na área de busca, e.g. Google e Yahoo, também são ótimos exemplos, pois indexam trilhões de páginas da internet, e atendem bilhões de requisições diariamente.

De acordo com a IDC (International Data Corporation), a quantidade de informação criada, capturada ou replicada em meio digital no ano de 2009 apresentou um crescimento de 62%, alcançando aproximadamente 800.000 petabytes. Para 2010, acredita-se que este valor chegue a 1.2 milhões de petabytes. Já em 2020 o crescimento esperado deve alcançar os 35 zetabytes, equivalente a 35 milhões de petabytes (GANTZ; REINSEL, 2010).

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 software 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.

A fim de facilitar este processo, a multinacional Google desenvolveu o *MapReduce*, um modelo de programação paralela para processamento largamente distribuído de grandes volumes de dados (DEAN; GHEMAWAT, 2008).

Além do *framework* da Google, diversas implementações para o MapReduce foram desenvolvidas, dentre as quais, a mais conhecida e divulgada está inserida no projeto *Hadoop* (HADOOP, 2010a), mantido pela Apache Software Foundation. Um aspecto importante desta implementação é seu escalonador de tarefas, o qual supõe que o ambiente distribuído é homogêneo. Assim, quando submetido à execução em ambientes heterogêneos, o Hadoop MapReduce apresenta um comportamento inadequado, prejudicando seu desempenho (KONWINSKI, 2009; ZAHARIA et al., 2008).

1.1 Motivação

Há muito constatou-se que as estações de trabalho apresentam longos períodos de ociosidade, principalmente durante a noite e finais de semana, quando ficam aproximadamente 95% do tempo disponíveis (FOSTER; KESSELMAN, 2003), utilizando, ao longo do tempo, apenas 30% de sua capacidade de processamento (MUTKA; LIVNY, 1988).

Pesquisadores passaram, então, a estudar o uso de estações de trabalho como recursos de processamento distribuído, em alternativa aos *clusters* de dedicação exclusiva, que apresentam um custo muito mais elevado. Este tipo de ambiente é conhecido por *Desktop Grid* (FOSTER; KESSELMAN, 2003), e tem como uma importante característica a heterogeneidade de seus nós.

Este trabalho tem como objetivo estudar o Hadoop MapReduce, seu comportamento em ambientes heterogêneos, e também propor uma alternativa ao escalonador atual, a fim de melhorar o desempenho do *framework* nestas condições.

As adaptações propostas ao escalonador do Hadoop, visam o beneficio dos usuários de Desktop Grids, uma vez que as janelas de disponibilidade das estações que constituem estas grades poderão ser aproveitadas de maneira mais eficiente.

Espera-se, também, que este aperfeiçoamento do Hadoop para Desktop Grids promova a utilização deste tipo de ambiente, o que leva a um melhor aproveitamento de recursos já disponíveis nas organizações, sem a necessidade da aquisição de novos *clusters*. Esta prática impacta fortemente no que toca a questão de economia energética e, consequentemente, emissão de dióxido de carbono, compactuando com os conceitos de *Green Computing* (MURUGESAN, 2008), indispensáveis em nossa atualidade para alcançar um ambiente de processamento computacional sustentável.

1.2 Organização do Texto

A continuidade deste texto está organizada como descrito a seguir. No capítulo 2 são apresentadas as tecnologias relacionadas com este trabalho. No capítulo 3 é realizado um estudo sobre o comportamento do MapReduce em ambientes heterogêneos, no qual são indicados possíveis problemas, e apresentadas propostas para a solução dos mesmos. Em seguida, nos capítulos 4 e 5, passa-se à descrição do simulador desenvolvido durante o trabalho e do ambiente de testes. Por fim, nos capítulos 6 e 7, são respectivamente expostos os resultados obtidos na validação do simulador e na implementação das propostas do capítulo 3, e uma conclusão com as considerações finais do trabalho.

2 TECNOLOGIAS E CONCEITOS ENVOLVIDOS

Neste capítulo serão apresentadas as tecnologias envolvidas no estudo realizado. O funcionamento de cada tecnologia é exposto de maneira detalhada, a fim de permitir a compreensão dos problemas encontrados e das soluções propostas neste trabalho.

São expostos, também, os conceitos básicos por trás destas tecnologias. Estes conceitos permitem ao leitor contextualizar o estudo sendo apresentado, e facilitam a compreensão das técnicas utilizadas para a análise e solução de problemas.

2.1 Processamento Distribuído

Com o avanço tecnológico das últimas décadas, os computadores passaram a ficar cada vez mais populares, e consequentemente seu custo foi reduzido drasticamente, permitindo a multiplicação desses dispositivos tanto em ambientes empresariais, quanto domésticos. Este crescimento, aliado a avanços em áreas como armazenamento e comunicação de informações, possibilitou a colaboração entre recursos computacionais para a solução de problemas. Desta distribuição do trabalho que antes era realizado por uma única máquina, surgiu o paradigma de *processamento distribuído* (TOTH, 2008).

Esta linha de pesquisa envolve vários outros conceitos — relacionados com infraestrutura, técnicas de distribuição e balanceamento de carga, etc. — que serão comentados a seguir.

2.1.1 Clusters

Um *cluster* pode ser definido como um conjunto de máquinas, administrado e utilizado por um único indivíduo, com o objetivo de solucionar problemas que levariam muito tempo em uma única estação de trabalho (TOTH, 2008).

Dentre algumas características frequentemente observadas em um *cluster*, é possível destacar:

- O *cluster* é constituído por máquinas de prateleira, apresentando baixo custo se comparado a supercomputadores.
- Todos os nós estão geograficamente próximos, geralmente em um mesmo prédio.
- As conexões entre as máquinas possuem altas taxas de transferência.
- As máquinas são homogêneas, i.e. possuem capacidades de processamento similares.

2.1.2 Computação em Grade

Uma Grade Computacional (do inglês, *Grid*) consiste de um sistema que coordena recursos de maneira não centralizada, utilizando protocolos e interfaces padronizadas, abertas, e de propósitos gerais, a fim de prover uma variedade de serviços complexos de acordo com a demanda de seus usuários (FOSTER; KESSELMAN, 2003).

Ainda de acordo com Foster, o termo "*Grid*" é utilizado em analogia às redes de energia elétrica (chamadas de *power grids* em inglês), as quais proveem acesso difuso à eletricidade (FOSTER; KESSELMAN, 2003). Assim deveria ser nas grades, com o compartilhamento flexível de recursos computacionais, em forma de ferramentas colaborativas que poderiam ser acessadas de qualquer ponto.

2.1.3 Desktop Grid

Desktop Grids são grades computacionais que, em contraste aos *clusters*, apresentam as seguintes propriedades:

- Os nós da grade são tipicamente heterogêneos.
- As taxas de transferência das conexões também podem variar, podendo apresentar ligações extremamente lentas.
- Os nós não estão necessariamente centralizados em um mesmo local, e podem estar dispersos por todo o mundo, conectando-se através da Internet.
- Os nós são voláteis, i.e. as máquinas podem parar de responder (ou voltar a fazê-lo) a qualquer momento.

Enterprise Desktop Grid

As *Enterprise Desktop Grids* (KONDO et al., 2007), ou *Desktop Grids* Empresariais, são restritas à rede local de uma única corporação. Esta restrição normalmente implica em uma rede mais rápida, uma vez que é comum observar este tipo de grade com conexões de 100Mb/s e 1Gb/s.

2.1.4 Balanceamento de Carga

O balanceamento de carga é uma prática utilizada para atingir um aproveitamento ótimo dos recursos do sistema distribuído, através de uma politica de alocação de trabalho coerente com a capacidade de processamento dos dispositivos do sistema, a fim de obter o mesmo nível de esforço em todos os recursos (TOTH, 2008).

2.2 MapReduce

O MapReduce, criado pela Google, é um modelo de programação paralela para processamento largamente distribuído de grandes volumes de dados. Seu objetivo é *facilitar* a programação de aplicativos distribuídos com este perfil. Para tal, o modelo inspira-se nas primitivas *Map* e *Reduce* presentes em diversas linguagens funcionais, como Lisp e Haskell, por exemplo. Esta abordagem foi adotada pois verificou-se que, em muitos casos, era necessário mapear fragmentos dos dados de entrada a uma chave identificadora, e então processar todos os fragmentos que compartilhassem a mesma chave (DEAN; GHEMAWAT, 2008).

Assim, a tarefa principal do programador é implementar estas duas funções, indicando como o mapeamento e redução dos dados serão computados. 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*. Durante a execução, as funções recebem e emitem dados no formato de pares $\langle chave, valor \rangle$. Como o tipo destes elementos dependem da aplicação que será executada, cabe ao desenvolvedor, também, definir estas propriedades.

Um exemplo de uso do MapReduce é demonstrado na Figura 2.1, a qual contém uma aplicação cujo objetivo é contar a quantidade de ocorrências de cada palavra em um documento. No pseudocódigo, 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`, por sua vez, recebe como entrada uma palavra (`chave`), e um iterador para todos os valores emitidos pela função `map`, associados com a palavra em questão. Todos os valores são então somados e um par chave/valor, contendo a palavra e seu total de ocorrências, é emitido.

```
1  Function map (Integer chave, String valor):
2    #chave: número da linha no arquivo.
3    #valor: texto da linha correspondente.
4    listaDePalavras = split (valor)
5    for palavra in listaDePalavras:
6      emit (palavra, 1)
7
8  Function reduce (String chave, IntegerIterator valores):
9    #chave: palavra emitida pela função map.
10   #valores: conjunto de valores emitidos para a chave.
11   total = 0
12   for v in valores:
13     total = total + 1
14   emit (palavra, total)
```

Figura 2.1: Pseudocódigo de programação no MapReduce

Para auxiliar no entendimento, a Figura 2.2 ilustra o fluxo de execução e dados para esta aplicação. Como entrada é utilizado um arquivo texto com apenas duas linhas, cujos conteúdos são “primeira linha” e “segunda linha”.

Para cada linha de texto do arquivo de entrada é feita uma chamada à função `map`. Logo, como o arquivo possui duas linhas, duas chamadas são realizadas. Em seguida, cada função `map` gera n pares $\langle chave, valor \rangle$ intermediários, onde, nesta aplicação, n é igual à quantidade de palavras contidas na linha de texto recebida pela função. Como cada linha possui duas palavras, um total de quatro pares intermediários são criados. Na fase de redução, todos os pares intermediários que estão associados a uma mesma chave (neste caso, uma palavra do documento) são passados para uma chamada da função `reduce`. Portanto, como existem três palavras distintas (“primeira”, “linha” e “segunda”), três reduções são executadas. Por fim, a execução da função `reduce` retorna a soma de todos os valores presentes na lista de pares recebidos. Os resultados finais são armazenados no arquivo de saída.

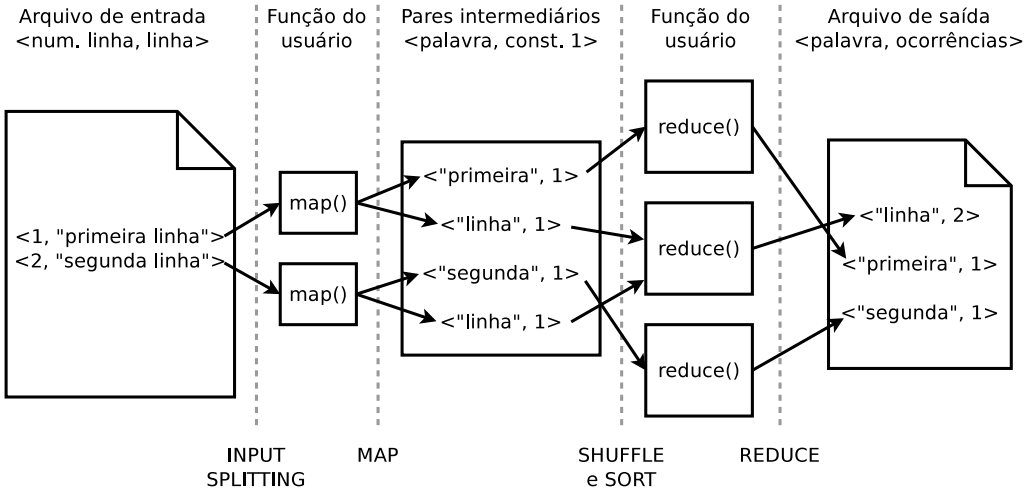

Figura 2.2: Fluxo simplificado de execução e dados do MapReduce

Outros exemplos de programas facilmente expressos no modelo MapReduce são: grep distribuído, frequência de acesso a URLs, grafo reverso de links web, índice invertido, ordenamento distribuído, dentre outros (DEAN; GHEMAWAT, 2008).

