Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Informática

Dissertação de Mestrado

Processamento Paralelo de Grandes Quantidades de Dados sobre um Sistema de Arquivos Distribuído POSIX

Jonhnny Weslley Sousa Silva

Campina Grande, Paraíba, Brasil
Maio - 2010

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Informática

Processamento Paralelo de Grandes Quantidades de Dados sobre um Sistema de Arquivos Distribuído POSIX

Jonhnny Weslley Sousa Silva

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Redes de Computadores e Sistemas Distribuídos

Francisco Vilar Brasileiro
(Orientador)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

S586p Silva, Jonhnny Weslley Sousa.
Processamento paralelo de grandes quantidades de dados sobre um sistema de arquivos distribuído POSIX / Jonhnny Weslley Sousa Silva. ─ Campina Grande, 2010.
52 f.: il. col.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.
Orientador: Prof. Dr. Francisco Vilar Brasileiro.
Referências.

1. Sistemas de Processamento Distribuído. 2. Sistema de Arquivo Distribuído. 3. Processamento Paralelo de Dados. I. Título.

CDU
004.75(043)

"PROCESSAMENTO PARALELO DE GRANDES QUANTIDADES DE DADOS SOBRE UM SISTEMA DE ARQUIVOS DISTRIBUÍDOS POSIX"

JONHNNY WESLLEY SOUSA SILVA

DISSERTAÇÃO APROVADA EM 21.05.2010

FRANCISCO VILAR BRASILEIRO, Ph.D
Orientador(a)

DALTON DARIO SEREY GUERRERO, D.Sc
Examinador(a)

CARLOS ANDRE GUIMARAES FERRAZ, Ph.D
Examinador(a)

CAMPINA GRANDE - PB

Resumo

Aplicações que processam grandes quantidades de dados estão se tornando cada vez mais presentes nos mais diversos setores, que envolvem desde a academia até sites de compras e redes sociais. Contudo, dispor de uma infraestrutura capaz de realizar este tipo de processamento hoje custa caro, já que as soluções existentes assumem a existência de um conjunto de máquinas dedicadas. Enquanto isso, as estações de trabalho não utilizam grande parte do espaço em disco que possuem. Visando agregar o espaço livre destas estações de trabalho conectadas por uma rede local (LAN), foi construído o Beehive File System (BeeFS), um sistema de arquivos distribuído desenvolvido para atender a requisitos de escalabilidade e manutenabilidade não oferecidos por sistemas de arquivos distribuídos amplamente utilizados na prática, como NFS e Coda. Considerando a intríseca distribuição dos dados no BeeFS, fica evidente a possibilidade de realizar processamento de grandes quantidades de dados de forma distribuída. Porém, uma vez que o BeeFS é formado por estações de trabalho, existe a preocupação quanto à intrusividade, ou seja, a experiência do usuário de uma máquina que faz parte do sistema pode ser prejudicada devido a execução não-solicitada de aplicações que processam grandes volumes de dados. Visando mitigar este problema, esta dissertação apresenta heurísticas para a alocação de arquivos no BeeFS. Estas heurísticas tentam aumentar as chances de que os arquivos estarão disponíveis para processamento em estações de trabalho ociosas. Para isto, as heurísticas consideram dados históricos sobre a utilização do sistema para decidir onde armazenar as réplicas de um arquivo que será utilizado para processamento. Isso, juntamente com um simples escalonador de aplicações que evita executar aplicações em máquinas que não estão ociosas, reduz drasticamente a inconveniência que estas aplicações podem levar a outros usuários. Os resultados mostram que as heurísticas que consideram a média histórica de disponibilidade das estações de trabalho e, ao mesmo tempo, realizam o balanceamento da quantidade de espaço de armazenando entre as máquinas possuem desempenho melhor do que as heurísticas que não consideram a disponibilidade das máquinas.

Abstract

Data-intensive applications are becoming increasingly more present in various sectors, since academia to shopping websites and social networks. However, the most of existing solutions assume the utilization of clusters to perform these applications, and clusters are an expensive resource. Meanwhile, the workstations do not use much of the local storage space they have. In order to use the free space of these workstations, we built the Beehive File System (BeeFS), a distributed file system designed to meet the requirements of scalability and maintainability not offered by distributed file systems widely used in practice, such as NFS and Coda. Considering the natural distribution of data in BeeFS, it is evident that BeeFS can be used to process vaste amounts of data in a distributed way. However, since BeeFS consists of shared workstations, the execution of unsolicited data-intensive applications may impact the performance that users logged in these workstations experience. To mitigate this problem, this work presents data placement heuristics for file allocation in BeeFS. These heuristics try to increase the probability that files will be available for processing on idle workstations. For this, the heuristics take into account historical data about the use of system to decide where to store the file replicas that will be used for processing. These heuristics, coupled with a simple application scheduler that prevents run applications on non-idle machines, it drastically reduces inconvenience that these applications can lead to other users. The results show that the heuristics that consider the historical availability of workstations and, at the same time, realize balancing the amount of storage space between the machines have better performance than the heuristics do not consider the availability of machines.

Dedicatória

Dedico este trabalho às minhas noites sem dormir com uma profunda apreciação pela experiência e conhecimento que cada uma me trouxe.

Agradecimentos

Quando começo a pensar em todas as pessoas às quais gostaria de expressar minha gratidão pela ajuda, sugestões e apoio para tornar este trabalho possível, a lista não para de crescer — segundo o Acordo ortográfico instituído no final de 2008 não será mais usado o acento diferencial, ou seja, para é para mesmo. Primeiramente, gostaria de agradecer à minha família, por criarem um ambiente onde pude sentir amor e incentivo, e onde meus pensamentos pudessem convergir calma e sabiamente rumo às melhores decisões, não apenas para a elaboração deste trabalho, mas em todos os momentos da minha vida.

A Erick Passos, pela amizade e incentivo para participar desta empreitada. Dois anos é tempo demais.

A Lauro Beltrão, meu primeiro mentor em Campina Grande, pelo conhecimento e amizade.

A Fubica, pela orientação, dicas sagazes e bom humor. Agradecimentos muito especiais aos meus companheiros de trabalho no projeto BeeFS, Thiago "Manel", aquele que possui um humor tão seco quanto o Atacama e é tão gentil quanto um Vogon; Alex "Gonzaguinha", pelas fantásticas interpretações de Maria Betânia; Carlinha, pelo toque feminino e Ana Clara, a novata que não fala :D.

Aos amigões, Paulo Ditarso, João Arthur, Marcus Carvalho, Raquel Vigolvino Lopes, Ricardo Araújo, Alan "Tokinho", Giovanni "Surubim", Tomás Barros e outros, pelo companherismo e momentos divertidos.

A Lesandro Ponciano, por ter coragem sucifiente para aceitar dividir um apartamento comigo e pelas tentativas fracassadas de dominar o mundo durante as noites de sexta-feira jogando Rise Of Nations.

À trilogia "O Guia do Mochileiro das Galáxias", de Douglas Adams, que li nos momentos de descontração enquanto escrevia incessantemente esta dissertação.

Às pessoas que fazem parte do Laboratório de Sistemas Distribuídos, por tornarem esse ambiente um excelente local de trabalho.

Sumário

Lista de Figuras

Lista de Códigos Fonte

Capítulo 1

Introdução

Este capítulo apresenta a dissertação em linhas gerais. Inicialmente, é mostrado o contexto no qual este trabalho está inserido, bem como a motivação para realizá-lo. Em seguida, é discutido o problema tratado e os requisitos estabelecidos para uma solução. O capítulo é finalizado com uma descrição da organização do restante dessa dissertação.

1.1 Motivação

Aplicações que processam grandes quantidades de dados estão se tornando cada vez mais presentes nos mais diversos setores, que envolvem desde a academia até sites de compras e redes sociais. Em áreas de pesquisa como bioinformática, processamento de imagens, física de altas energias, simulação de modelos sísmicos entre outras, existem aplicações que trabalham com miríades de dados. Além disso, já discute-se um paradigma emergente em ciência [HTT09], segundo o qual descobertas científicas serão realizadas por recursos computacionais que ajudam os pesquisadores a manipular e explorar grandes conjuntos de dados. De forma analóga, as corporações com fins lucrativos estão coletando e analisando um volume cada vez maior de dados. A partir desses dados, estas empresas podem compreender melhor as preferências dos clientes para aperfeiçoar o plano de *markerting*, ou melhorar a eficiência do serviço através da redução de gastos, por exemplo. Desta forma, elas podem produzir melhores serviços para os clientes [DH07] e se tornarem mais competitivas.

Para processar grandes conjuntos de dados, em geral, é necessário uma infraestrutura de computação de alto desempenho. Contudo, dispor desta infraestrutura capaz de realizar

este tipo de processamento hoje custa caro. Tecnologias mais recentes, como o Google File System [GGL03] e o Map/Reduce [DG08], são capazes de armazenar e processar grandes volumes de dados de forma eficiente utilizando *clusters* de máquinas convencionais. Embora máquinas commodity sejam mais baratas, estas tecnologias assumem que as máquinas que formam o *cluster* são dedicadas. Logo, os custos associados à aquisição e à manutenção de um *cluster* formado por dezenas ou centenas de máquinas dedicadas pode não ser viável para várias entidades que precisam realizar algum tipo de processamento envolvendo grandes conjuntos de dados.

Com o advento da computação na nuvem, tornou-se mais fácil utilizar recursos computacionais e pagar apenas pela quantidade de recursos consumidos. Porém, quando o recurso computacional envolve grandes quantidades de dados, significa que será necessário realizar o upload destes dados. E, ainda, há a preocupação quanto à confidencialidade e à privacidade dos dados. De qualquer forma, por que utilizar recursos externos, se este processamento pode ser realizado, ou pelo menos parte dele, nas estações de trabalho da rede local? Esta estratégia é viável, já que a capacidade dos discos rígidos atuais superou as necessidades individuais de muitos usuários, deixando-os com muito espaço de armazenamento livre em suas estações de trabalho [DB99]. Além disso, uma quantidade significativa das estações de trabalho de uma rede local apresentam baixo nível de utilização tanto do CPU quanto do disco rígido [BDET00].

Com o intuito de utilizar este espaço de armazenamento não utilizado das estações de trabalho, nosso grupo de pesquisa está desenvolvendo o Beehive File System [PSSB10]. O Beehive File System (BeeFS) é um sistema de arquivos distribuído desenvolvido para atender a requisitos de escalabilidade e manutenabilidade não oferecidos por sistemas de arquivos distribuídos amplamente utilizados na prática, como NFS [PJS+94] e Coda [SKK+90]. Ele explora o espaço de armazenamento não utilizado das estações de trabalho de uma rede local para compor um sistema de arquivos com visão global e acesso transparente para os arquivos, os quais estão armazenados nos discos das máquinas participantes. Por utilizar a infraestrutura subutilizada já existente, o BeeFS é capaz de fornecer uma solução de armazenamento distribuído mais barata do que as soluções existentes.

Considerando a intríseca distribuição dos dados no BeeFS, fica evidente a possibilidade de realizar processamento de grandes quantidades de dados de forma distribuída. Porém,

uma vez que o BeeFS é formado pelas estações de trabalho de uma rede local, existe a preocupação quanto à intrusividade, ou seja, a experiência do usuário de uma máquina que faz parte do sistema pode ser prejudicada devido a execução não-solicitada de aplicações que processam grandes volumes de dados.

Visando mitigar este problema, este trabalho apresenta heurísticas para a alocação de arquivos no BeeFS. Estas heurísticas foram desenvolvidas com o objetivo de aumentar as chances de que os arquivos estarão disponíveis para processamento em estações de trabalho ociosas. Para isto, as heurísticas consideram dados históricos sobre a utilização do sistema para decidir onde armazenar as réplicas de um arquivo que será utilizado para processamento. Juntamente com estas heurísticas, um simples escalonador de aplicações que evita executar aplicações em máquinas que não estão ociosas, reduz drasticamente a inconveniência que estas aplicações podem levar a outros usuários. Assim, espera-se viabilizar a execução de aplicações de uso intensivo de dados sobre o sistema de arquivos BeeFS de forma eficiente, sem prejudicar a experiência do usuário.

1.2 Objetivos

O objetivo deste trabalho é apresentar uma solução que viabilize a execução de aplicações de uso intensivo de dados sobre o sistema de arquivos BeeFS, sem prejudicar a experiência do usuário. Espera-se com isso reduzir os custos normalmente associados à execução dessas aplicações, uma vez que o BeeFS utiliza a infra-estrutura já existente para compor um sistema eficiente e escalável. A solução apresentada neste trabalho deve atender aos seguintes requisitos:

1. Viabilizar a execução de aplicações de uso intensivo de dados sobre um sistema de arquivos distribuído POSIX;

2. Demonstrar a praticabilidade de se utilizar, ao mesmo tempo, o sistema de arquivos distribuído BeeFS tanto para uso convencional quanto para aplicações de uso intensivo de dados;

3. Elaborar mecanismos que controlem os impactos de desempenho nas máquinas que executarão as aplicações de uso intensivo de dados.

O requisito 1 define que a solução proposta por este trabalho deve ser uma alternativa às propostas existentes no escopo das aplicações de uso intensivo de dados. Como o sistema de arquivos BeeFS foi originalmente concebido para atuar como um sistema de arquivos convencional — ou seja, salvar documentos pessoais, arquivos de mídia, etc. —, o requisito 2 estabelece que a solução deve permite que o sistema de arquivos BeeFS continue sendo utilizado para atender a essas aplicações, ao passo que, também, possa ser utilizado para a execução de aplicações de uso intensivo de dados. O requisito 3 especifica a existência de mecanismos que evitem a degradação do desempenho das máquinas dos usuários em decorrência da execução de aplicações de uso intensivo de dados. Este requisito é especialmente importante, uma vez que o sistema de arquivos BeeFS é instalado sobre um conjunto estações de trabalho, as quais não são recursos dedicados e, portanto, podem está sendo utilizadas pelos seus respectivos usuários.

1.3 Organização da Dissertação

O restante desta dissertação está organizada da seguinte forma. O Capítulo 2 contextualiza esse trabalho, apresentando o estado da arte da área. Assim, as propostas existentes para a execução de aplicações de uso intensivo de dados são descritas e relacionadas a este trabalho.

O Capítulo 3 apresenta e discute o resultado principal desta dissertação: o de que é possível executar aplicações de uso intensivo de dados sobre um sistema de arquivos distribuído POSIX em ambientes de recursos não-dedicados usando heurísticas para alocação de arquivos.

O Capítulo 4 apresenta a arquitetura do BeeFS e os conceitos relacionados ao funcionamento do mesmo, as motivações de se utilizar sistemas de arquivos distribuídos POSIX para se executar aplicações de uso intensivo de dados.

O Capítulo 5 discute as questões que devem ser abordadas para se conseguir bom desempenho em aplicações de uso intensivo de dados executando sobre sistemas de arquivos distribuídos em redes de estações de trabalho. Adicionalmente, é apresentado um estudo de caso no qual demonstra-se a viabilidade de se utilizar o BeeFS como substrato para a execução de aplicações de uso intensivo de dados de forma simples e eficiente.

Por fim, no Capítulo 6 são apresentadas as considerações finais e idéias de possíveis

trabalhos futuros.

Capítulo 2

Trabalhos Relacionados

Considerando o status quo dos sistemas de arquivos distribuídos para processamento paralelo de dados, este capítulo apresenta alguns deles classificando-os em dois grupos: sistemas de arquivos em *clusters* e sistemas de arquivos oportunistas.

Os sistemas de arquivos em *clusters* usam redes de máquinas dedicadas. Comumente, estes sistemas implementam alguma forma de *striping*, ou seja, os arquivos são divididos e espalhados através de várias máquinas do *cluster*. Desta forma, é possível aumentar a largura de banda agregada e, também, maximizar o tamanho máximo de um arquivo. Por causa desta técnica, é comum a existência de arquivos com vários *Gbytes* (GiB), como no Google File System. Geralmente, estes sistemas de arquivos distribuídos não disponibilizam uma interface para acesso transparente ao conteúdo dos arquivos, isso seria possível se eles implementassem uma interface POSIX. Pois, as aplicações já existentes que executam sobre um sistema de arquivos local, como ext3 [Twe00], poderiam executar também em quaisquer outros sistemas de arquivos, desde que estes atendam as normas POSIX. Além disso, dispor de uma infraestrutura de *cluster* capaz de realizar processamento paralelo de dados hoje custa caro.

Visando dimuinuir os custos associados à aquisição e manutenção de um *cluster*, foram desenvolvidos os sistemas de arquivos oportunistas. Estes sistemas exploram o espaço de armazenamento não utilizado das estações de trabalho de uma rede local para criar um sistema de armazenamento distribuído. Existem vários sistemas de armazenamento oportunistas, mas, este trabalho considera apenas aqueles que possibilitam a execução de aplicações de uso intensivo de dados. Neste caso, eles normalmente não implementam uma interface POSIX,

e o acesso aos dados se dá por meio de uma interface de programação própria do sistema.

2.1 Sistemas de arquivos em *clusters*

Lustre [CFS02] é um sistema de arquivos distribuído em *clusters*. Ele possui três unidades funcionais: $MetadataServers$, $ObjectStorageTargets$ e clientes. Os *Metadata Servers* (MDS) armazenam todos os metadados do sistema de arquivos — nomes dos arquivos, diretórios, permissões — e mantêm os registros das transações que resultam em mudanças nos arquivos. Além disso, os MDS suportam todas as operações que envolvem a criação, a manipulação de propriedades e a localização de arquivos no sistema. Adicionalmente, eles também são responsáveis pelo direcionamento das requisições de I/O para os *Object Storage Targets* (OSTs), os quais gerenciam os dados nos dispositivos de armazenamento. Lustre é baseado em objetos, ou seja, ao invés de utilizar discos rígidos para armazenamento dos dados são usados *Object Storage Devices* (OSDs) que são um conjunto formado por CPU, interface de rede e disco local ou RAID [GNA+98]. Os OSDs substituem a interface em nível de blocos por uma interface de objetos através da qual os clientes podem realizar operações de I/O com quantidades maiores de dados. Os clientes interagem com os MDSs para executar operações com metadados, e realizam operações de I/O comunicando-se diretamente com os OSDs. Adicionalmente, a comunicação dos clientes com as demais unidades funcionais do sistema (MDSs e OSTs) é realizada através de uma interface POSIX que aceita operações concorrentes de I/O para os arquivos do sistema.

Assim como Lustre, Ceph [WBM+06] é um sistema de arquivos distribuído em *clusters* baseado em objetos. Contudo, Ceph busca prover maiores escalabilidade, confiabilidade e tolerância às falhas. Enquanto Lustre assume que os mecanismos de RAID utilizados nos OSDs são suficientes para garantir confiabilidade, Ceph assume que a ocorrência de falhas é normal e não uma situação excepcional. Portanto, para manter a disponibilidade do sistema, Ceph aplica mecanismos de replicação com cópia primária [AD76]. Os dados são replicados em grupos de replicação, os quais mapeiam as cópias para uma lista de N OSDs, onde N representa o fator de replicação. Assim, os clientes enviam as operações de escrita para a cópia primária, que atribui um nova versão para o arquivo e propaga a operação para as réplicas. Para que os clientes possam acessar os dados, Ceph possui uma interface bem

parecida à interface provida pelo POSIX.

O Google File System é um sistema de arquivos distribuído em *clusters* projetado para aplicações que processam quantidades massivas de dados. A arquitetura deste sistema envolve a presença de um servidor central de metadados (*Master Server*), vários servidores de dados (*chunkservers*) e vários clientes. O servidor de metadados mantém todos os metadados e é responsável por todas as decisões do sistema. Os arquivos são divididos em pedaços menores (os *chunks*) que são armazenados nos *chunkservers*. Por causa dos *chunks* é possível a existência de arquivos com vários GiB. Para garantir confiabilidade, é utilizado um mecanismo de replicação dos *chunks*, assim a ocorrência de falhas em um ou mais *chunkservers* não afetará a disponibilidade dos arquivos. A maioria dos arquivos são alterados através de *appends* ao invés de escritas em posições randômicas do arquivo. O Google File System foi projetado para suportar aplicações cujo *workload* típico geralmente envolve a realização de leituras sequenciais nos arquivos. Entre as aplicações que executam no Google File System, estão BigTable [CDG+06][1], Map/Reduce [DG08] e Sawzall [PDGQ05] — estes dois últimos são detalhados em seguida.

Map/Reduce [DG08] é um framework que executa sobre o Google File System e permite o desenvolvimento de aplicações transparentes quanto à distribuição. O desenvolvimento de aplicações utilizando este framework é baseado em duas primitivas básicas: *map* e *reduce*. Na primeira, a aplicação processa um conjunto de pares chave-valor e produz um conjunto intermediário de pares chave-valor. Na segunda fase, a aplicação deve emitir uma coleção de valores associados a uma chave específica ao domínio da aplicação a partir do conjunto intermediário de pares chave-valor gerado na primeira fase. Entre a execução destas fases, o framework agrupa e ordena as chaves de forma distribuída para otimizar o processamento durante o reduce, a Figura 2.1 mostra este fluxo de execução. O framework Map/Reduce encapsula detalhes referentes ao processamento distribuído, tais como: tolerância a falhas, migração do processo e comunicação entre as máquinas.

Mais detalhadamente, o Map/Reduce possui dois componentes principais: o *JobTracker* e o *TaskTracker*. O *JobTracker* recebe as tarefas submetidas pelos programas clientes. Cada tarefa especifica as funções *map*, *combine* e *reduce* a serem utilizadas, bem como quais serão os arquivos de entrada e de saída. Inicialmente, o *JobTracker* determina o número de partes

[1] BitTable é um banco de dados distribuído de alta performance elaborado pela Google

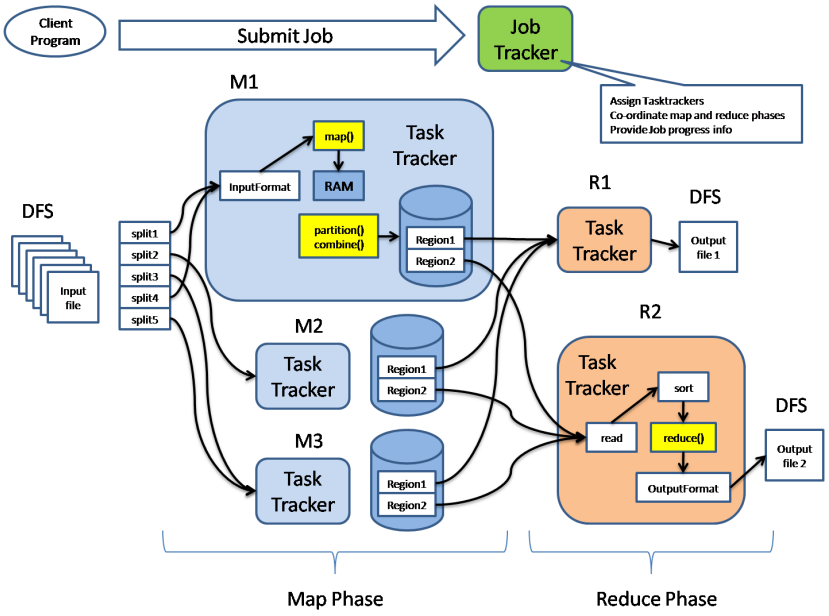

Figura 2.1: Fluxo de execução do Map/Reduce

no qual os arquivos de entrada serão divididos — geralmente, os arquivos de entrada são divididos em partes cujos tamanhos variam de 16 a 64 MiB, mas este valor é configurável. Em seguida, o *JobTracker* seleciona alguns *TaskTrackers* conforme a proximidade destes em relação às máquinas que contêm os arquivos de entrada, visando diminuir o tráfego de dados na rede. Então, o *JobTracker* envia as requisições de execução das tarefas para os *TaskTrackers* selecionados.