2.2.1 Arquitetura do Framework

O modelo MapReduce pode ser executado sobre uma variedade de plataformas e ambientes distintos. Logo, a melhor implementação do *framework* depende do ambiente alvo. Uma implementação feita para utilizar a GPU de uma máquina, por exemplo, provavelmente será beneficiada por um comportamento distinto a uma implementação destinada a um grande *cluster*.

O Google MapReduce foi desenvolvido para grandes *clusters* de máquinas de prateleira[1], interligadas por uma rede do tipo *switched Ethernet* (DEAN; GHEMAWAT, 2008), e é constituído por basicamente dois tipos de nós: *Master* e *Worker* (denominados *Mestre* e *Trabalhador* em português, respectivamente).

O nó mestre tem como função atender requisições de execução (denominadas *jobs*[2] em inglês) efetuadas pelos usuários, e gerenciá-las, criando várias tarefas (do inglês *tasks*) e delegando-as aos nós trabalhadores, que por sua vez são encarregados de executar de fato essas tarefas, aplicando, de acordo com seu tipo, as funções *map* ou *reduce* definidas pelo usuário. Como é possível perceber, trata-se de uma típica arquitetura *mestre-escravo* (do inglês *master-slave*) (DUBREUIL; GAGNÉ; PARIZEAU, 2006).

A arquitetura compreende, ainda, um sistema de arquivos distribuído, onde ficam armazenados os dados utilizados como entrada para os *jobs*. Para evitar a transferência excessiva de dados, os *workers* do MapReduce são também nós do sistema de arquivos. Mais detalhes sobre este recurso são apresentados nas seções 2.2.3 e 2.3.3.

[1]Computadores de baixo a médio custo. Sem hardware tolerante a falhas, por exemplo. Em inglês este tipo de equipamento é denominado *Commodity Machine*.

[2]Em mais detalhes, um *job* refere-se a cada solicitação de execução realizada por um usuário, e.g. executar o programa que computa a frequência de palavras. Cada *job*, por sua vez, é constituído por diversas tarefas (*tasks*) de mapeamento e redução, criadas pelo próprio *framework*.

2.2.2 Etapas e Otimizações Internas

Além das fases de mapeamento e redução, que consistem na execução de fato das funções map e reduce criadas pelo programador, quatro outras etapas são características durante a execução do *framework* MapReduce. Elas são conhecidas como *Input splitting*, *Shuffle*, *Sort* e *Combine*. As três primeiras são referenciadas, porém não detalhadas, na Figura 2.2.

A etapa *Input splitting* (VENNER, 2009) consiste na leitura e divisão dos dados de entrada. Cada intervalo gerado, chamado de *input split*, normalmente varia de 16 a 64 megabytes (diretamente relacionado com o tamanho dos blocos do sistema de arquivos distribuído) e é utilizado como entrada para uma tarefa de mapeamento. Após a divisão, cada tarefa lê o conteúdo de seu *input split* e gera pares chave/valor que serão passados a várias chamadas da função *map* definida pelo usuário. O componente responsável pela leitura dos *input splits* é conhecido como *Reader* (ou *leitor* em português) (DEAN; GHEMAWAT, 2008).

Cada implementação do modelo MapReduce possui uma variedade de leitores e, normalmente, permite que os programadores desenvolvam seus próprios componentes. Assim o programador pode criar *readers* para casos específicos, como ler tuplas de um banco de dados ou registros de um arquivo binário, por exemplo. Vale ressaltar, ainda, que cada *input split* recebido por uma **tarefa** de mapeamento, pode gerar várias chamadas da **função** *map* criada pelo usuário. No exemplo da seção 2.2, uma tarefa de mapeamento chamaria a função para cada linha de texto contida no *input split*, por exemplo. Esta possibilidade de extensão permite que o *framework* não apenas suporte uma variedade de sistemas de armazenamento, mas possibilita, também, a utilização de índices pré-gerados sobre os dados de entrada. Índices de bancos de dados e *log rollover* são exemplos citados em (DEAN; GHEMAWAT, 2010).

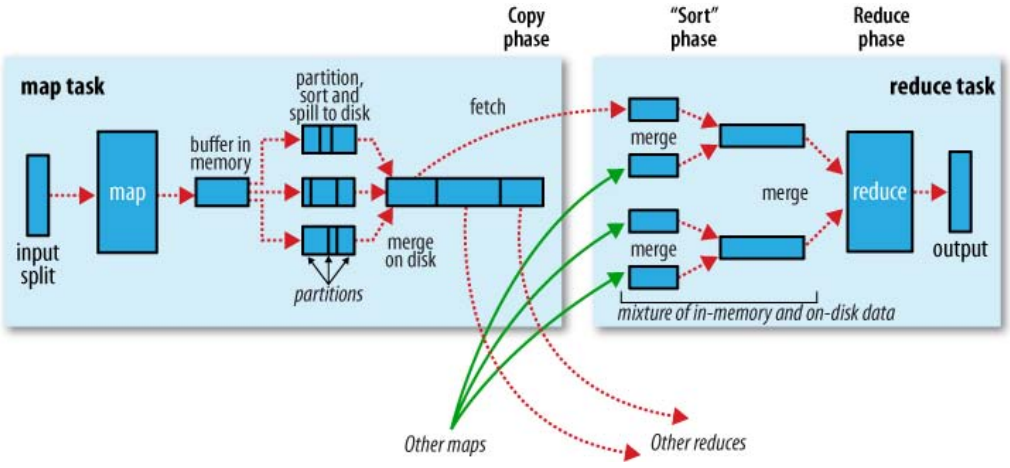

Figura 2.3: Fases de Shuffle e Sort no MapReduce (WHITE, 2009)

A fase *Shuffle* (VENNER, 2009; WHITE, 2009), realizada a partir do momento em que cada tarefa de mapeamento passa a produzir pares intermediários, é dividida em dois passos distintos: *particionamento* e *ordenamento*. No primeiro, os pares chave/valor são divididos em R partições de acordo com a tarefa de redução de destino de cada chave. Logo, o valor de R corresponde à quantidade de tarefas de redução do *job*. Já no passo de ordenamento, as chaves pertencentes a uma mesma partição são ordenadas para facilitar posterior processamento. É importante destacar que cada **tarefa** de redução pode processar várias chaves, porém uma única chamada à função *reduce* do usuário será feita para

cada chave, i.e. 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 destinadas. Estes dados, contudo, estão espalhados pelos nós da rede, pois o mapeamento normalmente é realizado em mais de um *worker*. Portanto, o redutor deve copiar e fundir as partições a ele destinadas, e que foram geradas durante o *Shuffle*. Este processo de aglutinação de partições comuns é realizado na fase *Sort* (VENNER, 2009; WHITE, 2009). A etapa recebe este nome pois a fusão dos dados dá-se através de um *merge-sort*, otimizado pelo passo de ordenamento realizado no *Shuffle*, e que resulta em uma garantia de saída ordenada (por partição). A Figura 2.3 ilustra o fluxo de dados nas fases de *Shuffle* e *Sort*.

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 (DEAN; GHEMAWAT, 2008). Esta etapa ocorre em cada *worker* que executa uma tarefa Map, após o passo de ordenamento realizado durante o *Shuffle*. Em muitos casos, a função *reduce* do usuário é associativa e comutativa, tornando possível o processamento disjunto dos valores associados a uma mesma chave. A aplicação que computa a frequência de palavras é um bom exemplo de tais propriedades. Considere que a função `combine` é igual à função `reduce` e que o arquivo de entrada contém a seguinte linha de texto duas vezes: “palavra palavra”.

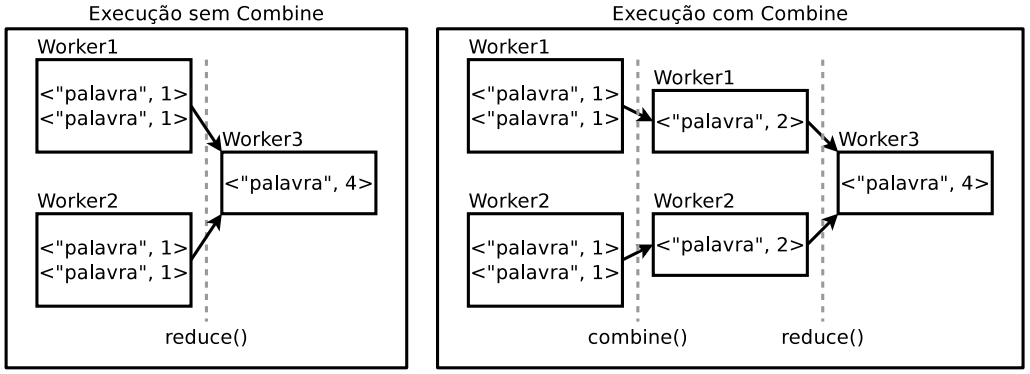

Figura 2.4: Fluxo de dados da função Combine

O fluxo de dados para o exemplo em questão é demonstrado na Figura 2.4, onde os dados iniciais, mais à esquerda de cada bloco, representam os pares intermediários emitidos pelas instâncias da função `map` (que são duas, devido à quantidade de linhas da entrada). Neste exemplo dois workers distintos realizam os mapeamentos. É possível perceber que sem a função `combine` o redutor teria que buscar e processar quatro pares intermediários, enquanto que, ao utilizar o *combine*, somente dois pares são manipulados na redução. Logo, verifica-se que esta etapa reduz o tráfego de informações na rede e a quantidade de trabalho centralizado no redutor.

2.2.3 GFS

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

os pares intermediários. É possível, ainda, observar que um *worker* responsável por um mapeamento, pode vir a realizar a redução dos dados computados durante o Map, evitando a cópia desses dados. Um outro importante recurso utilizado pelo *framework*, é a utilização de um *sistema de arquivos distribuído* (ou apenas DFS, abreviado do inglês *Distributed File System*), o qual influencia fortemente a implementação do MapReduce. No *framework* da Google, este recurso é representado pelo GFS.

O *Google File System* (GFS) é um sistema de arquivos distribuído para aplicações com processamento intensivo de dados (GHEMAWAT; GOBIOFF; LEUNG, 2003). Sua arquitetura consiste em um nó *master*, e diversos *chunkservers*. Ambos os tipos de nós são constituídos por máquinas de prateleira, rodando servidores a nível de usuário. A arquitetura, ilustrada na Figura 2.5, considera, ainda, máquinas clientes (*clients*) que acessam o sistema de arquivos concorrentemente.

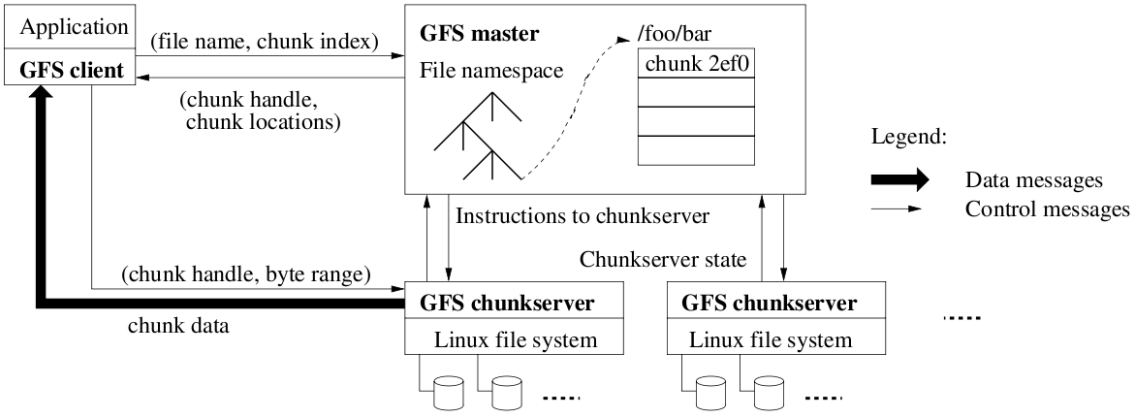

Figura 2.5: Arquitetura do GFS (GHEMAWAT; GOBIOFF; LEUNG, 2003)

O nó mestre é encarregado de manter e controlar todos os metadados do sistema de arquivos, incluindo espaço de nomes (*namespace*), mapeamento de arquivos a *chunks* ("pedaços" de arquivos, que podem ser traduzidos como *blocos* do sistema de arquivos), localização dos *chunks*, dentre outros. O nó mestre centraliza, também, diversas atividades como balanceamento de carga dos *chunkservers*, *garbage collection*, e atendimento a requisições dos clientes, por exemplo.

Os *chunkservers*, por sua vez, possuem a tarefa de armazenar os dados em si, e enviá-los *diretamente* aos clientes que os requisitaram. Esta característica é fundamental para o bom desempenho do sistema.