Cada *TaskTracker* executará a função *map* provida, a qual utiliza os dados extraídos por uma das partes do arquivo de entrada para produzir pares chave-valor. Como, em alguns casos, há uma significante repetição das chaves produzidas pela função *map*, foi adicionado um refinamento ao Map/Reduce que permite ao usuário definir uma função *combine*. Esta função realiza um merge parcial dos dados antes de enviá-los através da rede para o *TaskTracker* que realizará a função *reduce*. Geralmente, esta função é a mesma utilizada na fase *reduce*.

Quando as tarefas de *map* terminam, cada um dos *TaskTrackers* notificará o *JobTracker*, que irá notificar os *TaskTrackers* selecionados para executar as tarefas de *reduce*. Cada *TaskTracker* realizará uma leitura remota dos valores produzidos durante a fase *map*. Após a leitura, é realizada uma ordenação dos pares chave-valor e para cada chave, é invocada a função *reduce*, a qual coleta os valores agregados para o arquivo de saída — será produzido

um arquivo para cada *reduce* realizado.

Para tolerar falhas, o *JobTracker* mantém o progresso de cada uma das tarefas e periodicamente verifica o estado do *TaskTracker*. Quando um dos *TaskTrackers* falha, o *JobTracker* seleciona outro *TaskTracker* e reexecuta a tarefa. Contudo, na ocorrência de falhas, tarefas de *map* já completadas precisam ser reexecutadas, já que a saída destas é realizada em disco local; enquanto tarefas de *reduce* já completadas não precisam, pois são armazenadas em um sistema de arquivos distríbuido.

Map/Reduce é uma excelente abstração para o processamento de dados em larga escala, mas ele é genérico demais para o desenvolvimento de várias tarefas comuns — como contar o número de acessos de uma página web. Visando simplificar o trabalho necessário para implementar estas tarefas, foi desenvolvida uma linguagem para atuar como interface sobre o Map/Reduce, chamada Sawzall [PDGQ05]. A função básica de um programa Sawzall é agregar uma coleção de registros. Os registros são associados através de um agregador, que é conhecido como *tabela*. O resultado da associação apresenta os registros em um formato definido pelo usuário. Por exemplo, a agregação poderia ser a soma de um determinado valor dos registros, como mostrado no programa abaixo:

Código Fonte 2.1: Exemplo de uma aplicação Sawzall

```
1 total: table sum of float;
2 x: float = input;
3 emit total <- x;
```

Assim como Sawzall, Pig [ORS+08] é uma linguagem de alto-nível para processar grandes quantidades dados. No entanto, Sawzall executa sobre o Map/Reduce da Google e Pig executa sobre o Map/Reduce do Hadoop. Além disso, estas linguagens são diferentes entre si. A sintaxe de Sawzall é bastante influenciada por Java ou Pascal, enquanto Pig é motivada por tentar estender SQL ou Álgebra Relacional. O compilador Pig produz programas Map/Reduce para executar tarefas de pesquisa em termos de um conjunto de transformações. Por exemplo, aplicar uma função para cada registro de uma entrada, ou agrupar os registros conforme algum critério e aplicar uma função para cada grupo. Estas transformações são paralelizáveis, ou seja, o processamento lógico para cada registro (ou grupo de registros) é auto-contido, e a ordem da saída produzida é irrelevante. Além disso, os programas Pig sofrem otimizações para executar mais eficientemente.

GridBatch [LO08] é um modelo de programação que esconde a complexidade da programação paralela, mas permite ao usuário controle sobre como os dados serão particionados e distribuídos, com o intuito de aumentar a performance das aplicações. GridBatch consiste de duas partes: o sistema de arquivos distribuído e o *job scheduler*. O sistema de arquivos distribuído utilizado é o Hadoop Distributed File System, que é responsável pelo gerenciamento dos arquivos através de todos os nós que fazem parte do sistema. O *job scheduler* inclui um nó principal, o *mestre*, e vários nós *escravos*. O *mestre* é responsável por dividir um *job* em tarefas menores conforme a aplicação do usuário e atribuir estas tarefas para os nós *escravos*, que as executarão. As aplicações são desenvolvidas a partir de um conjunto de operadores fundamentais, os quais são fornecidos por uma API programática provida pelo GridBatch, diferentemente de Sawzall e Pig que fornecem linguagens específicas para o desenvolvimento das aplicações.

2.2 Sistemas de arquivos oportunistas

Freeloader [VMF+05] é um sistema de armazenamento que agrega o espaço disponível das estações de trabalho de uma rede local para fornecer armazenamento para grandes volumes de dados produzidos em grandes centros de pesquisa. Dessa forma, os cientistas destes centros podem utilizar os dados para executar experimentos e simulações. FreeLoader divide os arquivos em pedaços menores (*chunks*) e os distribui entre os nós que participam do sistema — esta técnica é conhecida como *striping* —, tornando possível que haja processamento paralelo no FreeLoader. Além das estações de trabalho que provêem armazenamento, há um servidor central que gerencia os metadados dos arquivos bem como a localização dos *chunks* de cada arquivo. Os clientes podem acessar os arquivos a partir da API fornecida pelo FreeLoader, mas não é permitido a modificação dos arquivos, ou seja, eles são imutáveis. Desta forma, diferentemente do BeeFS, o FreeLoader somente pode ser utilizado para processamento paralelo de dados. Enquanto isso, o BeeFS é um sistema de arquivos para uso genérico, que permite a modificação dos arquivos e possui a flexibilidade necessária para atender a vários tipos de aplicações através da extensibilidade provida pelo uso dos metadados.

BitDew [FHC08] é um sistema que gerencia dados distribuídos em uma grade computa-

cional, sistema de computação na nuvem ou grades formadas por estações de trabalho. Ele possibilita a execução de aplicações de uso intensivo de dados com varredura de parâmetros, aplicações que necessitam de serviços de *checkpoint* e aplicações de *workflow*. Porém, uma vez que Bitdew foi desenvolvido para ser um sistema genérico para gerenciamento de dados, nenhuma medida visando a amotização do impacto causado pela execução de aplicações, quando ele está instalado sobre uma infraestrutura de recursos não dedicados.

Capítulo 3

Processamento paralelo de grandes quantidades de dados usando recursos não-dedicados

Este capítulo descreve o modelo para escalonamento de aplicações de uso intensivo de dados sobre sistemas de arquivos distribuídos para armazenar os dados de entrada que foi utilizado para definir as heurísticas para alocação de arquivos descritas neste trabalho. Este modelo também foi implementado no simulador onde foram executadas as simulações realizadas para apresentação dos resultados desta dissertação.

Definido o modelo, serão apresentadas as heurísticas para alocação de arquivos elaboradas neste trabalho. Estas heurísticas consideram dados históricos sobre a utilização do sistema para decidir onde armazenar as réplicas de um arquivo que será utilizado para processamento. Foram definidas 6 heurísticas, nomeadas: all, $equalizer$, $maxAvail$, $eqMaxAvail$, $meanAvail$ e $eqMeanAvail$. O funcionamento de cada uma destas heurísticas será descrito detalhadamente neste capítulo.

A avaliação de desempenho foi realizada através de simulações que avaliaram cada uma das heurísticas citadas. Para avaliar os resultados produzidos, as métricas comparadas foram o tempo de execução da aplicação e a quantidade de armazenamento utilizado. Em seguida, serão descritos os cenários executados e os resultados obtidos.

3.1 Modelo do sistema

Esta seção apresenta o modelo adotado no restante da dissertação. O modelo elaborado é composto por duas partes principais. A primeira é formada por um sistema de arquivos distribuído, para o qual foram definidas as heurísticas de alocação de arquivos elaboradas neste trabalho. Enquanto a outra parte, define o modelo para escalonamento de aplicações de uso intensivo de dados. Este modelo também foi implementado no simulador onde foram executadas as simulações realizadas para apresentação dos resultados desta dissertação.

3.1.1 O sistema de arquivos

O modelo do sistema de arquivos se baseia na arquitetura do BeeFS. Desta forma, um sistema de arquivos é formado por um servidor de metadados e um conjunto de máquinas, que atuam tanto como servidores de dados quanto clientes. O servidor de metadados é responsável por mapear a localização dos arquivos. Para isso, ele utiliza as heurísticas de alocação de arquivos. Os servidores de dados armazenam os arquivos de entrada necessários para a execução das tarefas, bem como os dados de saída gerados. Os clientes podem acessar os arquivos armazenados no sistema. Visando simplificar o modelo, os clientes são responsáveis por executar as tarefas das aplicações.

O sistema de arquivos possui um conjunto de arquivos cujos tamanhos variam conforme alguma distribuição. Cada arquivo possui várias cópias que são armazenadas pelos servidores de dados. O número de cópias de cada arquivo é definido pelo nível de replicação. Os arquivos podem ser acessados a partir de qualquer máquina. Esta consideração implica que eles podem ser acessados de forma local ou remota. O acesso local ocorre quando uma cópia do arquivo está armazenada na mesma máquina que o está acessando. Por outro lado, quando uma cópia do arquivo não está presente na máquina que realiza o acesso, os dados são enviados pela rede a partir de um servidor de dados que mantenha uma cópia do arquivo desejado. Evidentemente, os acessos locais são mais rápidos do que os acessos remotos. Para modelar o impacto da sobrecarga causada pelo acesso remoto aos arquivos, foi considerada a seguinte equação:

$$t_r(A) = t_l(A) \times k \tag{3.1}$$

onde $t_r(A)$ é o tempo estimado para realizar o acesso remoto para um arquivo A, $t_l(A)$ é o tempo estimado para realizar o acesso local para o arquivo A e k é o fator de impacto para realizar acesso remoto aos arquivos, onde $k > 1$.

3.1.2 A aplicação e os dados de entrada

Uma aplicação é composta por um conjunto independente de tarefas que podem ser executadas em qualquer ordem, ou seja, são consideradas aplicações do tipo Bag-of-Tasks [CPC+03; SS96]. Geralmente, as aplicações de uso intensivo de dados possibilitam que os dados de entrada sejam particionados facilmente em fragmentos menores. Estes fragmentos podem ser processados em paralelo e independentemente, tornando mais simples o uso de recursos amplamente distribuídos para execução destas aplicações. Porém, para evitar que a lógica de particionamento dos dados torne o modelo mais complexo, são consideradas aplicações que recebem como entrada um conjunto de arquivos. Cada arquivo é processado por uma tarefa. Além disso, apenas uma única tarefa pode ser executada por uma máquina qualquer em um dado momento.

Os dados de entrada de uma aplicação são definidos como o conjunto de arquivos de entrada de todas as tarefas que formam a aplicação.

3.1.3 Disponibilidade

A disponibilidade é o intervalo de tempo que uma máquina permanece ociosa. Segundo o conceito de disponibiliade adotado no modelo deste trabalho, uma máquina está disponível se ela não estiver sendo utilizada por algum usuário. Em outras palavras, uma máquina é considerada disponível caso ela esteja apta a executar alguma tarefa de uma aplicação. A disponibilidade de uma máquina é definida por um par de valores que representam o momento em que a máquina entrou em ociosidade e a duração de tempo no qual ela permaneceu neste estado. Formalmente, temos:

$$D_M = \{< d_1, \Delta t_1 >, < d_2, \Delta t_2 >, \ldots, < d_n, \Delta t_n >\} \tag{3.2}$$

onde D_M é o conjunto de tuplas que definem todos os períodos de disponibilidade da máquina M. Cada tupla possui um par de valores, $< d_n$, $\Delta t_n >$, os quais correspondem

ao instante de tempo no qual a máquina ficou ociosa e a duração do tempo de ociosidade da máquina, respectivamente.

Por outro lado, uma máquina é considerada indisponível se ela não estiver ociosa, independentemente do motivo, seja ele devido à chegada do usuário ou porque a máquina foi desligada/reiniciada. Logo, uma tarefa será executada em uma determinada máquina apenas se esta encontrar-se disponível. Porém, uma máquina pode torna-se indisponível durante a execução de uma tarefa, ou seja, antes da tarefa encerrar a sua execução. Neste caso, a execução é abortada e uma nova execução desta tarefa será escalonada. Desta forma, garante-se que a execução de tarefas não interfira nas atividades do usuário.

3.1.4 Alocação de arquivos

A alocação de arquivos é o processo que determina em quais servidores de dados serão armazenadas as cópias de um dado arquivo. Neste trabalho, as heurísticas para alocação de arquivos são o principal foco de estudo. Estas heurísticas devem aumentar o máximo possível a probabilidade de que a cópia de um arquivo estará disponível para processamento em uma máquina ociosa. Portanto, as heurísticas para alocação de arquivos devem considerar as informações sobre o histórico de disponibilidade das máquinas para decidir os servidores de dados que armazenarão as cópias de um arquivo.

3.1.5 Métricas de desempenho

Para avaliar os resultados deste trabalho, foram consideradas duas métricas de desempenho: o tempo de execução das aplicações e a quantidade de armazenamento.

Os algoritmos que são analisados neste trabalho almejam, entre outros objetivos, reduzir o tempo de execução da aplicação. Logo, a avaliação da eficiência de cada heurística de escalonamento será efetuada através da verificação do tempo de execução da aplicação. O tempo total de execução da aplicação é definido como o intervalo de tempo compreendido entre a submissão da aplicação e o momento em que a última tarefa escalonada finalizou sua execução. Esta métrica também é útil para avaliar heurísticas para alocação de arquivos, já que o tempo de execução de uma aplicação está estritamente relacionado à disponibilidade dos arquivos em máquinas ociosas.

As heurísticas para alocação de arquivos também foram avaliadas em relação à quantidade de armazenamento utilizado para manter as cópias dos arquivos. Desta forma, as heurísticas podem ser comparadas quanto à quantidade de armazenamento necessário para garantir que os arquivos estejam disponíveis para processamento em máquinas ociosas. Esta métrica é importante porque um dos meios de reduzir o tempo de execução de uma aplicação é por meio do aumento do nível de replicação dos arquivos.

3.2 Modelo de simulação

Uma vez que este trabalho visa demonstrar a utilização de heurísticas para a alocação de arquivos baseadas em dados históricos sobre a utilização do sistema, é desejável que sejam utilizados traces reais que descrevam a disponibilidade das máquinas do sistema. Sendo assim, as simulações realizadas utilizaram os traces produzidos em [KTI+04] para modelar a disponibilidade das máquinas. Este trace foi escolhido por caracterizar um ambiente muito semelhante àquele para o qual o sistema BeeFS visa atender, ou seja, estações de trabalho conectadas por uma rede local. Com base neste trace, foi possível simular um sistema constituído de 244 máquinas por um período de 14 dias seguindo o mesmo modelo de disponibilidade descrito anteriormente.

Nas simulações, o sistema de arquivos é formado por um servidor de metadados e um conjunto de 244 máquinas, que atuam tanto como servidores de dados quanto clientes. O sistema de arquivos possui inicialmente um conjunto de 1000 arquivos cujos tamanhos variam conforme uma distribuição uniforme entre $500MiB$ e $2GiB$. Os arquivos possuem cópias armazenadas em um ou mais servidores de dados dependendo do nível de replicação. Considerando que os arquivos podem ser acessados de forma local ou remota, foi definido um fator de impacto para modelar o atraso causado pelas leituras de arquivos remotas. Por simplicação, o fator de impacto adotado é definido por um valor constante equivalente a 4.87. Este fator de impacto foi calculado a partir de experimentos realizados também neste trabalho. Estes experimentos avaliaram a sobrecarga das leituras de arquivos remotas em relação às leituras locais. No caso da leituras remotas, a transferência dos dados pela rede causa redução da velocidade nominal da conexão. Neste caso, o modelo de rede mantém o controle sobre o número de conexões de transferência de dados simultâneas na rede e reduz

a largura de banda disponível de acordo com a utilização da banda.

Em aplicações de uso intensivo de dados, o tempo de execução da aplicação é tipicamente uma função do tamanho dos dados de entrada. A explicação para isso é bastante simples. Quanto mais dados devem ser processados, mais tempo é necessário para processá-los. Neste caso, foi adotado o mesmo fator de conversão utilizado em [RF02], formalmente definido por:

$$t(F) = 300 \times S(F) \tag{3.3}$$

onde $t(F)$ é o tempo de processamento do arquivo F e $S(F)$ representa o tamanho do arquivo F em GiB.

O modelo das aplicações adotadas nas simulações caracterizam aplicações do tipo Bag-of-Tasks. Para cada cenário, foi simulada a execução de 300 aplicações Bag-of-Tasks, cada uma destas formada por um conjunto de tarefas. A quantidade de tarefas por aplicação varia conforme uma distribuição uniforme entre 3 e 10. As tarefas de uma aplicação são mapeadas por um escalonador para as máquinas que realizarão o processamento. A literatura descreve vários escalonadores de aplicações com os mais diversos propósitos. Contudo, neste trabalho será adotado o escalonador de aplicação Storage Affinity [SNCBL04]. Uma vez que este escalonador considera a presença dos dados para determinar as máquinas que realizarão o processamento, ele apresenta melhores ganhos em termos de tempo de execução do que outros escalonadores. O escalonador de aplicação Storage Affinity escalona as tarefas nas máquinas de acordo com a afinidade entre elas. Essa afinidade é calculada pela heurística de escalonamento Storage Affinity, que corresponde à quantidade dos dados de entrada da tarefa, em bytes, que já está armazenada na máquina onde o arquivo está localizado. Primeiro, a afinidade de todas as tarefas com todas as máquinas é calculada. Então, a tarefa com o maior valor de afinidade é escolhida. No caso de mais de uma tarefa ter o mesmo valor de afinidade, é escolhida a tarefa com maior quantidade de dados de entrada.

3.3 Heurísticas para alocação de arquivos

Este capítulo descreve as heurísticas para alocação de arquivos elaboradas neste trabalho. O funcionamento destas heurísticas é baseado na utilização de dados históricos sobre a uti-

lização do sistema. A partir dessas informações sobre a disponibilidade das máquinas, as heurísticas são capazes de decidir quais máquinas deverão armazenar as réplicas de um arquivo que será utilizado para processamento. Esta alocação visa ampliar as chances de que as cópias de um arquivo previamente armazenadas no sistema estarão disponíveis para processamento em máquinas ociosas. Esta preocupação tem origem no fato do sistema de arquivos utilizar recursos não-dedicados, ou seja, os recursos responsáveis pelo armazenamento das réplicas podem está sendo utilizados pelo usuário e, portanto, não está disponível para a execução de aplicações.

O processo de alocação de arquivos é feito da seguinte forma: a heurística recebe uma coleção de máquinas e o valor do nível de replicação como parâmetro. Então, R réplicas de um arquivo são alocadas nas máquinas de acordo com a heurística utilizada, onde R representa o nível de replicação indicado. Cada uma das heurísticas avaliadas calcula quais máquinas serão responsáveis por uma réplica conforme alguma informação sobre o sistema, mais especificamente, a disponibilidade das máquinas.

Neste trabalho, foram elaboradas 6 heurísticas para alocação de arquivos: all, $equalizer$, $maxAvail$, $eqMaxAvail$, $meanAvail$ e $eqMeanAvail$. Os detalhes do funcionamento de cada uma delas serão descritos a seguir.

A heurística all cria cópias de cada arquivo em todas as máquinas disponíveis. Na prática, esta heurística não é viável devido aos elevados custos de armazenamento associados. Porém, ela representa o melhor cenário existente para a execução de aplicações, quando se trata de alocação de arquivos em sistemas de armazenamento distribuídos com recursos não-dedicados. Esta afirmação é justificada pelo seguinte fato: em um ambiente com espaço de armazenamento ilimitado, seria possível replicar todos os arquivos em todas as máquinas, diminuindo ao máximo a probabilidade de que todas as máquinas estejam em uso pelo usuário. Portanto, neste trabalho, esta heurística serve de referencial para comparação com as demais heurísticas.

A heurística $equalizer$ cria e espalha as réplicas dos arquivos de forma a balancear o espaço de armazenamento nas máquinas que participam do sistema. Esta heurística representa o mecanismo padrão para alocação de arquivos do sistema de arquivos BeeFS. Assim como a heurística anterior, ela pode ser considerada como valor de referência de comparação com as heurísticas que consideram informações sobre a disponibilidade das máquinas para realizar

a alocação dos arquivos.

A heurística $maxAvail$ cria cópias dos arquivos nas máquinas mais disponíveis do sistema. Por máquinas mais disponíveis, entende-se que são aquelas que acumulam os maiores períodos de ociosidade. Entretanto, é possível que algumas máquinas possuam valores de disponibilidade significativamente maiores do que outras. Estas máquinas mais disponíveis podem rapidamente concentrar um grande número de réplicas de vários arquivos. Esta grande concentração de réplicas em poucas máquinas, além de exigir bastante espaço de armazenamento em cada um delas, pode prejudicar a execução das aplicações, uma vez que embora não esteja sendo utilizada pelo usuário, a máquina pode estar ocupado com a execução de outra aplicação que foi submetida anteriormente. Visando mitigar este problema de concentração de réplicas, criou-se a heurística $eqMaxAvail$. Esta heurística, assim como a anterior, cria cópias dos arquivos nas máquinas mais disponíveis do sistema. Porém, ela também realiza um balanceamento de réplicas visando reduzir a criação de $hotspots$.

A heurística $meanAvail$ cria réplicas dos arquivos nas máquinas que possuem as maiores médias de disponibilidade do sistema. As médias de disponibilidade representam a duração média dos períodos de disponibilidade de uma determinada máquina. Uma heurística que considera a média dos períodos de disponibilidade pode ser mais eficiente do que outra que considera apenas os valores acumulados de disponibilidade, já que torna-se mais difícil a formação de $hotspots$ uma vez que as máquinas geralmente apresentam intervalos de disponibilidade semelhantes, porém em diferentes horários do dia. Além disso, também foi elaborada outra heurística, a $eqMeanAvail$. Esta heurística, assim como a anterior, cria réplicas dos arquivos nas máquinas que possuem as maiores médias de disponibilidade do sistema. Porém, ela também realiza um balanceamento de réplicas visando otimizar o processo de utilização do espaço de armazenamento.

3.4 Avaliação de desempenho

Esta seção descreve os resultados da avaliação de desempenho das heurísticas para alocação de arquivos apresentadas na seção anterior. Antes disso, são descritos os cenários de simulação realizados neste trabalho e a validade dos resultados obtidos.

3.4.1 Descrição dos cenários

Com o objetivo de investigar o desempenho das aplicações em relação a cada uma das heurísticas para alocação de arquivos, as simulações foram executadas considerando um vasto conjunto de cenários distintos. As características de cada cenário variam de acordo com dois aspectos principais: as heurísticas para alocação de arquivos e o nível de replicação. Nas simulações, foram consideradas todas as heurísticas apresentadas anteriormente na Seção 3.3. Estas heurísticas são all, $equalizer$, $maxAvail$, $eqMaxAvail$, $meanAvail$ e $eqMeanAvail$.

Visando investigar o impacto do nível de replicação no desempenho das aplicações, foram feitas simulações variando o nível de replicação dos arquivos durante o processo de alocação. Esse nível de replicação corresponde ao número de réplicas que um arquivo possui. Neste trabalho, foram simulados cenários considerando níveis de replicação que variavam de 1 até 244, valor máximo que nível de replicação poderia possuir considerando o número de máquinas do sistema simulado.

3.4.2 Validade dos resultados

Para que os resultados das simulações ficassem estatisticamente válidos, foi preciso simular pouco mais de 1000 cenários distintos. As médias dos resultados obtidos têm um erro máximo de 5% para mais ou para menos, com um nível de confiança de 95%. A Figura 3.1 mostra o gráfico *box plot* dos tempos de execução das aplicações considerando cada uma das heurísticas para alocação de arquivos apresentadas neste trabalho.