Como comentado, existe uma forte ligação entre o DFS e a implementação do MapReduce. Isto ocorre pois 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 já possua, em sua unidade de armazenamento local, uma réplica 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[3] ao *chunkserver* que possui os dados (DEAN; GHEMAWAT, 2008).

[3]O conceito de proximidade em redes é bastante subjetivo, e pode referir-se, por exemplo, à quantidade de nós em um caminho, ao tempo de transferência de dados entre dois nós, e até mesmo à proximidade geográfica. No contexto do *framework* MapReduce, considera-se que dois nós são próximos quando eles compartilham uma *storage* de dados no *cluster*, ou estão ligados a um mesmo *switch*, por exemplo.

Assim, quando executa-se grandes operações MapReduce, envolvendo uma significante fração de nós do *cluster*, a maioria dos dados são lidos localmente e o consumo de banda é praticamente nulo (DEAN; GHEMAWAT, 2008).

2.2.4 Backup Tasks

Considerando a arquitetura e funcionamento do MapReduce, seus criadores identificaram um possível problema de desempenho causado por máquinas cuja performance está aquém dos demais nós da grade. Essas máquinas são denominadas *stragglers* (DEAN; GHEMAWAT, 2008).

Uma máquina pode tornar-se lenta por diversos problemas de hardware e software. Neste caso, as tarefas executadas em um *straggler* atrasam o processamento como um todo, especialmente quando a lentidão das tarefas ocorre no final de um *job* (DEAN; GHEMAWAT, 2008).

Para contornar este problema no MapReduce, foram introduzidas as *Backup Tasks*. Estas tarefas de apoio são cópias de tarefas em andamento no final de operações de mapeamento e redução. Quando qualquer uma das cópias de uma tarefa é finalizada com sucesso, as demais são encerradas (DEAN; GHEMAWAT, 2008).

A execução das *backup tasks* resulta em um alto ganho de desempenho nos *jobs* do MapReduce. Como exemplo deste ganho, (DEAN; GHEMAWAT, 2008) aponta um algoritmo de ordenamento que, sem a utilização das *backup tasks*, apresenta um aumento de 44% no tempo de execução do *job* em relação ao comportamento normal (com *backup tasks*).

2.3 Hadoop

Uma das implementações mais conhecidas do MapReduce faz parte do projeto Hadoop, mantido pela Apache Software Foundation (APACHE, 2010), e que tem como finalidade desenvolver software livre para computação distribuída, escalável e confiável (HADOOP, 2010a). O *Hadoop MapReduce* é uma implementação em Java do modelo e *framework* criado pela Google, o qual foi originalmente desenvolvido em C++. Um exemplo de programação no Hadoop pode ser visto no Anexo A.

Os conceitos que servem como base para o *framework* do Hadoop seguem o modelo definido no trabalho de (DEAN; GHEMAWAT, 2008), o qual foi apresentado nas seções anteriores. Logo, seu funcionamento é extremamente semelhante ao *framework* da Google.

Google	**Hadoop**
Master	JobTracker
Worker	TaskTracker
Backup task	Speculative task
GFS Master	HDFS NameNode
GFS Chunkserver	HDFS DataNode

Tabela 2.1: Comparação da terminologia utilizada no Hadoop

Em alguns pontos da implementação, a terminologia utilizada difere do que foi previamente apresentado, porém é possível relacionar os termos equivalentes. Os conceitos de

nó *Master* e *Worker*, por exemplo, são respectivamente denominados *JobTracker* e *TaskTracker* no Hadoop. A Tabela 2.1 relaciona mais algumas equivalências de nomenclatura e tecnologias.

2.3.1 Escalonamento de Tarefas

Assim como o *framework* da Google, o Hadoop MapReduce apresenta uma arquitetura cliente-servidor. O código dos *workers*, portanto, é constituído de um simples *heartbeat*[4] que envia o estado do nó ao mestre, e aguarda tarefas para então processá-las.

Quando um *worker* indica que está disponível para receber mais trabalho, o nó mestre lhe atribui uma tarefa seguindo o processo descrito a seguir.

1. Se ainda existe alguma tarefa de mapeamento a ser finalizada, o nó mestre procura, em ordem, uma tarefa

   (a) que processe dados de um bloco local, já armazenado no *worker* (uma vez que estes são também nós do HDFS).

   (b) que processe dados que estão em um outro nó (neste caso o *worker* deve receber o bloco pela rede diretamente do nó que o possui).

   (c) especulativa (tarefas já em execução, porém lentas).

2. Se não existem mais tarefas de mapeamento, o nó mestre passa a escalonar as tarefas de redução. Primeiramente tarefas pendentes e, por fim, tarefas especulativas. Na redução não existe o conceito de blocos locais, uma vez que a entrada para estas tarefas consiste dos resultados das tarefas de mapeamento, que estão distribuídos entre os nós da grade.

Existe ainda um escalonamento entre tarefas de *jobs* distintos. Atualmente o algoritmo utilizado é denominado *Fair Scheduler*, e busca uma distribuição justa da capacidade do *cluster* entre os usuários (WHITE, 2009). O escalonamento entre *jobs*, contudo, não será abordado neste trabalho.

2.3.2 Execução Especulativa

No Hadoop, a *execução especulativa* (do inglês *speculative execution*) (WHITE, 2009) representa o mecanismo de *backup tasks* do *framework* da Google.

As tarefas especulativas são executadas nos nós do *cluster* somente quando todas as outras tarefas (*map* ou *reduce*)[5] regulares já foram concluídas, e somente para tarefas que já estão executando há pelo menos um minuto e não conseguiram atingir o mesmo progresso, em média, que as demais tarefas do *job* (WHITE, 2009).

Para identificar as máquinas lentas, cada nó do *cluster* envia seu progresso, e outras informações, através do sinal de *heartbeat* (WHITE, 2009). Para tarefas de mapeamento, o progresso é calculado através da fração de dados já processados, enquanto que nas tarefas de redução, cada fase (cópia, ordenamento e redução) representa $1/3$ do andamento (cada fase, contudo, tem seu progresso determinado em função dos dados processados, como no mapeamento) (ZAHARIA et al., 2008).

[4] Mensagem enviada periodicamente ao nó mestre, contendo informações sobre o estado do *worker*.

[5] O mecanismo de execução especulativa atua no fim das fases de mapeamento e redução de maneira independente. No Hadoop é possível desativar o mecanismo em qualquer uma das fases, ou ambas.

2.3.3 HDFS

Assim como sua implementação do MapReduce, o projeto Hadoop provê uma alternativa Java para o GFS. O *Hadoop Distributed File System* (HDFS) é um sistema de arquivos distribuído altamente tolerante a falhas projetado para rodar sobre hardware de prateleira (HADOOP, 2010b).

A plataforma Hadoop suporta diversos sistemas de arquivos distintos, como Amazon S3 (Native e Block-based), CloudStore, HAR, sistemas mantidos por servidores FTP e HTTP/HTTPS (este último apenas como leitura), Local (destinado a unidades de armazenamento conectadas localmente), dentre outros (WHITE, 2009). O HDFS, contudo, é seu sistema de arquivos padrão.

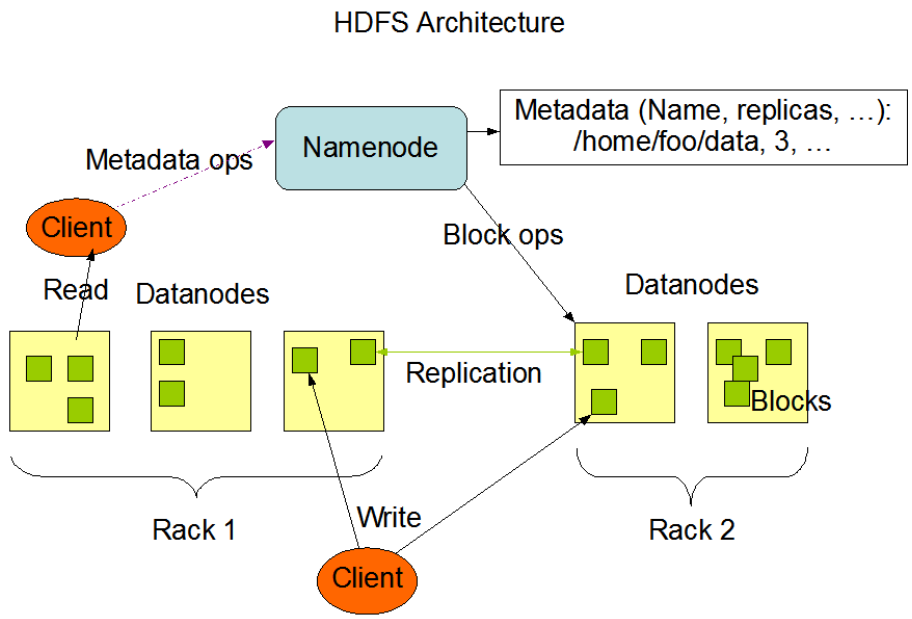

Figura 2.6: Arquitetura do HDFS (HADOOP, 2010b)

A arquitetura do HDFS, ilustrada na Figura 2.6, segue os moldes do GFS, apresentados em (GHEMAWAT; GOBIOFF; LEUNG, 2003) (HADOOP, 2010c). Assim como o sistema de arquivos da Google, o HDFS possui um nó mestre (denominado *NameNode*) responsável por atender requisições dos usuários e gerenciar a localização dos dados, e vários nós de dados (*DataNodes*) que armazenam e transmitem os dados aos usuários que os requisitarem.

Outra propriedade importante do HDFS é a distribuição de dados. O sistema de arquivos busca manter um *balanceamento* na porcentagem de ocupação das storages que o constituem através da redistribuição e re-replicação de blocos (WHITE, 2009).

Uma importante funcionalidade do HDFS, e que impacta fortemente no desempenho do MapReduce, é conhecida como *rack awareness*. Através desse recurso, o HDFS é capaz de identificar quais DataNodes pertencem a um mesmo *rack*, e a partir dessa informação, distribuir as réplicas de maneira mais inteligente aumentando a performance e resiliência do sistema (WHITE, 2009).

2.4 Considerações Finais

Este capítulo proveu um conhecimento razoavelmente detalhado das tecnologias necessárias para a compreensão e contextualização dos problemas estudados neste trabalho. Foi possível observar, também, que a implementação do Hadoop MapReduce, e o ambiente alvo do *framework*, são muito próximos à solução da Google.

O próximo capítulo avalia possíveis problemas nos sistemas apresentados, fazendo uso das informações deste capítulo.

3 MAPREDUCE EM AMBIENTES HETEROGÊNEOS

A seguir é realizado um estudo sobre o comportamento do Hadoop MapReduce em ambientes heterogêneos. São apontados possíveis problemas de desempenho advindos da implementação do *framework*, e posteriormente são propostas adaptações ao sistema de execução especulativa e distribuição de dados, a fim de adequá-los a ambientes heterogêneos.

3.1 Descrição do Problema

O Hadoop MapReduce, apesar de construído para máquinas de prateleira, considera que o ambiente de execução é *homogêneo*. De acordo com (ZAHARIA et al., 2008; KONWINSKI, 2009), a partir desta, as seguintes outras suposições são feitas pelo *framework*:

- Os nós apresentam uma taxa de trabalho semelhante.
- As tarefas progridem com uma taxa constante no tempo (exceto no caso de máquinas faltosas).
- Não há custo no lançamento de tarefas especulativas para um nó que, caso contrário, estaria ocioso.
- Tarefas tendem a ser lançadas e finalizadas pelos nós no mesmo instante de tempo, logo uma tarefa com baixo progresso indica um *straggler*.
- Tarefas do mesmo tipo (Map ou $Reduce$) requerem aproximadamente o mesmo esforço.

Este comportamento suposto pelo Hadoop se confirma em uma importante parcela dos jobs executados em ambientes homogêneos. Porém, quando o ambiente de execução é heterogêneo, muitas dessas suposições são quebradas. A última, contudo, se mantém em ambientes heterogêneos pois é inerente ao paradigma MapReduce (ZAHARIA et al., 2008).

Quando o ambiente é heterogêneo, tarefas apresentam um progresso díspar em função do poder computacional dos *workers*. Logo, é esperado que máquinas mais lentas apresentem uma taxa de trabalho menor, levando mais tempo para completar as tarefas, apesar de não estarem em um estado defeituoso. Neste caso, o lançamento de tarefas especulativas resulta em um consumo excessivo de recursos, prejudicando os demais nós da grade, especialmente quando vários *jobs* são executados simultaneamente (ZAHARIA et al., 2008; KONWINSKI, 2009).