3.4.3 Resultados

Nesta seção são analisados os resultados obtidos nas simulações realizadas com o objetivo de confrontar as heurísticas de alocação de arquivos apresentadas neste trabalho. Devido à natureza não-dedicada dos recursos, as heurísticas de alocação de arquivos utilizaram informações históricas sobre a disponibilidade das máquinas. Com isso, as simulações tiveram como foco principal mensurar o impacto que a disponibilidade das máquinas causa no tempo de execução das aplicações em relação a cada heurística avaliada. Além disso, foi avaliado como o nível de replicação dos arquivos pode influenciar o tempo de execução das aplicações

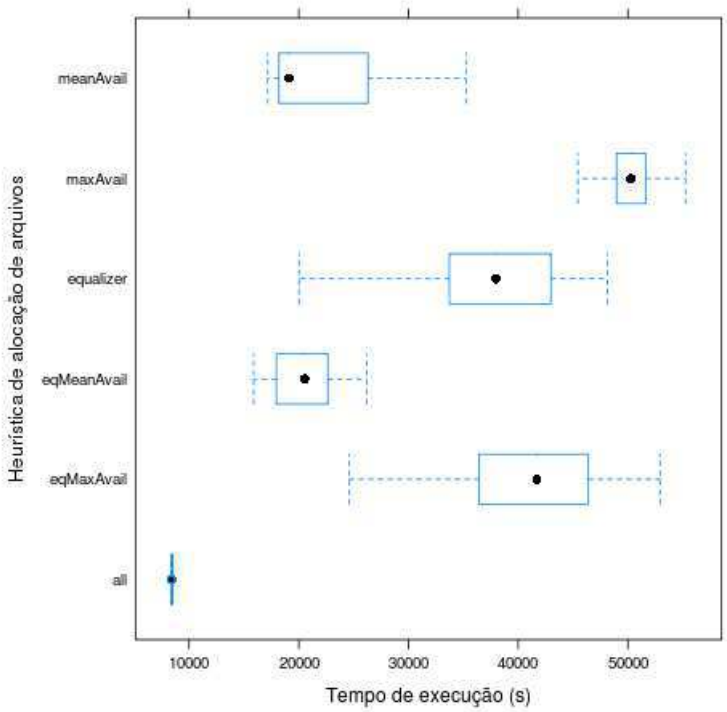

Figura 3.1: Gráfico box plot das médias do tempo de execução das aplicações considerando cada uma das heurísticas para alocação de arquivos.

e a quantidade de recursos de armazenamento utilizado.

Tempo de execução das aplicações

Nesta seção é realizada a análise da influência da disponibilidade das máquinas no tempo de execução das aplicações considerando cada uma das heurísticas de alocação de arquivos avaliadas. A Figura 3.2 mostra a comparação dos tempos médios de execução das aplicações em função do nível de replicação para cada uma das heurísticas.

De acordo com os resultados mostrados, o tempo de execução das aplicações varia conforme o nível de replicação dos arquivos, como esperado, exceto para a heurística all que, para cada arquivo, cria cópias em todas as máquinas ignorando o nível de replicação. Por isso, esta heurística apresenta tempos de execução praticamente constantes, independentemente do nível de replicação utilizado. Porém, este desempenho custa caro, devido à grande quantidade de espaço de armazenamento necessária para armazenar as cópias dos arquivos

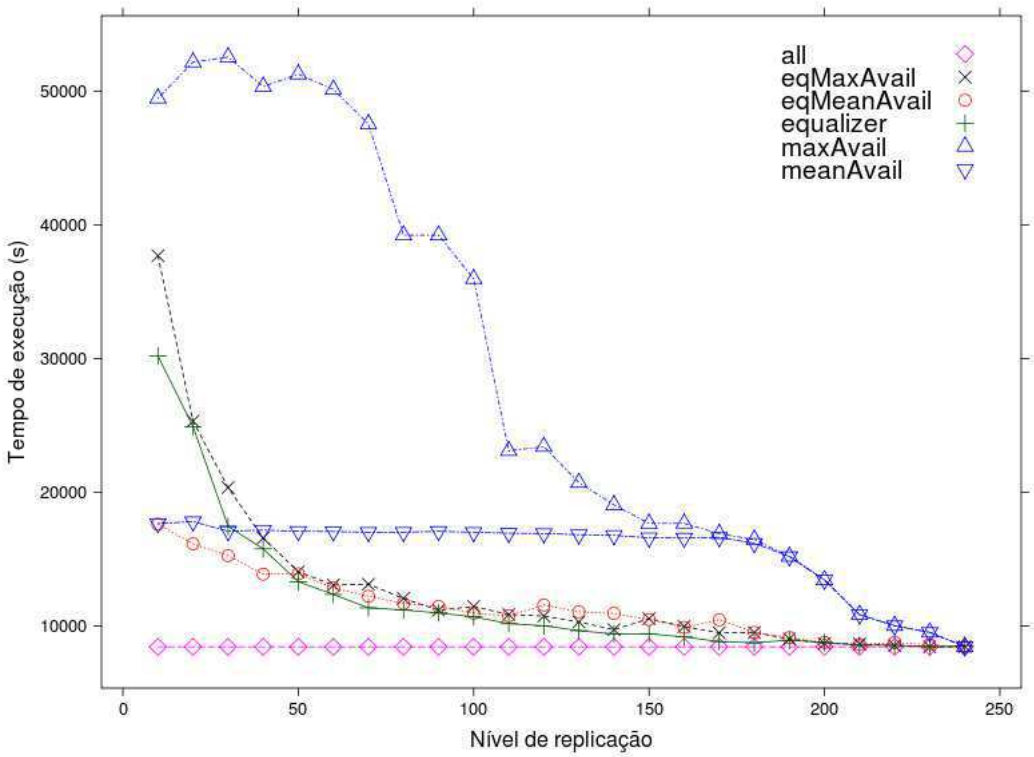

Figura 3.2: Comparação das heurísticas de alocação de arquivos em relação ao tempo de execução das aplicações em função do nível de replicação.

em todas as máquinas. A heurística de alocação de arquivos padrão do sistema de arquivos BeeFS, $equalizer$, não possui bons resultados para pequenos níveis de replicação. Contudo, ela apresenta ganhos significativos de desempenho quando o nível de replicação possui valores acima de 20% do número total de máquinas. Comportamento similar pode ser percebido na heurística $eqMaxAvail$. Porém, as heurísticas $meanAvail$ e $eqMeanAvail$ apresentam ganhos de desempenho relevantes sem necessitar de altos valores de replicação, logo, elas são mais indicadas para cenários onde espaço de armazenamento não é abundante. Entretanto, vale salientar que, ao contrário da heurística $eqMeanAvail$, a heurística $meanAvail$ não reduz o tempo de execução das aplicações à medida que o nível de replicação aumenta. Por fim, a heurística $maxAvail$ apresenta um péssimo desempenho quando comparada às demais heurísticas. Este fenômeno é facilmente explicado pelo fato de que as máquinas mais disponíveis armazenam uma quantidade maior de cópias dos arquivos. Deste modo, uma vez que poucas máquinas são responsáveis por manter uma grande quantidade de cópias, são

criados $hotspots$ durante a execução das aplicações.

Espaço de armazenamento

Nesta seção é avaliada a quantidade de armazenamento utilizado para manter as cópias dos arquivos considerando cada uma das heurísticas de alocação de arquivos avaliadas. Uma vez que a heurística all armazena uma cópia de cada arquivo em todas as máquinas, o espaço de armazenamento utilizado por esta heurística é constante, indepedentemente do nível de replicação. Por outro lado, não surpreende que as demais heurísticas apresentam a mesma demanda por espaço de armazenamento em função do nível de replicação. Mais especificamente, a quantidade de armazenamento destas heurísticas é diretamente proporcional ao nível de replicação.

Capítulo 4

O sistema de arquivos BeeFS

Em ambientes onde os usuários estão conectados através de uma rede local (LAN) é importante que os arquivos dos usuários estejam acessíveis a partir de quaisquer máquinas que façam parte da rede. Além disso, os usuários devem poder compartilhar arquivos com outros usuários que participam de uma mesma rede. Sistemas de arquivos que gerenciam recursos de armazenamento através de uma rede são chamados sistemas de arquivos distribuídos.

Sistemas de arquivos distribuídos têm sido amplamente adotados para realizar o compartilhamento de arquivos entre os usuários pertencentes a uma mesma rede. No entanto, os sistemas de arquivos distribuídos mais utilizados na prática, como NFS [PJS+94] e Coda [SKK+90], possuem algumas desvantagens. O NFS usa uma arquitetura cliente-servidor, a qual obviamente limita a escalabilidade. Adicionalmente, lidar com o aumento repentino na demanda de capacidade do sistema de arquivos — normalmente desencadeada pela chegada de novos usuários — é caro e complicado, uma vez que envolve a aquisição de mais discos para armazenamento, tornar o sistema indisponível para que seja realizada a migração de dados e, possivelmente, mudanças nas atividades de administração — por exemplo, para acomodar novos procedimentos de backup. Por outro lado, Coda permite que vários servidores compartilhem a demanda e simplifica o procedimento para o crescimento da capacidade, porém, isto vem com um aumento substancial nos custos administrativos de manutenção do sistema e não traz redução no custo de crescer a capacidade de armazenamento do sistema.

Este capítulo apresenta o Beehive File System (BeeFS), um sistema de arquivos distribuído desenvolvido para atender a requisitos de escalabilidade e manutenabilidade, não

oferecidos pelos sistemas de arquivos distribuídos mais usados hoje. O BeeFS utiliza o espaço não utilizado das estações de trabalho para compor um sistema de arquivos com visão global e acesso transparente para os arquivos, os quais estão armazenados nos discos das máquinas participantes. Essa abordagem para o armazenamento é desejável e viável, uma vez que a capacidade dos discos rígidos atuais superou as necessidades individuais de muitos usuários, deixando-os com muito espaço de armazenamento livre em suas estações de trabalho [DB99]. Por utilizar a infraestrutura subutilizada já existente, o BeeFS é capaz de fornecer uma solução de armazenamento distribuído mais barata do que as soluções existentes. Além disso, a necessidade de incrementar a capacidade de armazenamento causada pela chegada de novos usuários é automaticamente satisfeita quando se utiliza o BeeFS. Isso é facilmente justicável, pois em virtude da chegada de novos usuários, geralmente, novas estações de trabalho são necessárias, e, consequentemente, mais espaço em disco estará disponível.

4.1 Arquitetura

A arquitetura do BeeFS consiste em três componentes principais: um único servidor de metadados (*queenbee*), servidores de dados (*honeycombs*) e clientes (*honeybees*). Estes componentes seguem um modelo de distribuição híbrido que mistura aspectos de sistemas cliente-servidor e entre-pares, o que permite i) reduzir a carga de trabalho no componente centralizado e ii) atender o aumento da demanda de forma mais granular, através da adição de novos servidores de dados. O servidor de metadados e os servidores de dados fornecem serviços para vários clientes como é ilustrado na Figura 4.1.

O servidor de metadados é um componente confiável que deve ser instalado, preferencialmente, em uma máquina dedicada. Além do armazenamento de metadados, o servidor de metadados é responsável pelo serviço de descoberta de servidores de dados, controle de acesso e pela coordenação do mecanismo de replicação de dados. O serviço de descoberta mapeia os nomes dos arquivos para os servidores de dados que os armazenam. Dessa forma, os clientes podem acessar os arquivos sem tomar conhecimento do local onde eles estão realmente armazenados. Após descobrir a localização de um arquivo, os clientes contactam os servidores de dados para realizar operações de escrita/leitura. Contudo, o servidor de

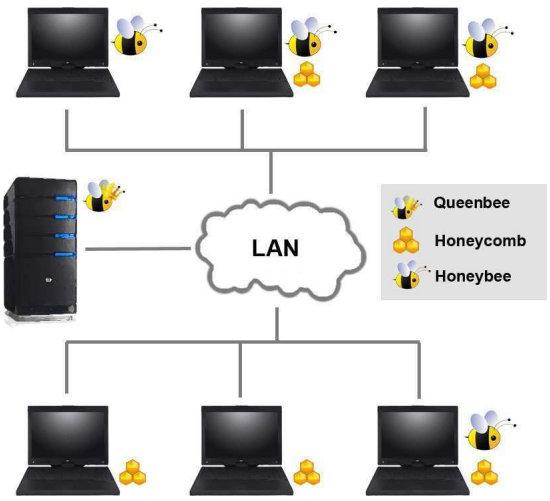

Figura 4.1: Componentes do BeeFS

metadados não participa, em hipótese alguma, do processo de transferência de dados. Transferências de dados são realizadas exclusivamente entre os servidores de dados e os clientes.