A distribuição de dados no sistema de arquivos também pode impactar negativamente do desempenho do MapReduce em ambientes heterogêneos. Como previamente comentado, o HDFS busca manter a mesma proporção de uso dos discos do *cluster*. Assim, em determinadas configurações, uma máquina lenta pode possuir a mesma quantidade (ou mais) de dados que as demais. Caso outra máquina receba a tarefa de processar estes dados, é necessário transferi-los pela rede, consumindo banda e tempo.

3.2 Trabalhos Relacionados

O comportamento do Hadoop MapReduce em ambientes heterogêneos já foi previamente analisado e trabalhado por outros pesquisadores.

Em (ZAHARIA et al., 2008; KONWINSKI, 2009) é realizado um estudo que mostra a degradação de performance 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 o mecanismo de tarefas especulativas. 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.

O trabalho realizado por (UCAR et al., 2006), apesar de não relacionar-se diretamente com MapReduce, apresenta contribuições na distribuição de tarefas para processadores heterogêneos. Assim, como os fluxos de execução em um *cluster* Hadoop estão vinculados com os processadores dos nós, as ideias apresentadas no estudo podem ser aproveitadas no ambiente MapReduce.

Em (WANG et al., 2009a,b) é apresentado um simulador para o modelo MapReduce, chamado MRPerf, que utiliza o simulador de redes ns-2 como base para a simulação de rede. No trabalho são expostos os recursos do MapReduce modelados pelo simulador, e são apresentados os resultados de uma validação do mesmo. De acordo com os autores, a replicação de blocos do sistema de arquivos distribuído, e o mecanismos de execução especulativa, não estão presentes no simulador. A falta destes recursos influenciam na simulação do MapReduce sobre ambientes heterogêneos, motivando o desenvolvimento de um novo simulador.

3.3 Modificações Propostas

Com base nos problemas apresentados, advindos da execução do Hadoop em ambientes heterogêneos, este trabalho propõe modificações no comportamento deste sistema, através de adaptações no mecanismo de execução especulativa e distribuição de dados. Ambas modificações são implementadas e avaliadas através de simulação, conforme será demonstrado nos capítulos seguintes.

As modificações, contudo, não abordarão o problema da volatilidade de nós em *Desktop Grids*, i.e. considera-se que as máquinas possuem capacidades de processamento distintas, mas que estarão *disponíveis* durante todo o período de execução. Consequentemente, problemas de tolerância a falhas serão também desconsiderados.

O ambiente sobre o qual as modificações serão avaliadas, portanto, consiste em um *Desktop Grid* empresarial não volátil. Esta configuração permite realizar uma análise inicial das soluções propostas, que atacam mais diretamente a propriedade de heterogeneidade, a qual está presente no ambiente escolhido.

3.3.1 Modificação 1 - Distribuição de Dados

A primeira adaptação proposta, consiste em realizar um balanceamento de carga vinculado à **capacidade de processamento** dos nós, e não à capacidade de *armazenamento*, como é feito na versão atual do Hadoop.

A Figura 3.1 demostra a distribuição de dados realizada no HDFS quando os nós possuem mesma capacidade de armazenamento. Neste caso, mesmo que as máquinas apresentem poderes computacionais distintos, a quantidade de blocos armazenados em cada nó é a mesma. Na figura, a máquina `B` possui metade da capacidade de processamento de `A`, mas ambas armazenam 6 *chunks* cada, totalizando 12 blocos. Desta forma, o nó `A` acabaria de processar seus dados locais antes de `B`, e seria ordenado pelo nó mestre a processar dados armazenados no outro nó, gerando tráfego de rede.

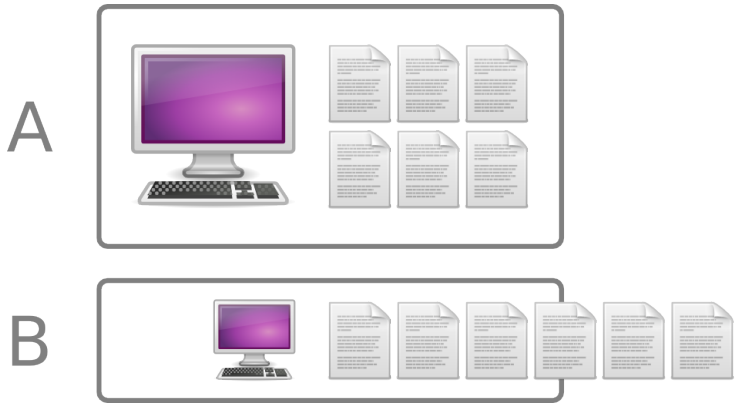

Figura 3.1: Distribuição de dados no HDFS

A modificação proposta pode ser observada na Figura 3.2, onde a capacidade dos nós é considerada na distribuição dos dados. A máquina `A`, por apresentar o dobro do poder computacional de `B`, armazena também o dobro de dados, obviamente totalizando os mesmos 12 blocos.

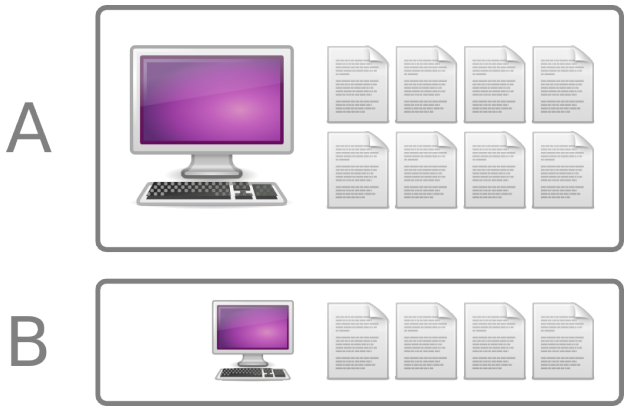

Figura 3.2: Nova distribuição de dados proposta

O objetivo deste balanceamento de carga é explorar ao máximo a localidade dos dados durante a execução dos *jobs* MapReduce, evitando a transferência dos dados em tempo de execução.

Se considerarmos que c_i é a capacidade de processamento de um nó i, C é o somatório de todos os c_i (representando o poder computacional da grade como um todo) e B é a quantidade de blocos da entrada, então a quantidade de blocos que cada nó i deve receber (b_i) é definida através da fórmula

$$b_i = \frac{c_i}{C} \times B. \tag{3.1}$$

Portanto, no exemplo anterior, as máquinas `A` e `B` têm suas quantidades de blocos respectivamente definidas pelas equações

$$b_A = \frac{100}{150} \times 12 = 8$$

$$b_B = \frac{50}{150} \times 12 = 4$$

Com esta distribuição, ambos os nós processam a totalidade de seus dados locais ao mesmo tempo (Figura 3.3). Suponha que a máquina `A` processe 2 blocos por minuto, e `B` apenas 1 bloco por minuto. Neste caso, tanto `A` quanto `B` levariam 4 minutos para processar seus 8 e 4 blocos respectivamente, evitando transferência de informações pela rede.

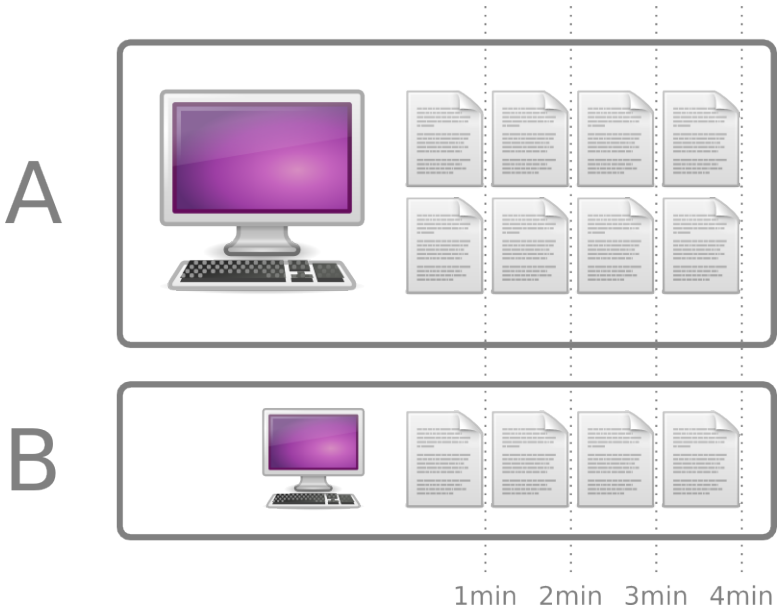

Figura 3.3: Processamento dos dados distribuídos no HDFS

3.3.2 Modificação 2 - Execução Especulativa

Em um ambiente heterogêneo, os nós mais lentos, i.e. com menor capacidade de processamento, serão sempre considerados como *stragglers* pelo Hadoop. Porém, em um ambiente heterogêneo, este progresso díspar é considerado normal, e deve ser previsto pelo escalonador de tarefas. Assim, uma vez que a quantidade de dados tenha sido distribuída de acordo com a poder computacional de cada nó — como proposto na primeira

modificação — espera-se que, mesmo com um progresso díspar, os nós acabem a totalidade de suas tarefas locais em um mesmo intervalo de tempo.

Portanto, a segunda modificação busca identificar *stragglers* através de uma medida proporcional a sua capacidade de processamento, e não em relação à média da grade. Consequentemente, um nó com menor poder computacional será marcado como *straggler* apenas quando estiver abaixo de sua lentidão esperada.

3.4 Considerações Finais

As modificações apresentadas são exemplos de estudos e melhorias que podem ser buscadas no modelo MapReduce. Outros possíveis problemas e situações podem ser estudados, bem como diferentes soluções para os problemas apresentados, conforme os trabalhos relacionados demostram.

Após a identificação de uma solução teórica, uma etapa de validação deve ser executada. O plano para a realização destes testes, e os recursos necessários para tal, são apresentados no capítulo seguinte.

4 METODOLOGIA

Neste capítulo serão apresentadas as decisões tomadas para a realização do estudo proposto neste trabalho, como esta tarefa será realizada, bem como as ferramentas e recursos utilizados neste processo, que representam o ambiente utilizado durante os testes.

4.1 Plano

Atualmente as redes de computadores variam de dezenas a milhares de dispositivos, com vários níveis de heterogeneidade. As modificações propostas ao Hadoop, portanto, devem apresentar resultado positivo em qualquer configuração. A fim de viabilizar testes em uma variedade aceitável de configurações, optou-se pelo desenvolvimento de um **simulador** — referenciado a partir de agora como **MRSG** (acrônimo para *MapReduce-SimGrid*) — para implementar e testar o novo comportamento.

A decisão de implementar um novo simulador decorreu da escassez de variedade, e da falta de alguns recursos das soluções existentes. A indisponibilidade destes recursos impacta de maneira importante na simulação do MapReduce em ambientes heterogêneos. O *MRPerf* (WANG et al., 2009b,a), por exemplo, é um simulador para o modelo MapReduce que utiliza o simulador de redes *ns-2* como base para sua solução. Dentre os recursos mais relevantes para este trabalho, e não implementados pelo MRPerf, é possível citar a replicação de blocos de dados do HDFS — limitada a 1 (uma) réplica por bloco —, e o mecanismo de execução especulativa (WANG et al., 2009b,a). Em ambientes heterogêneos, a falta de réplicas implica a transferência desnecessária de blocos na fase de mapeamento, uma vez que o nó mais rápido acabará suas tarefas locais primeiro, e será forçado a buscar mais dados nos demais nós. A execução especulativa também é de fundamental importância, conforme explicado nos capítulos anteriores.

4.2 Ambiente de Desenvolvimento e Testes

A seguir serão apresentadas as tecnologias e recursos utilizados no desenvolvimento do simulador MRSG, bem como em sua validação em relação ao comportamento real do sistema.

4.2.1 SimGrid

O desenvolvimento de um simulador de grades envolve diversos subproblemas relacionados com a simulação de redes e compartilhamento de recursos de processamento. Esta não é uma tarefa trivial, e por si só implica grande esforço de projeto e validação do simulador. Por esta razão optou-se pela utilização de um simulador de grades existente

como base para o desenvolvimento deste trabalho. O simulador escolhido para este fim foi o *SimGrid*.

O SimGrid (SIMGRID, 2010a) é um conjunto de ferramentas que provê funcionalidades para a simulação de aplicações distribuídas em ambientes heterogêneos. Seu objetivo é justamente facilitar a pesquisa na área de plataformas paralelas e distribuídas.

Diversos pesquisadores, vinculados a uma variedade de universidades de todo o mundo, utilizam, ou já utilizaram, o SimGrid em publicações científicas, reforçando a confiabilidade do sistema (SIMGRID, 2010b).