Os servidores de dados são componentes simples que armazenam colaborativamente as cópias dos arquivos servidos pelo BeeFS. Ao contrário do servidor de metadados, os servidores de dados não residem em máquinas dedicadas, mas em estações de trabalho que fazem parte de uma rede local. Os servidores de dados disponibilizam apenas primitivas básicas de escrita e leitura. Estas primitivas são utilizadas pelos clientes para realizar leituras e modificações nos arquivos, respectivamente; ou por outros servidores de dados, quando precisam atualizar o conteúdo das cópias dos arquivos.

Os processos dos usuários têm acesso aos arquivos por meio de uma interface POSIX implementada pelo cliente BeeFS (honeybee). Dessa forma, os clientes provêm acesso transparente aos arquivos armazenados pelo sistema. Geralmente, clientes e servidores de dados coexistem em uma mesma máquina. Assim, o algoritmo de alocação de arquivos padrão do BeeFS explora essa possibilidade para fins de melhoria de desempenho.

4.2 Tolerância a falhas

Considerando a natureza distribuída do BeeFS, fica evidente a necessidade de mecanismos que matenham a consistência e a disponibilidade do sistema perante a ocorrência de falhas. No BeeFS, há mecanismos para tolerar falhas tanto nos servidores de dados quanto no servi-

dor de metadados. Os clientes não precisam de mecanismos de tolerância a falhas, já que eles não armazenam nenhum tipo de informação.

4.2.1 Replicação de arquivos

Assim como em outros sistemas de arquivos distribuídos, um dos mecanismos para tolerância a falhas no BeeFS é a replicação. O BeeFS implementa uma estratégia de replicação passiva não-bloqueante [SPSB09]. Segundo esse modelo, ilustrado na Figura 4.2, os dados originais são denominados primários, enquanto as cópias são denominadas de dados secundários ou réplicas. Logo, cada arquivo possui um grupo de replicação, ou seja, uma cópia primária e um conjunto de cópias secundárias. Operações de escrita de dados são realizadas imediatamente apenas nos servidores de dados que mantêm as cópias primárias. Posteriormente, as atualizações dos arquivos são propagadas para as cópias secundárias. A atualização, isto é, a propagação das alterações nos dados primários para as réplicas, é realizada de modo não-bloqueante. Dessa forma, o cliente não precisa aguardar a conclusão da atualização.

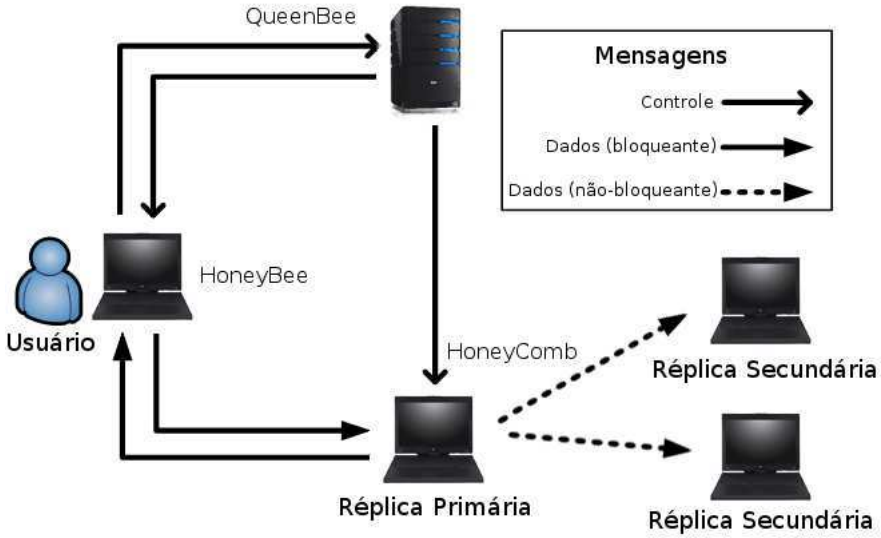

Figura 4.2: Modelo de Replicação BeeFS

O servidor de metadados é responsável pela coordenação das atualizações das réplicas secundárias. Isso mantém consistente o estado do grupo de replicação associado a cada arquivo do sistema. Um arquivo é considerado consistente se todas as suas réplicas possuirem

o mesmo número de versão. Assim, cada réplica armazenada nos servidores de dados possui uma versão. Esta versão é atualizada sempre que o cliente envia para o servidor de metadados uma chamada para fechamento de um arquivo que foi modificado. No final da chamada, o servidor de metadados agenda um processo de propagação do conteúdo da réplica primária para as demais. O tempo compreendido entre o fechamento do arquivo e a execução da propagação do conteúdo é conhecido como "tempo para coerência".

O servidor de metadados é, também, responsável pelo monitoramento dos servidores de dados. Quando é detectada a falha de um servidor de dados, os grupos de replicação associados a este servidor devem ser reorganizados. Assim, outros servidores de dados devem substituir aquele que foi comprometido. Dessa forma, cada grupo de replicação permanece com a mesma quantidade de réplicas que existiam antes de ocorrer a falha.

4.2.2 Replicação de metadados

Com relação ao servidor de metadados, existe a possibilidade de ocorrer dois tipos de falhas: transientes e permanentes. São consideradas falhas transientes aquelas que não comprometem a integridade do sistema, mas tornam o serviço indisponível. Normalmente, estas falhas são causadas por queda da força elétrica. Neste caso, serviços de monitoramento podem ser utilizados para automaticamente reiniciar o servidor de metadados.

Por outro lado, as falhas permanentes são causadas por defeitos de hardware e/ou de software que afetem o armazenamento dos metadados. Estas falhas comprometem definitivamente o servidor de metadados, fazendo com que todas as referências para os arquivos armazenados no sistema sejam perdidas, tornando-os inacessíveis. O BeeFS possui um modelo de armazenamento de metadados tolerante a falhas permanentes. De acordo com esse modelo, os metadados armazenados no servidor de metadados também são mantidos pelos servidores de dados.

Para se recuperar desse tipo de falha, o BeeFS possui um modo de recuperação de falha do sistema [SPSB09]. Ou seja, na ocasião de um falha que danifique a persistência dos metadados, o servidor precisa ser iniciado no modo de recuperação. Geralmente, o servidor de metadados é iniciado em modo de operação normal, no qual os metadados são carregados a partir do disco. No modo de recuperação, ao invés de carregar as informações dos metadados a partir do disco, o servidor de metadados recebe, como parâmetro de entrada, a

lista de endereços dos servidores de dados que constituíam o sistema. Então, o servidor de metadados inicia o processo de recuperação.

O servidor de metadados contata cada um dos servidores de dados presentes na lista e requisita o conjunto de nós-i e de atributos estendidos. O conjunto original de nós-i e de atributos estendidos é obtido através de uma simples operação de junção dos conjuntos de metadados fornecidos pelos servidores de dados. Os grupos de replicação são inferidos a partir da localização das réplicas nos metadados. A estrutura de diretórios é remontada sob demanda, a partir das cadeias de caracteres armazenadas nos conjuntos de atributos estendidos recuperados. Desta forma, é possível reconstituir o estado no qual o sistema se encontrava antes da falha ocorrer para a maioria dos casos [SPSB09].

4.3 Metadados

No BeeFS, as operações que envolvem metadados são executadas pelo servidor de metadados. Os metadados representam o estado do sistema. Eles são utilizados para diferentes propósitos, como armazenar informações básicas sobre cada arquivo (tamanho, dono, grupo, permissões de acesso), manter a localização dos arquivos e, até mesmo, controlar o comportamento do sistema.

Por razões de desempenho, o servidor de metadados adota uma política semelhante ao GFS [GGL03], mantendo os metadados armazenados na memória e persistindo periodicamente as atualizações para o disco por motivos de tolerância a falhas. Esta estratégia é viável já que a quantidade de espaço de armazenamento necessária para manter os metadados é muito pequena, como sugerido por um estudo sobre o conteúdo do sistema de arquivos [DB99]. Considerando que um sistema armazena aproximadamente 1 milhão de arquivos e que 100 bytes por arquivo são suficientes para armazenar os metadados, apenas 100 Mbyte é necessário para armazenar todos os metadados do sistema.

4.3.1 Tipos de metadados

Os metadados armazenados pelo BeeFS podem ser classificados em três categorias: nós-i, estruturas de dados do sistema e atributos estendidos.

Os nós-i mantêm informações referentes aos arquivos armazenados pelo sistema que

incluem: tamanho do arquivo, proprietário, data de criação, data da última modificação, diretório pai, entre outros.

As estruturas de dados do sistema mantêm informações referentes à estrutura de diretórios, aos grupos de replicação e ao conjunto de servidores de dados. A estrutura de diretórios consiste em uma árvore contendo os diretórios e arquivos dos usuários. Todos os usuários têm a mesma visão da árvore de diretórios. Os grupos de replicação mapeiam a localização dos servidores de dados que armazenam as réplicas de cada arquivo. Além disso, o sistema mantém informações acerca dos servidores de dados. A partir dessas informações, o servidor de metadados é capaz de realizar o monitoramento dos servidores de dados e alimentar os algoritmos de alocação de arquivos — os quais necessitam conhecer quanto espaço disponível cada servidor de dados possui, por exemplo.

Assim como outros sistemas [SNAKA$^+$08] [FHC08], o BeeFS utiliza-se de metadados para que outras aplicações customizem o comportamento de algumas partes do sistema. Esta customização pode ser controlada por meio dos atributos estendidos que são suportados pelo padrão POSIX. Estes atributos são pares nome/valor associados a um arquivo. Comumente, eles são utilizados pelo núcleo do sistema operacional para armazenar objetos do sistema, como a lista de controle de acesso de um arquivo, e por processos de usuário para armazenar informações arbitrárias (ex. a codificação do conteúdo de um arquivo). O BeeFS utiliza os atributos estendidos para armazenar informações adicionais sobre os arquivos. Essas informações são úteis para algumas aplicações, bem como para o próprio BeeFS. O BeeFS faz uso dos atributos estendidos para armazenar informações sobre o nível de replicação dos arquivos, o tempo para coerência e a dispersão de arquivos. Além disso, os atributos estendidos podem ser utilizados para fornecer informações sobre o estado atual do sistema.

O nível de replicação e o tempo para coerência são informações importantes para o mecanismo de replicação de arquivos. Estes atributos afetam, respectivamente, a capacidade de armazenamento do sistema e o consumo de banda passante da rede. O mecanismo de replicação utiliza o nível de replicação para determinar o número de cópias secundárias de um arquivo. O tempo para coerência é utilizado para estabelecer o tempo decorrido entre a última modificação de um arquivo e a realização do processo de atualização das cópias secundárias. Assim, uma vez que os atributos estendidos são definidos no nível de arquivo, é possível configurar que arquivos importantes terão um número maior de réplicas e serão

atualizados mais rapidamente do que os demais, por exemplo.

Os atributos estendidos tornam possível que o BeeFS dê suporte a alguns recursos mais avançados. Um desses recursos é a dispersão de arquivos. Como o principal objetivo do BeeFS é prover compartilhamento de arquivos em uma LAN, o algoritmo de alocação de arquivos padrão tenta manter os arquivos o mais próximo possível dos clientes que os acessam, preferencialmente, na mesma máquina. Porém, esse comportamento não é adequado para o processamento paralelo. Neste caso, o BeeFS permite que os usuários marquem diretórios como dispersos. Ou seja, o usuário pode associar um atributo estendido a um diretório para marcá-lo como um diretório de dispersão de arquivos. Desta forma, todos os arquivos nesse diretório serão alocados conforme outro algoritmo de alocação de dados, o qual espalha as cópias do arquivo, inclusive a cópia primária, nas estações de trabalho que fazem parte do sistema. Para aumentar as chances de que os arquivos localizados em diretórios de dispersão estarão disponíveis para processamento em estações de trabalho ociosas, o algoritmo para dispersão de arquivos será implementado com base nas heurísticas para alocação de arquivos elaboradas neste trabalho.