O SimGrid provê diversas interfaces de programação (ou API, quando abreviado do inglês *Application Programming Interface*) para uma variedade de linguagens. A Figura 4.1 ilustra a arquitetura do simulador, e a relação entre seus componentes.

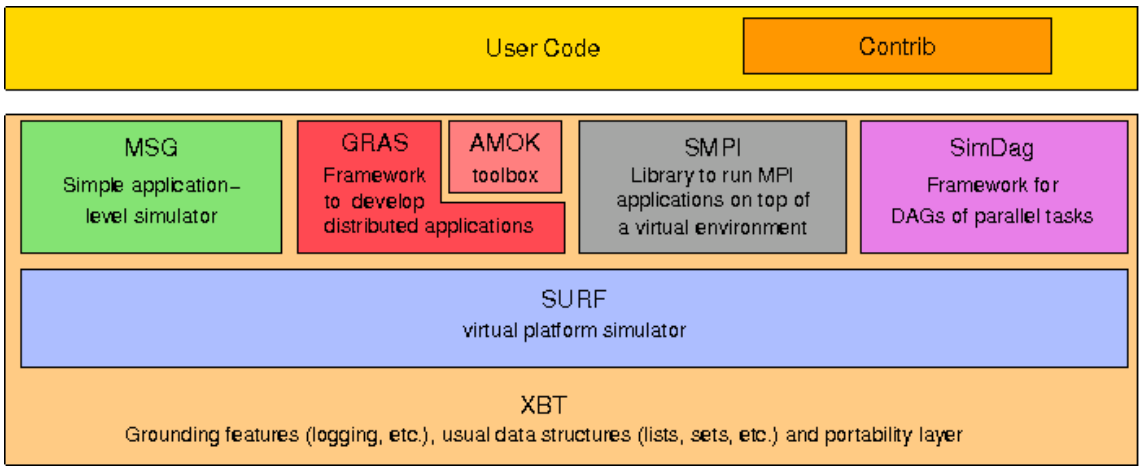

Figura 4.1: Relação entre os componentes do SimGrid (SIMGRID, 2010a)

Elencadas abaixo, estão as APIs disponibilizadas pelo SimGrid, e suas respectivas características.

MSG Voltada ao estudo e comparação de algoritmos e heurísticas, como escalonamento de tarefas em sistemas distribuídos, por exemplo. A interface provê funções simuladas de troca de mensagens e processamento de tarefas. Sua linguagem nativa de programação é o C, porém existem implementações para Lua e Java.

GRAS Utilizada no desenvolvimento de aplicações reais. Além de realizar simulações dos aplicativos desenvolvidos com a API (com a finalidade de testar a implementação), a interface permite a execução em ambientes distribuídos reais.

SMPI Como o nome sugere, esta interface visa o estudo do comportamento de aplicações MPI através de emulação, i.e. utiliza-se código de aplicações MPI reais não modificadas, e o simulador emula a execução paralela dos fluxos. Este ambiente de programação ainda encontra-se em fase de desenvolvimento.

SimDag Utilizado para simular aplicações que seguem um modelo de grafo acíclico direcionado (DAG, abreviado do inglês *Directed acyclic graph*).

Apesar do modelo MapReduce e seu fluxo de dados seguirem um formato DAG, suas implementações, incluindo o Hadoop, proveem uma série de recursos para tolerância a falhas e otimização de desempenho – como a execução especulativa, por exemplo — que

divergem deste modelo. Dito isso, a interface de programação escolhida para a simulação do Hadoop é a MSG, que provê um ambiente simplificado para a análise dos algoritmos de distribuição de dados e escalonamento de tarefas. Através de abstrações no tamanho de tarefas e dados, a API permite também a variação dessas propriedades de forma simples e ágil, facilitando os testes sobre uma variedade de situações e configurações.

4.2.2 Grid'5000

Para validar a simulação do Hadoop, é necessário comparar os resultados de sua execução com aplicações executadas em um ambiente real. Para tal, utilizou-se a infraestrutura da *Grid'5000* como ambiente de validação.

A **Grid'5000** (GRID'5000, 2010a) é uma grade computacional científica, que visa o estudo de sistemas distribuídos e paralelos em larga escala, através de uma plataforma experimental altamente reconfigurável, controlável e monitorável.

Atualmente, os recursos da plataforma estão geograficamente distribuídos em nove locais da França — Bordeaux, Grenoble, Lille, Luxembourg, Lyon, Nancy, Orsay, Rennes, Sophia-Antipolis e Toulouse (ilustrados na Figura 4.2) — abrangendo 17 laboratórios de pesquisa do país.

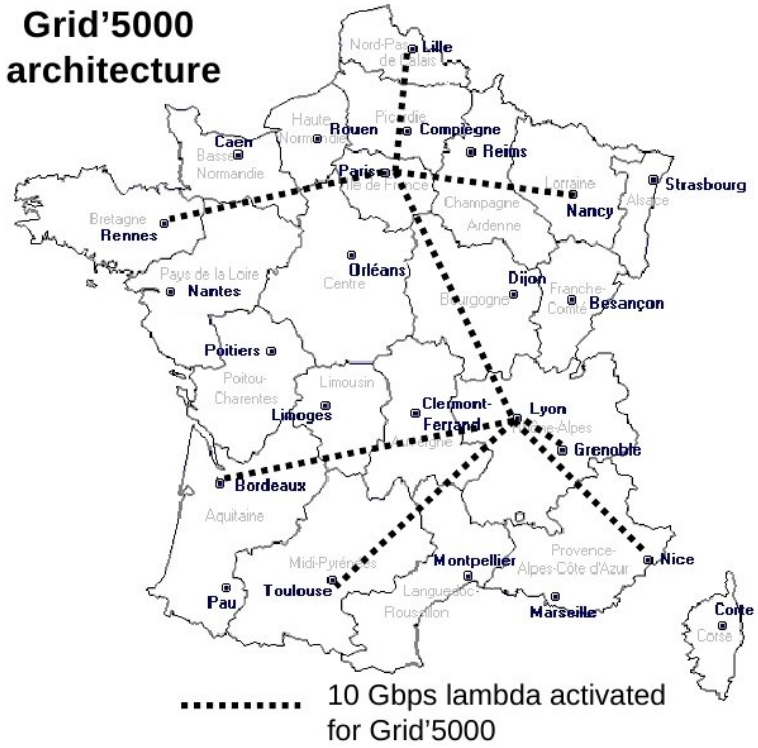

Figura 4.2: Distribuição geográfica dos *clusters* na Grid'5000 (GRID'5000, 2010a)

Para a configuração e instalação de ambientes de testes na Grid'5000, existe uma variedades de ferramentas disponíveis. As ferramentas utilizadas nos testes deste trabalho compreendem o OAR e o Kadeploy.

A ferramenta OAR é utilizada para gerenciar os recursos dos *clusters*, proporcionando uma maneira de reservar quantidades arbitrárias de nós da grade, por períodos de tempo especificados pelo usuário.

O Kadeploy, por sua vez, proporciona a possibilidade de clonar a instalação de sistemas operacionais e aplicativos, reinstalá-los em outros nós, e configurar cada imagem através do uso de *scripts*.

Hadoop na Grid'5000

O Hadoop pode ser instalado em 3 modos: *Standalone*, *Pseudo-Distributed* e *Fully-Distributed*. O primeira é útil para testar a aplicação e depurar o código, e roda como um único processo Java, em uma única máquina. O modo *Pseudo-Distributed*, assim como no *Standalone*, é executado em apenas uma máquina, porém cada *daemon* do Hadoop roda em um processo distinto. Já o modo *Fully-Distributed* é utilizado em sistemas de produção, realmente distribuídos.

A instalação *Standalone* é bastante simples, e requer apenas o Java 1.6 (preferivelmente da Sun). Para instalar, basta descompactar o pacote com os fontes, e editar o arquivo texto `conf/hadoop-env.sh` (no diretório de instalação do Hadoop) indicando a localização de sua instalação Java na variável `JAVA_HOME`. A partir deste ponto já é possível escrever e testar aplicativos MapReduce.

A instalação dos modos *Pseudo-Distributed* e *Fully-Distributed* requerem maiores configurações. Como requisitos, é ainda necessário instalar um serviço de *SSH*, e configurar o HDFS. As configurações necessárias envolvem definir qual máquina será o *master* no MapReduce e no HDFS, e definir os *workers*.

A Grid'5000 é uma grade real, logo, para executar aplicativos MapReduce, é necessário realizar uma instalação *Fully-Distributed* do Hadoop. Para facilitar este processo foi utilizado um *script* de configuração, que automatiza a configuração dos nós. O *script*, bem como instruções detalhadas de utilização, podem ser encontrados no tutorial presente na *Wiki* da Grid'5000 (GRID'5000, 2010b). O processo realizado para os testes deste trabalho é apresentado abaixo.

Primeiramente é necessário conectar-se via SSH a um local (`SITE`) da Grid'5000 no qual se possua um usuário válido (`USER`).

```
ssh USER@access.SITE.grid5000.fr
```

Em seguida deve-se acessar a máquina `frontend`, de onde é possível reservar recursos para o experimento.

```
ssh frontend
```

Uma vez conectado ao `frontend`, utiliza-se o comando `oarsub` para realizar a reserva, indicando a quantidade *hosts* que deseja-se alocar (`nodes=NUM_HOSTS`), e por quanto tempo (`walltime=HH:MM:SS`). A quantidade de nós especificados aqui, será a quantidade de hosts que o Hadoop usurá para distribuir a aplicação (este valor está relacionado com a quantidade de reduções explicitados no código).

```
oarsub -I -t deploy -l nodes=NUM_HOSTS,walltime=HH:MM:SS
```

Se a alocação for realizada com sucesso, e a conexão com o job for estabelecida, é possível partir para a instalação do Hadoop. Para facilitar, será utilizada uma imagem pronta da instalação, e um script de configuração que definirá o nó mestre e demais propriedades relevantes. A instalção das máquinas virtuais na Grid'5000 dá-se através do comando `kadeploy`.

```
kadeploy3 -a ~/lenny-x64-nfs-hadoop.dsc3 -f \
OAR_FILE_NODES -k ~/.ssh/id_dsa.pub -s ~/config.sh
```

Uma vez instalado o ambiente, deve-se conectar ao nó mestre do Hadoop.

```
ssh USER@MASTER.SITE.grid5000.fr
```

Em seguida é necessário copiar o arquivo de entrada para o novo sistema HDFS instalado. Para tal utiliza-se a ferramenta do hadoop, com a opção `dfs` para manipular o sistema de arquivos.

```
/opt/hadoop/bin/hadoop dfs -copyFromLocal ~/data input
```

Por fim basta executar a aplição, informando a localização do arquivo de entrada e o diretório onde a saída deve ser gravada. Ambos os caminhos referem-se ao sistema de arquivos distribuído.

```
/opt/hadoop/bin/hadoop jar ~/tracefilter.jar \
br.ufrgs.TraceFilter input output
```

4.3 Considerações Finais

Este capítulo apresentou o ambiente de testes e os componentes utilizados na implementação e validação do MRSG. Uma descrição mais detalhada do simulador e suas funcionalidades é exposta no capítulo 5.

5 ESPECIFICAÇÃO DO SIMULADOR

Neste capítulo serão expostas as decisões na modelagem do MRSG, suas características e limitações, sua entrada e saída, bem como uma breve comparação com os recursos do MRPerf.

5.1 Objetivos do Simulador

O MRSG tem o objetivo de facilitar a pesquisa sobre o comportamento do MapReduce e possíveis modificações nessa tecnologia. Para tal, o simulador busca proporcionar:

- Uma maneira simplificada de traduzir algoritmos teóricos (escalonamento de tarefas, distribuição de dados, etc) para código executável, facilitando a análise desses algoritmos.
- Facilidade na variação e teste de configurações do sistema.
- A possibilidade de validar algoritmos em larga escala, sendo que não é necessária uma infraestrutura real.

Abaixo estão listados os recursos considerados, pelo autor, os mais importantes na simulação do Hadoop em ambientes heterogêneos, e que devem estar presentes no MRSG. Assim, pretende-se simular:

- As fases de mapeamento, cópia e redução, modelando todas as comunicações de dados da aplicação.
- O compartilhamento de recursos de rede e processamento entre os nós da grade.
- A replicação de blocos do HDFS.
- O mecanismo de execução especulativa.

5.2 Limitação do Estado Atual

Devido à inviabilidade de implementar todos os recursos desejados para o MRSG, optou-se pela exclusão de algumas propriedades de baixo impacto para a simulação dos novos algoritmos de distribuição de dados e escalonamento de tarefas. Logo, a versão atual do sistema desenvolvido não simula:

- Volatilidade e falha dos nós.

- Configuração de *racks* em *clusters*.

As propriedades desconsideradas podem ser ignoradas no contexto deste trabalho, pois o ambiente de *Desktop Grid* empresarial estudado não apresenta as características de um *cluster* convencional, como unidades de armazenamento compartilhadas (*storages*), por exemplo. Como também não serão abordados ambientes faltosos ou voláteis, onde os nós da grade variam sua disponibilidade, a primeira propriedade não será simulada. Vale ressaltar que o modelo atual do MapReduce também não considera volatilidade, i.e. períodos de indisponibilidade são tratados como falhas.

Assim, para o foco deste trabalho, o MRSG implementa todos os recursos necessários para avaliar o comportamento do MapReduce em ambientes heterogêneos quando submetido às modificações propostas na seção 3.3.

5.3 Uso do Simulador

O simulador MRSG é um executável de linha de comando (testado em plataforma Linux, x86 de 32 e 64 bits), que recebe como parâmetros um arquivo descritor de plataforma, e outro de aplicação.

O descritor de plataforma (*platform file*) é um arquivo XML que descreve a topologia da rede e a capacidade dos recursos envolvidos, incluindo o poder computacional dos nós da grade, e latência e banda das conexões (*links*, em inglês) que interligam os nós. Este arquivo é um padrão do SimGrid, e possui uma forma bem defina de acordo com uma DTD (*Document Type Definition*). Um exemplo pode ser observado no Anexo B.

O arquivo de aplicação (*deploy file*), por sua vez, indica a função de cada nó e identifica o nó mestre. É neste arquivo que o usuário do simulador indica as configurações de seu ambiente MapReduce, através de argumentos passados ao nó mestre no XML, como demostrado no Anexo C.

Como resultado de uma simulação no MRSG, além do progresso apresentado em tempo real no terminal, o usuário pode analisar arquivos de *log* criados pelo simulador, os quais proveem informações relativas à localização de blocos do HDFS e à quantidade de tarefas processadas por cada nó.

5.4 Comparação

A Tabela 5.1 apresenta uma breve comparação entre o MRPerf e o novo simulador MRSG, destacando alguns dos recursos mais importantes na simulação de um modelo MapReduce, presentes nos respectivos sistemas.

Os recurso não implementados no MRSG podem ser posteriormente incluídos ao simulador, para auxiliar em pesquisas com o MapReduce. Nenhuma restrição tecnológica existe em relação às bibliotecas utilizadas no desenvolvimento do simulador.

5.5 Considerações Finais

Uma vez apresentada a especificação do simulador desenvolvido durante o trabalho, o capítulo seguinte analisará o simulador, avaliando e validando seu comportamento em comparação a execução de um sistema MapReduce real. Após esta validação, as soluções para os problemas encontrados no MapReduce em ambientes heterogêneos serão implementadas simulador, e também terão seu desempenho avaliado.

Propriedade	**MRPerf**	**MRSG**
Compartilhamento de recursos (rede, processamento)	**Sim**	**Sim**
Adequado a ambientes homogêneos e heterogêneos	Não	**Sim**
Permite a configuração de *racks*	**Sim**	Não
Replicação de blocos do HDFS	Não	**Sim**
Mecanismo de execução especulativa	Não	**Sim**
Simula etapas internas das tarefas de forma independente	**Sim**	Não
Várias unidades de armazenamento por nó	Não	Não

Tabela 5.1: Comparação de recursos com o MRPerf

6 RESULTADOS

Na seção seguinte serão apresentados os resultados da validação do MRSG, comparando as simulações realizadas com execuções reais na grid'5000, e posteriormente serão apresentados os resultados obtidos com a aplicação das novas políticas de distribuição de dados e execução especulativa, conforme proposto na seção 3.3.

6.1 Validação do Simulador

A validação do simulador foi realizada através da comparação entre execuções reais do Hadoop na Grid'5000 e simulações com parâmetros que adequados à execução real.

6.1.1 Descrição do Aplicativo Utilizado na Validação

O aplicativo desenvolvido para os testes de validação consiste basicamente em um filtro de *log*. Mais especificamente, deseja-se ler um grande arquivo, com registros de mudança disponibilidade das máquinas de um ambiente de computação voluntária, e calcular a média de disponibilidade de um certo conjunto de máquinas, que serão escolhidas através de alguns pré-requisitos.

O arquivo de *log* que será utilizado nos testes, contém informações coletadas pelo projeto BOINC (BOINC, 2010) — uma importante plataforma para computação voluntária — e é formado por diversos registros de disponibilidade dos nós que constituem este sistema de computação voluntária. Os dados são agrupados pelo ID dos nós, i.e. todos os registros de um nó aparacem de forma consecutiva e ordenados por horário. Cada linha do *log* contém os seguintes campos:

1. Número do registro para um determinado nó (começando em zero);
2. Identificador do nó;
3. Tipo do evento (0 ou 1, indicando a disponibilidade);
4. Início do evento;
5. Fim do evento.

Cada campo é separado por um caractere de tabulação. Um exemplo de conjunto de registros deste arquivo pode ser representado pelas seguintes linhas:

```
0   100416828   1   1211524407.906   1211524417.922
1   100416828   0   1211524417.922   1211524440.312
2   100416828   1   1211524440.312   1211615915.156
```

A partir deste exemplo é possível destacar que:

- Todos os registros são referentes à mesma máquina, pois o ID é 100416828 nas três linhas.
- São os 3 primeiros registros desta máquina, pois o primeiro campo varia de 0 a 2.
- A primeira e terceira linha são registros de disponibilidade, pois contém o valor 1 no tipo do evento (terceiro campo), enquanto a segunda linha é um registro de indisponibilidade.
- Os campos 4 e 5 representam, respectivamente, o tempo de início e fim de cada evento.

No total, o arquivo utilizado contém 8.8GB de tamanho, contendo mais de 1.5 ano de registros, para 226,208 *hosts*. A filtragem realizada em cima dos registros, consiste em selecionar apenas máquinas que participaram pelo menos 300 dias do projeto, e que possuem disponibilidade média de no máximo 4000 segundos (pouco mais de 1 hora).

Ao final da execução, espera-se obter um arquivo contendo todos os *hosts* que passaram pelo filtro, e seus respectivos tempos médios de disponibilidade e os horários de início e fim dos seus registros. Ou seja, apenas uma linha por máquina válida. Considerando o exemplo de *log* anterior, caso o *host* passasse pelo filtro, o arquivo resultante conteria uma linha:

```
100416828   45742   1211524407   1211615915
```

O código fonte completo desta aplicação para o Hadoop MapReduce pode ser observado no Anexo D.

6.1.2 Configuração dos Jobs e Resultados

No total foram realizados testes de validação com 16, 20, 32 e 200 núcleos, de acordo com a disponibilidade encontrada na grid'5000. São apresentados a seguir os resultados obtidos para os testes com 20 e 200 núcleos. Os testes com 16 e 32 núcleos obtiveram resultados semelhantes ao teste de 20, pois as configurações utilizadas eram semelhantes, como por exemplo, a quantidade de reduções, que era igual à quantidade de núcleos nos três casos.

A tabela 6.1 apresenta a configuração para o teste com 20 núcleos em maiores detalhes.

Parâmetro	**Valor/Descrição**
Processador	AMD Opteron 250 / 2.4GHz / 1MB / 400MHz
Memória	2 GB
Quantidade de núcleos	20
Tamanho da entrada	8,8 GB
Quantidade de reduces	20

Tabela 6.1: Parâmetros da validação com 20 núcleos

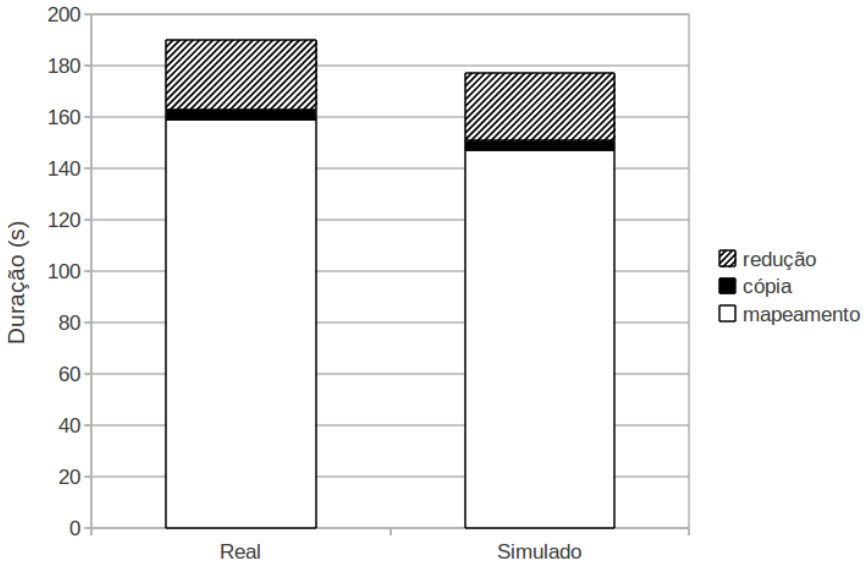

Figura 6.1: Validação do MRSG com 20 núcleos

Estas configurações foram reproduzidas no simulador, e a comparação do tempo de execução pode ser observada na figura 6.1. Cada uma das fases (mapeamento, cópia e redução) é comparada e apresentada na figura 6.2.

Através dos gráficos 6.1 e 6.2, percebe-se uma importante proximidade entre os tempos de execução real e simulado, com proporções adequadas de duração para todas as fases do job. Estes resultados foram observados também nas execuções com 16 e 32 núcleos.

Na validação com 200 núcleos, a grande diferença na configuração está relacionada com a quantidade de tarefas de redução, conforme a tabela 6.2. Enquanto os testes anteriores foram realizados com uma quantidade de reduções igual à quantidade de núcleos, nesta comparação foi utilizada apenas uma tarefa de redução. Este teste foi realizado para verificar como o Hadoop e o simulador comportam-se em situações peculiares.

Parâmetro	**Valor/Descrição**
Processador	AMD Opteron 2218 / 2.6 GHz / 2 MB L2 cache / 800 MHz
Memória	4 GB (4x512 MB + 2x1024 MB)
Quantidade de núcleos	200
Tamanho da entrada	8,8 GB
Quantidade de reduces	1

Tabela 6.2: Parâmetros da validação com 200 núcleos

É possível observar (figura 6.3) que a diferença entre os tempos totais dos *jobs* real e simulado, neste caso, é mais acentuada que nos testes anteriores. Para uma melhor comparação, a figura 6.4 apresenta a diferença entre cada fase do MapReduce, apontando a cópia como a causa desta discrepância nos tempos de execução.

Este comportamento pode ser explicado por diversos fatores que ocorrem no ambiente real utilizado (Grid'5000), e que não foram simulados no MRSG. Como exemplo é possível citar a interferência de outras pesquisas que são executadas na grade, e que, apesar de

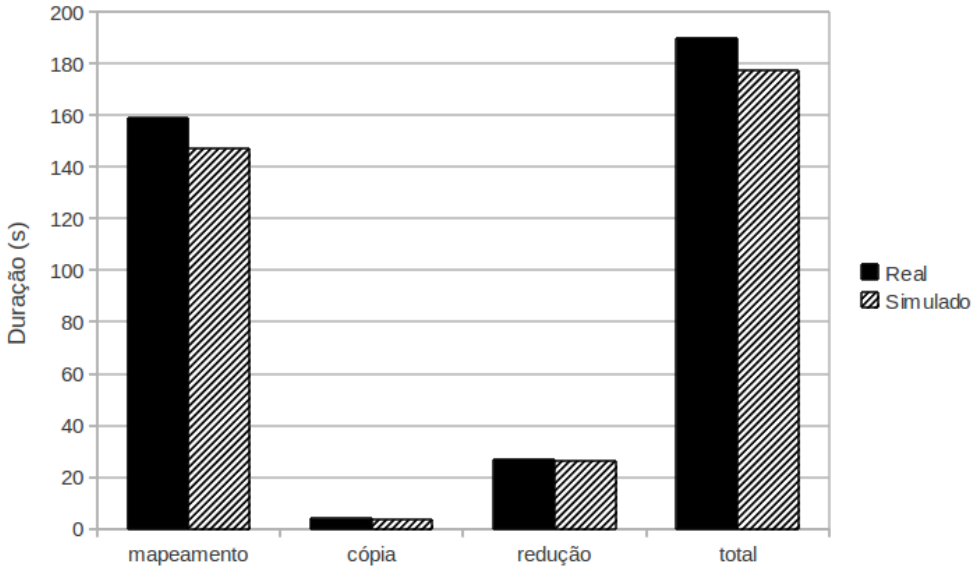

Figura 6.2: Validação do MRSG com 20 núcleos (fases)

não compartilharem recursos de processamento com os nós testados, compartilham recursos de armazenamento e rede; e a criação de apenas uma tarefa de redução, que implica uma única máquina ter que copiar todos os resultados intermediários, deixando a conexão mais sensível a interferências externas, uma vez que seu link estava sobrecarregado.