Os atributos estendidos também são utilizados pelo BeeFS para fornecer informações sobre o estado de algumas estruturas de dados do sistema, de forma semelhante ao *proc* file system do Unix [Bac86]. Estes atributos estendidos não podem ser modificados pelos usuários. Eles servem apenas para prover informações importantes sobre os arquivos, como o número de versão e a localização de um arquivo. Diferentes aplicações podem utilizar essas informações. Por exemplo, a partir desses atributos, aplicações que realizam processamento paralelo podem descobrir a localização de um conjunto de arquivos e, então, disparar remotamente processos nas máquinas que o armazenam. Desta forma, não há necessidade de transferir dados pela rede. O próximo Capítulo explica com mais detalhes como utilizar o BeeFS para realizar processamento paralelo de forma eficiente.

Capítulo 5

Processamento paralelo com BashReduce usando o BeeFS como substrato de armazenamento

A intrínseca distribuição dos arquivos, juntamente com a facilidade de customização e acesso a informações acerca do sistema provida pelos metadados, possibilita que algumas aplicações possam utilizar o BeeFS como substrato para executar processamento paralelo de forma simples e eficiente. As melhorias no desempenho destas aplicações podem ser ainda mais significativas, desde que envolvam o processamento de grandes quantidades de dados. Neste capítulo é apresentado um estudo de caso que demonstra a viabilidade e a praticabilidade de utilizar o BeeFS como camada de armazenamento para aplicações de uso intensivo de dados.

5.1 Introdução

Visando demonstrar a viabilidade de se utilizar o BeeFS como substrato para a execução de aplicações de uso intensivo de dados, será apresentado um estudo de caso no qual se explora a distribuição dos dados no BeeFS para aumentar substancialmente o ganho de desempenho do aplicativo BashReduce [Fre09]. BashReduce é uma ferramenta simples que facilita a execução de aplicações que processam dados conforme o modelo Map/Reduce [DG08], utilizando um conjunto de estações de trabalho conectadas por uma rede local. Porém, ao

contrário de outras implementações do modelo Map/Reduce, BashReduce não foi desenvolvido considerando uma camada de armazenamento distribuído. Logo, o desempenho das aplicações BashReduce é bastante prejudicado, principalmente pela transferência de dados pela rede.

A partir deste estudo de caso, ficará evidente como uma simples modificação no algoritmo de escalonamento do BashReduce permitirá que ele apresente consideráveis ganhos de performance por explorar a natureza distribuída do BeeFS. Para mensurar este ganhos de performance, foram realizados uma série de experimentos. Nestes experimentos foram auferidos ganhos de desempenho que variam entre 17 e 42 vezes, comparando-se a execução de aplicações onde os arquivos são armazenados em um sistema de arquivos NFS (abordagem normalmente utilizada pelos usuários BashReduce) com aquelas que usam o BeeFS como alternativa. Contudo, a metodologia utilizada para a realização destes experimentos fez algumas simplificações, mais especificamente, ela assume que as estações de trabalho são dedicadas, fato este que não condiz com a realidade. O impacto dessas simplificações, bem como algumas alternativas para eliminá-lo são discutidas no final deste capítulo.

5.2 Map/Reduce

Map/Reduce [DG08] é um modelo de programação proposto pela Google para executar aplicações de uso intensivo de dados. Este modelo de programação está se tornando bastante utilizado para processar grandes volumes de dados com as mais diversas finalizadades, como mineração de dados, processamento de linguagem natural e indexação de conteúdo da Web [Pow10]. Esta popularidade deve-se à facilidade de desenvolvimento das aplicações e à escalabilidade e robustez do sistema, o qual é formado por *cluster* de máquinas convencionais.

Uma aplicação Map/Reduce é definida com base em duas funções: uma função *map* que recebe os dados de entrada e produz um conjunto intermediário de valores, e uma função *reduce* que processa os valores intermediários produzidos pela função *map* para gerar o produto final da aplicação. A partir destas funções, a implementação Map/Reduce automaticamente cria e distribui as tarefas para execução entre as máquinas disponíveis, tentando evitar o tráfego de dados através da rede que conecta as máquinas. Além disso, ele é o respon-

sável pelo particionamento dos dados de entrada, gerenciamento da comunicação entre as máquinas e tratamento de falhas. Desta forma, ele pode melhorar a produtividade dos desenvolvedores, já que estes poderão concentrar seus esforços apenas na lógica da aplicação em si, sem se preocupar com os detalhes da programação distribuída.

Atualmente, as duas implementações mais conhecidas do modelo Map/Reduce são a versão proprietária desenvolvida pela Google, e o Hadoop [Had10], uma implementação *opensource* mantida pela Apache Software Foundation. A implementação desenvolvida pela Google utiliza o Google File System (GFS) [GGL03] como infraestrutura de armazenamento, enquanto o Hadoop é suportado pelo Hadoop Distributed File System (HDFS) [HDF10]. Ambas as implemetações foram desenvolvidas para executar em *clusters* de máquinas convencionais, onde cada máquina compartilha a capacidade de armazenamento local disponível com o sistema de arquivos distribuído para armazenar os arquivos que serão processados.

Com a crescente popularidade do modelo de programação Map/Reduce, muitos esforços foram aplicados na tentativa de portá-lo para outras arquiteturas [RRP+07; HFL+08] e explorá-lo de outras formas para atender às necessidades de outras aplicações [SC08; JB09]. Seguindo esta tendência, Erik Frey desenvolveu uma implementação bastante simples do modelo Map/Reduce para sistemas compatíveis com a especificação POSIX [TW06], chamado BashReduce [Fre09]. BashReduce foi desenvolvido visando simplicidade de uso, e impõe poucos requesitos em relação à infraestrutura utilizada. Mais especificamente, ele não assume a existência de um sistema de arquivos distribuído. Evidentemente, esta característica limita substancialmente o desempenho das aplicações BashReduce.

5.3 BashReduce

BashReduce é uma implementação baseada em *bash script* do modelo de programação Map/Reduce para sistemas compatíveis com a especificação POSIX [TW06]. Ele possibilita a execução de aplicações Map/Reduce de forma simples utilizando apenas um conjunto de estações de trabalho conectadas por uma rede local. Portanto, ele é bastante útil quando uma infraestrutura de *cluster* não está disponível. Além disso, o usuário pode utilizar as ferramentas do shell Unix (tais como: `sort`, `awk`, `grep`, `cut`, `paste` entre outras), para descrever

tanto a função *map* quanto a função *reduce* das aplicações BashReduce. Sendo, portanto, mais simples do que realizar a sequência de passos necessários para executar aplicações com as implementações mais populares do modelo Map/Reduce. Normalmente, o fluxo para executar uma aplicação Map/Reduce consiste em escrever um programa, compilá-lo e, finalmente, submetê-lo para execução. Além disso, ainda que uma infraestrutura Map/Reduce esteja disponível, se os dados a serem processados não foram gerados no sistema de armazenamento específico do framework Map/Reduce em uso (e.g. HDFS para sistemas Hadoop), talvez seja mais interessante utilizar o aplicativo BashReduce para processar estes dados sem a necessidade de transferi-los para um outro sistema de arquivos.

BashReduce transparentemente gerencia o particionamento dos dados, paraleliza a computação, coleta os resultados produzidos, juntando-os para produzir o resultado final. Contudo, o aplicativo BashReduce não possui suporte para tolerância a falhas. A única forma disponível para identificar erros ocorridos durante a execução de uma aplicação é por meio da análise dos logs gerados a partir da saída de erro padrão (`stderr`) dos processos remotos. Embora arquivos de logs sejam úteis para a identificação de erros, a análise deve ser realizada de forma manual pelo usuário. Logo, BashReduce não é indicado para executar aplicações que necessitam de várias horas para terminar o processamento dos dados. Mesmo assim, BashReduce ainda é uma ferramenta que pode ajudar em tarefas diárias e, até mesmo, para realizar análise dos logs de outras aplicações [Cro09].

Com BashReduce, o usuário pode utilizar como entrada um único arquivo ou um diretório — neste último caso, todos os arquivos localizados no diretório serão processados — para produzir um único arquivo com o resultado da computação realizada. Uma aplicação BashReduce é expressa com alguns poucos argumentos, dos quais, os mais importantes são o arquivo de entrada, o arquivo de saída, a lista de máquinas que serão utilizadas para realizar o processamento e os programas *map* e *reduce*. O Código 5.1 mostra a linha de comando de uma simples aplicação BashReduce:

Código Fonte 5.1: Exemplo de uma aplicação BashReduce simples

```
$> br -i input.txt -o output.txt \
    -h "host1 host2 host3 host4" \
    -m "grep pattern" -r "uniq -c"
```

Na aplicação BashReduce mostrada no Código 5.1, o BashReduce irá dividir o ar-

quivo `input.txt` e enviar os pedaços para as máquinas especificadas pela opção "-h" (`host1`, `host2`, `host3` e `host4`). Em cada uma destas máquinas o programa *map* (`grep pattern`) vai processar o fragmento do arquivo de entrada enviado para a respectiva máquina. A saída produzida pelos programas *map* servem de entrada para o programa *reduce* (`uniq -c`), cujos resultados produzidos serão armazenados no arquivo `output.txt` na máquina que disparou o comando BashReduce.

Mais detalhadamente, quando o usuário executa um comando BashReduce, ocorre a seguinte sequência de ações:

1. O aplicativo BashReduce usa o comando *netcat* (`nc`) para preparar os canais de comunicação com cada máquina especificada pelo usuário (*slaves*); durante a computação, cada máquina *slave* deve possuir duas conexões abertas; enquanto uma delas recebe dados para serem processados, a outra conexão envia os dados produzidos para a máquina onde o comando BashReduce foi executado (*master*);

2. Os programas *map* e *reduce* são disparados remotamente nas máquinas *slave* através do comando *secure shell* (`ssh`); e ficam aguardando pelos dados de entrada enviados pela máquina *master* para iniciar o processamento;

3. Os arquivos de entrada são particionados em H fragmentos, onde H é o número de máquinas *slaves* especificadas pelo usuário; após o particionamento, o aplicativo BashReduce inicia a transferência dos fragmentos para cada uma das máquinas *slaves*;

4. À medida que as máquinas *slaves* completam o processamento dos programas *map* e *reduce*, elas enviam os dados produzidos para a máquina *master*, onde o aplicativo BashReduce concatena todos os resultados produzidos para produzir um único arquivo — por padrão, a concatenação é realizada pelo programa *sort*, mas o usuário pode especificar outro programa para realizar a concatenação.

Infelizmente, esta técnica transfere dados da máquina *master* para as máquinas *slaves* através da rede local. Logo, o tempo de execução da aplicação é prologando devido ao gargalo criado pela transferência dos dados. Porém, considerando que os arquivos de entrada estão armazenados em um sistema de arquivos distribuído, e.g. NFS, e são acessíveis em todas as máquinas por meio de uma hierarquia de arquivos global, então o BashReduce pode

ser executado de forma mais eficiente. Ao invés de particionar os arquivos e enviar os dados pela rede local, o usuário pode evitar a transferência desnecessária dos dados utilizando um sistema de arquivos distribuído. Neste caso, em vez de armazenar os dados em si, o arquivo de entrada deve especificar a lista de arquivos a serem processados. Adicionalmente, o usuário precisa passar um parâmetro extra que faz com que o BashReduce distribua o nome dos arquivos listados no arquivo de entrada ao invés de distribuir os dados em si pela rede. Contudo, se o sistema de arquivos distribuído utilizado baseia-se em uma arquitetura cliente-servidor, o gargalo da transferência de dados pela rede continuará a existir, porém o gargalo não será mais a máquina que disparou o comando BashReduce, mas o próprio servidor centralizado de arquivos. Em seguida, será demonstrado como o sistema de arquivos BeeFS pode ser utilizado para assegurar melhores ganhos de desempenho em aplicações BashReduce.