Apesar desta diferença na cópia, o simulador apresentou o comportamento esperado, indicando um acréscimo razoável no tempo de cópia. Assim, com os resultados destas comparações, pode-se concluir que, apesar do simulador ser passível de melhorias, seu comportamento é suficientemente próximo ao comportamento real do Hadoop para testar e analisar o comportamento de algoritmos teóricos.

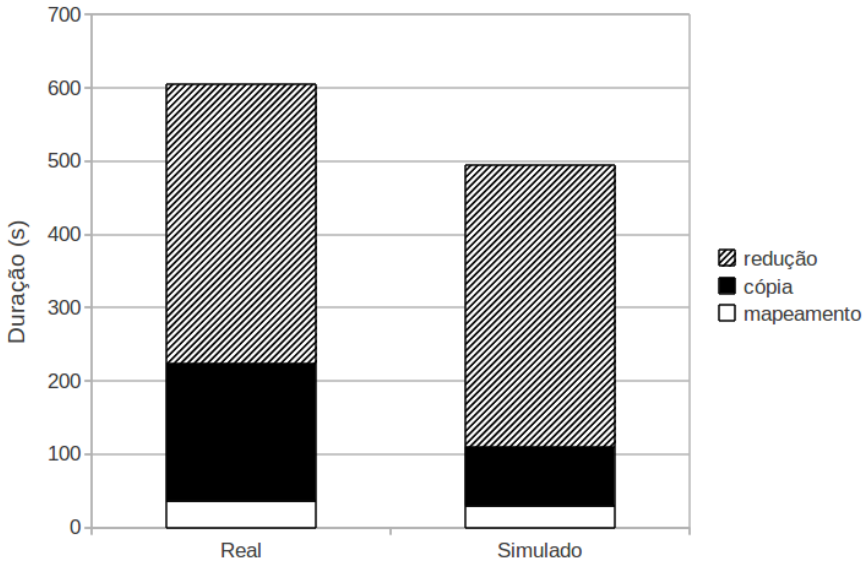

Figura 6.3: Validação do MRSG com 200 núcleos

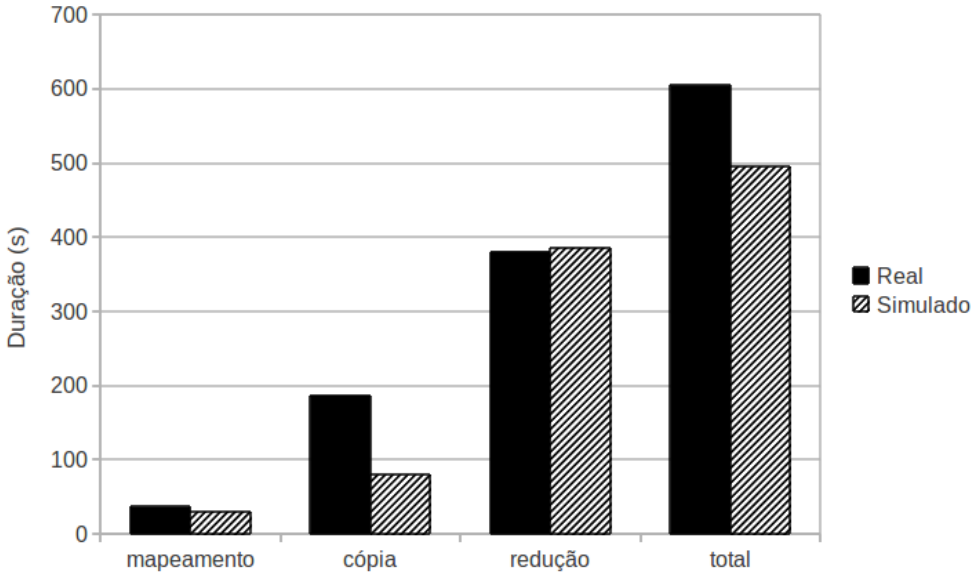

Figura 6.4: Validação do MRSG com 200 núcleos (fases)

6.2 Adaptação a Ambientes Heterogêneos

Uma vez que a validação do simulador MRSG foi realizada, e resultados satisfatórios puderam ser observados, as adaptações propostas na seção 3.3 foram desenvolvidas com o simulador. Os resultados obtidos com as simulações são apresentados a seguir, e demostram os ganhos de desempenho obtidos pelas novas políticas de distribuição de dados e execução especulativa, em um ambiente heterogêneo.

Os testes foram executados simulando ambientes com 100, 200, 300 e 400 núcleos. O poder computacional desses recursos foi distribuído uniformemente, representando processadores de 800MHz a 2,4GHz. Como entrada de dados, foram definidos 10 blocos por núcleo, i.e. conforme a quantidade de núcleos da simulação aumentava, o mesmo acontecia com a entrada. A configuração de rede foi mantida como uma *Switched Ethernet* de 100Mb/s.

Os resultados foram analisados considerando a quantidade de tarefas geradas que necessitam de transferência de dados. Desta forma é possível avaliar custos de desempenho para diversas situações, com tempos de latência e larguras de banda variáveis.

A figura 6.5 mostra a quantidade de tarefas especulativas geradas pelo comportamento original do Hadoop, onde é possível observar que uma grande quantidade de tarefas auxiliares são lançadas pelo escalonador, em ambas as fases de mapeamento e redução. Este comportamento prejudica o próprio *job*, pois possíveis *slots* de redução são ocupados por mapeamentos especulativos, e *jobs* concorrentes que sofrem com o uso desses recursos de processamento e rede.

Em contrapartida, como esperado, o modificação proposta no mecanismo de execução especulativa não gerou nenhuma tarefa deste tipo. Como o trabalho não considera falhas, e as máquinas sempre seguem a capacidade de processamento descrita no arquivo de plataforma, os nós nunca ficam abaixo de sua média. Esta modificação, contudo, só faz sentido quando os dados são distribuídos conforme explicado na seção 3.3.

O resultado do balanceamento de carga na distribuição de dados pode, então, ser analisado pelo gráfico da figura 6.6, que apresenta a quantidade de mapeamentos sobre blocos não locais para o modelo original do Hadoop e para a modificação proposta.

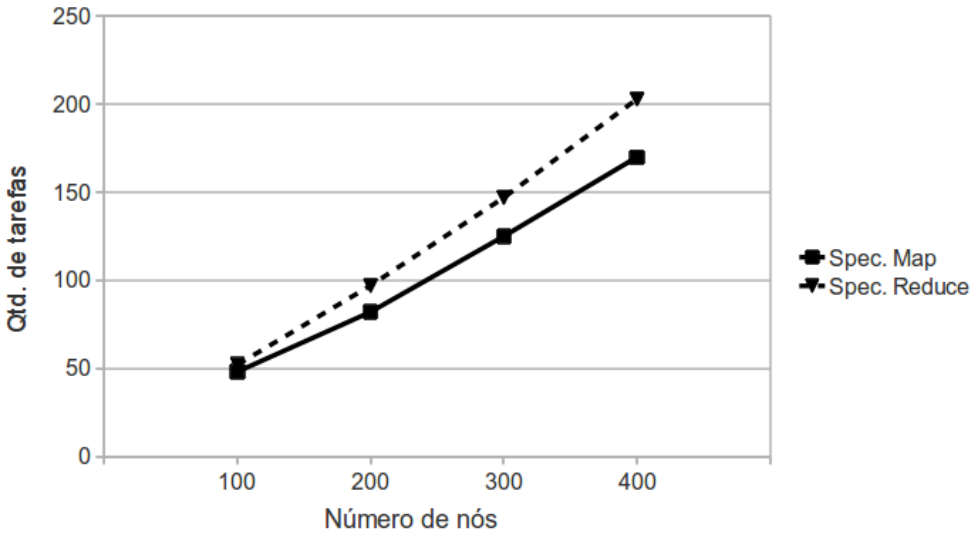

Figura 6.5: Quantidade de tarefas especulativas no modelo original

Através deste resultado fica evidente o ganho em termos de blocos transferidos entre os nós da grade, que é muito inferior ao modelo original de distribuição do Hadoop, o qual apresenta um crescimento linear de transferências em relação ao tamanho da grade.

Estes resultados preliminares obtidos com o simulador, apesar de apresentarem uma certa margem de erro, conseguem demonstrar claramente o impacto que novas políticas de distribuição de dados e escalonamento de tarefas podem causar num *framework* complexo como Hadoop MapReduce, e como a abordagem por simulação facilita a tradução de um algoritmo teórico para testes executáveis que auxiliam no desenvolvimento de novas ideias.

6.3 Considerações Finais

Os resultados obtidos na validação com 16, 20 e 32 núcleos, onde a quantidade de reduções era igual à quantidade de núcleos, obteve-se uma proximidade bastante importante com a execução real. A simulação ficou em torno de 6,75% abaixo do tempo total do *job* observado na Grid'5000. Este resultado parece satisfatório quando comparado ao MRPerf, que apresentou resultados semelhantes em sua validação, que também utiliza uma quantidade de reduções equivalente ao número de núcleos, com diferenças de 3,42% no mapeamento e 19,32% na redução, para um *cluster* com um único rack (WANG et al., 2009a,b).

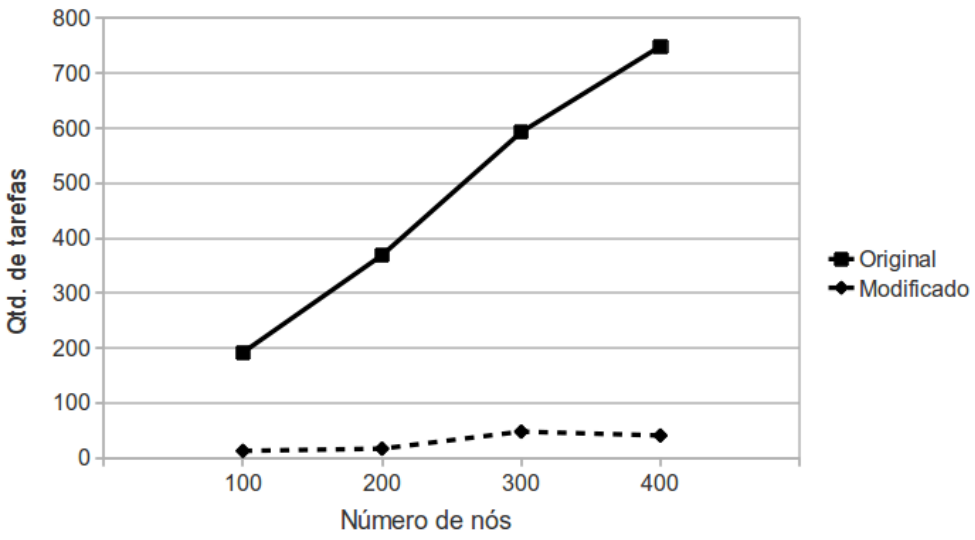

Figura 6.6: Quantidade de mapeamentos não locais

7 CONCLUSÃO

Neste trabalho foi apresentado um estudo sobre o funcionamento interno do Hadoop MapReduce, necessário para compreender todas as etapas envolvidas na execução deste sistema, e que possibilitou propor melhorias sobre a plataforma.

Foi demostrado, também, que as adaptações propostas à distribuição de dados e ao escalonamento de tarefas especulativas apresentam impacto positivo na execução do MapReduce, reduzindo a transferência de dados e consumindo menos recursos. Estes novos algoritmos de distribuição de dados e escalonamento puderam ser testados de maneira mais eficiente devido ao simulador desenvolvido durante o trabalho.

O simulador MRSG foi satisfatoriamente testado e validado contra execuções reais do MapReduce e, apesar de ser passível de melhorias e aprimoramentos, foi suficiente para a pesquisa realizada neste trabalho.

A partir da pesquisa realizada, e destes resultados obtidos, é possível citar como possíveis trabalhos futuros na área: introduzir, além da heterogeneidade, a volatilidade ao ambiente para melhor caracterizar *Desktop Grids*; expandir o desenvolvimento do simulador através do aprimoramento dos recursos existentes e da adição dos recursos não implementados, sendo que um simulador para a tecnologia auxilia em pesquisas na área.

O estudo realizado, portanto, proporciona uma introdução ao MapReduce, e levanta questões interessantes sobre as possíveis adaptações às quais o modelo está sujeito, que não estão restritas às propostas deste trabalho. Assim, espera-se que tenha sido atingido o objetivo de incentivar a pesquisa desta tecnologia, a qual é utilizada em ambientes de produção de grandes corporações, e que representa as inovações que buscamos alcançar em nossa área de atuação.

REFERÊNCIAS

APACHE. **Welcome! - The Apache Software Foundation**. Disponível em: <http://www.apache.org/>. Acesso em: abril 2010.

BOINC. **BOINC**. Disponível em: <http://boinc.berkeley.edu/>. Acesso em: Maio 2010.

DEAN, J.; GHEMAWAT, S. MapReduce: simplified data processing on large clusters. **Commun. ACM**, New York, NY, USA, v.51, n.1, p.107–113, 2008.

DEAN, J.; GHEMAWAT, S. MapReduce: a flexible data processing tool. **Commun. ACM**, New York, NY, USA, v.53, n.1, p.72–77, 2010.

DUBREUIL, M.; GAGNÉ, C.; PARIZEAU, M. Analysis of a master-slave architecture for distributed evolutionary computations. **Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on**, [S.l.], v.36, n.1, p.229–235, 2006.

FOSTER, I.; KESSELMAN, C. **The Grid 2**: blueprint for a new computing infrastructure. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

GANTZ, J.; REINSEL, D. **The Digital Universe Decade – Are You Ready?** [S.l.]: IDC, 2010. White Paper.

GHEMAWAT, S.; GOBIOFF, H.; LEUNG, S.-T. The Google file system. In: SOSP '03: PROCEEDINGS OF THE NINETEENTH ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, 2003, New York, NY, USA. **Anais...** ACM, 2003. p.29–43.

GRID'5000. **Grid5000**: home - grid5000. Disponível em: <https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home>. Acesso em: abril 2010.

GRID'5000. **Run Hadoop On Grid'5000**. Disponível em: <https://www.grid5000.fr/mediawiki/index.php/Run_Hadoop_On_Grid'5000>. Acesso em: abril 2010.

HADOOP. **Welcome to Apache Hadoop!** Disponível em: <http://hadoop.apache.org/>. Acesso em: abril 2010.

HADOOP. **HDFS Architecture**. Disponível em: <http://hadoop.apache.org/common/docs/current/hdfs_design.html>. Acesso em: abril 2010.

HADOOP. **HDFS - Hadoop Wiki**. Disponível em: <http://wiki.apache.org/hadoop/HDFS>. Acesso em: maio 2010.

HADOOP. **Map/Reduce Tutorial**. Disponível em: <http://hadoop.apache.org/common/docs/r0.20.1/mapred_tutorial.html>. Acesso em: junho 2010.

KONDO, D.; FEDAK, G.; CAPPELLO, F.; CHIEN, A. A.; CASANOVA, H. Characterizing resource availability in enterprise desktop grids. **Future Generation Computer Systems**, [S.l.], v.23, n.7, p.888–903, 2007.

KONWINSKI, A. **Improving MapReduce Performance in Heterogeneous Environments**. 2009. Dissertação (Mestrado em Ciência da Computação) — EECS Department, University of California, Berkeley. (UCB/EECS-2009-183).

MURUGESAN, S. Harnessing Green IT: principles and practices. **IT Professional**, [S.l.], v.10, n.1, p.24–33, jan.-feb. 2008.

MUTKA, M. W.; LIVNY, M. Profiling Workstations' Available Capacity for Remote Execution. In: PERFORMANCE '87: PROCEEDINGS OF THE 12TH IFIP WG 7.3 INTERNATIONAL SYMPOSIUM ON COMPUTER PERFORMANCE MODELLING, MEASUREMENT AND EVALUATION, 1988, Amsterdam, The Netherlands, The Netherlands. **Anais...** North-Holland Publishing Co., 1988. p.529–544.

SIMGRID. **SimGrid Home Page**. Disponível em: <http://simgrid.gforge.inria.fr/>. Acesso em: agosto 2010.

SIMGRID. **SimGrid**: people around simgrid. Disponível em: <http://simgrid.gforge.inria.fr/doc/people.html>. Acesso em: agosto 2010.

TIAN, C.; ZHOU, H.; HE, Y.; ZHA, L. A Dynamic MapReduce Scheduler for Heterogeneous Workloads. In: GCC '09: PROCEEDINGS OF THE 2009 EIGHTH INTERNATIONAL CONFERENCE ON GRID AND COOPERATIVE COMPUTING, 2009, Washington, DC, USA. **Anais...** IEEE Computer Society, 2009. p.218–224.

TOTH, D. M. **Improving the productivity of volunteer computing**. 2008. Tese (Doutorado em Ciência da Computação) — Bucknell University.

UCAR, B.; AYKANAT, C.; KAYA, K.; IKINCI, M. Task assignment in heterogeneous computing systems. **J. Parallel Distrib. Comput.**, Orlando, FL, USA, v.66, n.1, p.32–46, 2006.

VENNER, J. **Pro Hadoop**. Berkely, CA, USA: Apress, 2009.

WANG, G.; BUTT, A. R.; PANDEY, P.; GUPTA, K. 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.

WANG, G.; BUTT, A. R.; PANDEY, P.; GUPTA, K. A simulation approach to evaluating design decisions in MapReduce setups. In: 2009. **Anais...** [S.l.: s.n.], 2009. p.1–11.

WHITE, T. **Hadoop**: the definitive guide. [S.l.]: O'Reilly Media, Inc., 2009.

ZAHARIA, M.; KONWINSKI, A.; JOSEPH, A. D.; KATZ, R. H.; STOICA, I. **Improving MapReduce Performance in Heterogeneous Environments**. [S.l.: s.n.], 2008. (UCB/EECS-2008-99).

ANEXO A EXEMPLO DE PROGRAMAÇÃO NO HADOOP

O código fonte deste anexo apresenta um exemplo de implementação da aplicação para contagem de ocorrências de palavras em arquivos texto utilizando o Hadoop Map-Reduce. Fonte: (HADOOP, 2010d).

```
package org.myorg;

import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

public class WordCount {

  public static class Map extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {
      private final static IntWritable one = new IntWritable(1);
      private Text word = new Text();

      public void map(LongWritable key, Text value,
          OutputCollector<Text, IntWritable> output, Reporter reporter)
        throws IOException {
          String line = value.toString();
          StringTokenizer tokenizer = new StringTokenizer(line);
          while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            output.collect(word, one);
          }
        }
    }

  public static class Reduce extends MapReduceBase
    implements Reducer<Text, IntWritable, Text, IntWritable> {
      public void reduce(Text key, Iterator<IntWritable> values,
          OutputCollector<Text, IntWritable> output, Reporter reporter)
        throws IOException {
          int sum = 0;
          while (values.hasNext()) {
            sum += values.next().get();
          }
          output.collect(key, new IntWritable(sum));
```

```
        }
    }

  public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");

    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);

    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);

    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);

    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));

    JobClient.runJob(conf);
  }
}
```

ANEXO B ARQUIVO DESCRITOR DE PLATAFORMA

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "simgrid.dtd">
<platform version="2">
  <host id="Host␣0" power="2.4853517588906598E8" />
  <host id="Host␣1" power="5.1901508236964965E8" />
  <host id="Host␣2" power="8.250212734076045E8" />

  <link id="l0" bandwidth="1.25E8" latency="1.0E-4" />
  <link id="l1" bandwidth="1.25E8" latency="1.0E-4" />

  <route src="Host␣0" dst="Host␣1">
    <link:ctn id="l0"/>
  </route>

  <route src="Host␣0" dst="Host␣2">
    <link:ctn id="l1"/>
  </route>

  <route src="Host␣1" dst="Host␣0">
    <link:ctn id="l0"/>
  </route>

  <route src="Host␣1" dst="Host␣2">
    <link:ctn id="l0"/>
    <link:ctn id="l1"/>
  </route>

  <route src="Host␣2" dst="Host␣0">
    <link:ctn id="l1"/>
  </route>

  <route src="Host␣2" dst="Host␣1">
    <link:ctn id="l1"/>
    <link:ctn id="l0"/>
  </route>

</platform>
```

ANEXO C ARQUIVO DESCRITOR DE APLICAÇÃO

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "simgrid.dtd">
<platform version="2">
        <process host="Host␣0" function="master">
                <argument value="2"/>   <!-- reduces -->
                <argument value="64"/>  <!-- chunk size (MB) -->
                <argument value="10"/>  <!-- chunk count -->
                <argument value="3"/>   <!-- chunk replicas -->
                <argument value="50"/>  <!-- map output size (%) -->
                <argument value="250"/> <!-- map cpu cost per byte -->
                <argument value="500"/> <!-- reduce cpu cost per byte -->
        </process>
        <process host="Host␣1" function="worker"/>
        <process host="Host␣2" function="worker"/>
</platform>
```

ANEXO D CÓDIGO FONTE DA VALIDAÇÃO DO SIMULADOR

```
package br.ufrgs;

import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;

public class TraceFilter {

  public static class Map extends MapReduceBase
    implements Mapper<LongWritable, Text, LongWritable, Text> {

      private LongWritable k = new LongWritable();
      private Text v = new Text();

      /* Implementacao da funcao de mapeamento. */
      public void map(LongWritable key, Text value,
          OutputCollector<LongWritable, Text> output, Reporter reporter
          ) throws IOException {

        /* Separa os campos da linha em um array. */
        String[] fields = value.toString().split("\\s");

        /* Pega o ID da maquina do segundo campo da linha. */
        Long machine = new Long(fields[1]);

        /* Verifica se a linha representa um evento de disponibilidade.
           Os registros de indisponibilidade sao ignorados. */
        if (fields[2].equals("1")) {
          /* Emite um par chave/valor intermediario, contendo o ID da
             maquina (chave) e os tempos de inicio e fim do evento
             concatenados em uma string (valor). */
          k.set(machine);
          v.set(fields[3] + ":" + fields[4]);
          output.collect(k, v);
        }
      }
    }

  public static class Reduce extends MapReduceBase
```

```
  implements Reducer<LongWritable, Text, LongWritable, Text> {

    /* Implementacao da funcao de reducao. */
    public void reduce(LongWritable key, Iterator<Text> values,
        OutputCollector<LongWritable, Text> output, Reporter reporter
        ) throws IOException {

      Long sum = new Long(0);
      Long count = new Long(0);
      Long traceStart = new Long(Long.MAX_VALUE);
      Long traceEnd = new Long(0);
      Long start = new Long(0);
      Long end = new Long(0);

      /* Percorre todos os registros de uma maquina, para calcular a
         media e identificar o primeiro e ultimo registro. */
      while (values.hasNext()) {
        String line = values.next().toString();

        /* Pega o inicio e fim do evento atual. */
        String[] tokens = line.split(":");
        start = new Double(tokens[0]).longValue();
        end = new Double(tokens[1]).longValue();

        /* Verifica se eh o registro mais antigo ateh o momento. */
        if (start < traceStart) {
          traceStart = start;
        }
        /* Verifica se eh o registro mais recente ateh o momento. */
        if (end > traceEnd) {
          traceEnd = end;
        }

        /* Acumula o tempo do evento atual a disponibilidade total. */
        sum += (end - start);

        /* Incrementa a contagem de registros. */
        count++;
      }

      /* Calcula a media de disponibilidade para a maquina atual. */
      Long mean = new Long(sum / count);

      /* Filtra os registros onde a media eh menor que 4000 segundos
         e o tempo total eh de pelo menos 300 dias (25920000s = 300d) */
      if ((mean <= 4000) && ((traceEnd - traceStart) >= 25920000)) {
        String v = String.format("% 15d ", mean);
        v += String.format("% 15d ", traceStart);
        v += String.format("% 15d ", traceEnd);
        /* Emite o ID da maquina atual (chave), e sua disponibilidade media
           e os tempos de inicio e fim dos registros concatenados em uma
           string (valor). */
        output.collect(key, new Text(v));
      }
    }
  }

public static void main(String[] args) throws Exception {
```

```
    JobConf conf = new JobConf(TraceFilter.class);

    conf.setJobName("TraceFilter");

    /* Define o a quantidade de tarefas de reducao. */
    conf.setNumReduceTasks(20);

    /* Define o tipo dos pares chave/valor de saida. */
    conf.setOutputKeyClass(LongWritable.class);
    conf.setOutputValueClass(Text.class);

    /* Indica as classes que implementam o mapeamento e reducao. */
    conf.setMapperClass(Map.class);
    conf.setReducerClass(Reduce.class);

    /* Indica o tipo dos arquivos de entrada e saida. */
    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);

    /* Le da linha de comando a origem e destino dos arquivos no HDFS. */
    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));

    JobClient.runJob(conf);
  }
}
```