5.4 Utilizando o BeeFS como substrato de armazenamento do BashReduce

Como mostrado anteriormente, o aplicativo BashReduce pode ser utilizado de duas formas distintas. Na primeira, os dados de entrada são divididos e enviados a partir da máquina *master* para as máquinas *slaves* utilizando a rede local. Na segunda, apenas os nomes dos arquivos de entrada são enviados, o que reduz drasticamente a quantidade de dados transferidos pela rede. Contudo, esta última técnica necessita que as máquinas tenham acesso aos arquivos por meio de um sistema de arquivos distribuído que obedeça à semântica POSIX e possua uma hierarquia de arquivos global, assim como o NFS ou o BeeFS. Desta forma, os dados podem ser obtidos pelas máquinas *slaves* sem a interferência da máquina *master*. Portanto, se o usuário dispõe de um sistema de arquivos distribuído compatível com a semântica POSIX, a segunda técnica produz melhores resultados em termos de tempo de execução da aplicação, uma vez que o tráfego de dados na rede é minimizado.

A utilização de um sistema de arquivos distribuído que armazena os dados de forma distribuída, como o BeeFS, pode melhorar significativamente o desempenho das aplicações BashReduce. Este ganho de desempenho deve-se à redução da quantidade de dados transferidos por meio da rede local. Mas, para obter a vantagem da distribuição dos dados no BeeFS,

o programa BashReduce precisa descobrir a localização das máquinas onde os arquivos estão armazenados durante a execução de uma aplicação. Para resolver este problema, foram realizadas algumas modificações no programa BashReduce para que ele pudesse operar sobre o BeeFS de forma mais eficiente. Mais precisamente, a modificação consiste em acessar os atributos estendidos dos arquivos de entrada para descobrir onde os mesmos estão armazenados. Se o arquivo de entrada está armazenado no BeeFS, ele possui um atributo estendido chamado `beefs.replicationGroup`, o qual contém entre outras informações, o nome da máquina que armazena a réplica primária do arquivo em questão. Caso contrário, se este atributo não existir, significa que o arquivo não está armazenado em um sistema de arquivos BeeFS e o programa BashReduce continua o fluxo original de execução. A partir da localização das máquinas, o programa BashReduce poderá executar a aplicação diretamente nas máquinas que armazenam os arquivos de entrada, evitando a transferência de dados pela rede local.

Além disso, se todos os arquivos de entrada estiverem armazenados no sistema de arquivos BeeFS, torna-se desnecessário que o usuário informe a lista de máquinas a serem utilizadas para executar a aplicação, já que o programa BashReduce descobrirá o nome das máquinas em tempo de execução.

5.5 Avaliação de desempenho de aplicações BashReduce usando diferentes sistemas de arquivos distribuídos

Visando demonstrar os ganhos de desempenho das aplicações BashReduce usando o BeeFS, foi realizado um experimento comparando os tempos de execução de uma aplicação BashReduce considerando dois sistemas de arquivos distribuído para armazenar os arquivos de entrada, o BeeFS e o NFS. O sistema NFS foi escolhido por ser bastante utilizado para compartilhar arquivos em rede locais e representar a solução de armazenamento normalmente utilizada pelos usuários do programa BashReduce.

O experimento realizado é equivalente ao “Grep distribuído”, uma aplicação descrita no artigo original que descreve o modelo de programação Map/Reduce, o qual é uma aplicação representativa para processamento de grandes volumes de dados [DG08]. Para este experimento, os arquivos de entrada utilizados são arquivos de log das simulações de pesquisas

anteriores do nosso grupo de pesquisa envolvendo escalonamento de tarefas em grades computacionais. No experimento, a aplicação BashReduce deve localizar nos arquivos de entrada a ocorrência dos nomes dos eventos processados na simulação com o programa de *map*, enquanto o programa *reduce* contabiliza o número de ocorrências para cada evento. Adicionalmente, para gerar o resultado final, a aplicação executa um programa que foi desenvolvido para concatenar os resultados produzidos pelas máquinas *slaves* e produz o sumário do número de ocorrências para cada evento da simulação.

Para avaliar a escalabilidade de cada um dos sistemas de arquivos, os experimentos foram executados em três cenários distintos. Em cada cenário, o número de máquinas *slaves* utilizadas é proporcional à quantidade de dados a serem processados. Os cenários usaram 5, 10 e 20 máquinas *slaves* para processar $5GB$, $10GB$ e $20GB$ de dados brutos, respectivamente.

Os experimentos foram realizados usando estações de trabalho conectadas por uma rede Ethernet de $100Mbps$. Cada estação de trabalho possui um processador Intel Core 2 Duo $2.40GHz$ com $2GB$ de memória RAM e $100GB$ de espaço em disco, executando o sistema operacional Ubuntu 9.04 (kernel $2.6.28$). Além disso, tanto o servidor de metadados do BeeFS quanto o servidor NFS foram instalados em uma máquina com configuração superior, a qual possui um processador Quadcore - Intel(R) Xeon(R) CPU E5310 $1.60GHz$ com $4GB$ de memória RAM e $500GB$ de espaço em disco, executando o sistema operacional Debian 5 (kernel $2.6.26$) com arquitetura de 64-bits.

Os experimentos foram repetidos várias vezes, pouco mais de 50 execuções, para garantir um nível de confiança de 95% com base nas médias dos tempos de execução mensurados, e com uma taxa de erro inferior a 5%. A Figura 5.1 mostra a média dos tempos de execução deste experimento em função do número de máquinas *slaves* tanto para o BeeFS quanto para o NFS.

Como esperado, à medida que o número de máquinas *slaves* é incrementado, o tempo de execução das aplicações BashReduce que utilizaram NFS também aumenta. Este resultado é explicado pela transferência dos dados pela rede a partir de um servidor centralizado, o qual rapidamente torna-se um gargalo. Assim, o atraso causado pela transferência dos dados representa a maior parte do tempo de execução das aplicações.

Por outro lado, as aplicações que executam utilizando BashReduce sobre o BeeFS apresentam praticamente o mesmo desempenho em todos os cenários avaliados, demonstrando a

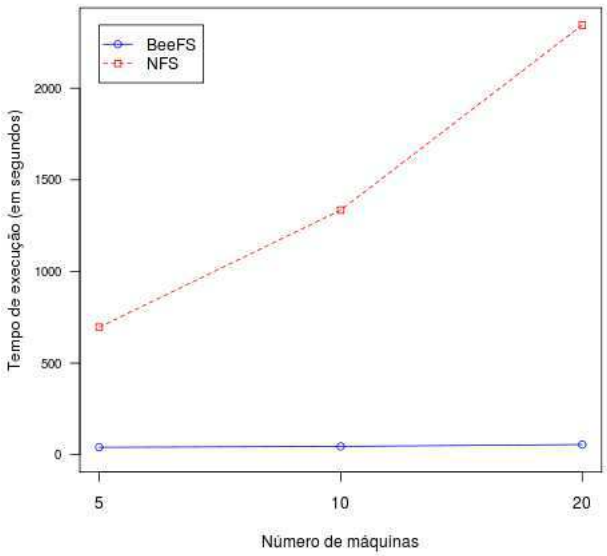

Figura 5.1: Tempo de execução (em segundos) para as aplicações BashReduce executando sobre o BeeFS e o NFS em função do número de máquinas *slaves* usadas.

escabilidade linear do sistema. Este grau de escalabilidade deve-se ao fato de que os dados são lidos diretamente dos discos locais das máquinas *slaves* em paralelo, reduzindo o tráfego da rede. Este resultado comprova a suposição de que uma arquitetura híbrida que utiliza servidores de dados distribuídos incrementa substancialmente o paralelismo ao passo que reduz o gargalo no servidor central.

5.6 Considerações

Este capítulo apresentou um estudo de caso que comprova os benefícios de utilizar o sistema de arquivos BeeFS como substrato para executar processamento paralelo de forma simples e eficiente.

No estudo de caso, foram realizadas pequenas alterações no aplicativo BashReduce, uma ferramenta que permite executar aplicações no estilo Map/Reduce de forma simples utilizando apenas as máquinas de uma LAN. Uma vez que o aplicativo BashReduce não considerava a existência de um sistema de armazenamento distribuído — diferentemente de outras implementações do modelo Map/Reduce — as modificações efetuadas tornaram possível

que ele pudesse usar o sistema de arquivos BeeFS como uma camada de armazenamento distribuído. Desta forma, o aplicativo BashReduce pode explorar a distribuição dos dados em um sistema BeeFS para melhorar o desempenho das aplicações.

Considerando os experimentos realizados, notou-se um expressivo ganho de desempenho nas aplicações BashReduce que utilizaram o sistema de arquivos BeeFS para armazenar os dados a serem processados. A otimização no desempenho das aplicações é resultado, principalmente, da arquitetura híbrida do sistema BeeFS. Esta arquitetura incrementa substancialmente o paralelismo do sistema já que os dados podem ser acessados simultaneamente a partir dos vários servidores de dados, evitando que o servidor central se torne um gargalo. Além disso, o aplicativo BashReduce obteve vantagem desta arquitetura para diminuir a quantidade de dados transferidos pela rede, reduzindo significativamente o tempo de execução das aplicações.

Contudo, vale salientar que as estações de trabalho que foram utilizadas nos experimentos estavam dedicadas à execução das aplicações. Ou seja, elas não estavam sendo utilizadas pelos usuários durante a realização dos experimentos. Porém, este cenário não condiz com a realidade, já que o sistema de arquivos BeeFS é formado por um conjunto de estações de trabalho em uma rede local, as quais podem está em uso pelo dono da máquina, já que não são recursos dedicados. Logo, fica claro a necessidade de métodos que levem em consideração a natureza não-dedicada dos recursos para garantir que aplicações possam ser executadas eficientemente sem interferir nas atividades normais dos usuários.

Como foi apresentado no capítulo 3, a interferência causada pela execução de aplicações pode ser facilmente resolvida, abortando-se a tarefa quando o usuário volta a utilizar a máquina. Porém, para evitar que uma grande quantidade de tarefas sejam canceladas e, portanto, prejudicando o desempenho das aplicações, pode-se utilizar heurísticas que consideram a disponibilidade das máquinas para decidir em quais máquinas as cópias de um arquivo devem ser armazenadas. Esta solução foi avaliada no capítulo 3, no qual descreve uma solução que baseia-se em heurísticas para a alocação de arquivos em um sistema de arquivos distribuído sobre estações de trabalho. Com base nestas heurísticas, é possível aumentar as chances de que os arquivos estarão disponíveis para processamento em estações de trabalho ociosas. Desta forma, viabiliza-se a execução de aplicações de uso intensivo de dados sobre o sistema de arquivos BeeFS de forma eficiente, sem prejudicar a experiência

do usuário.

Capítulo 6

Conclusões e Trabalhos Futuros

Esta dissertação mostra que é possível executar eficientemente aplicações de uso intensivo de dados sobre um sistema de arquivos distribuído POSIX usando recursos não-dedicados através de heurísticas para alocação de arquivos. Estas heurísticas consideram dados históricos sobre a utilização do sistema para decidir onde armazenar as réplicas de um arquivo que será utilizado para processamento. Desta forma, estas heurísticas tentam aumentar as chances de que os arquivos estarão disponíveis para processamento em máquinas ociosas. Além disso, este trabalho apresenta escalonadores de tarefas que evitam executar aplicações em máquinas que não estão ociosas. A utilização destes escalonadores juntamente com as heurísticas para alocação de arquivos reduzem drasticamente a inconveniência que a execução não-solicitada de aplicações podem levar a outros usuários.

Através de simulações foi constatado que as heurísticas com melhores desempenhos são aquelas que consideram alocar as réplicas dos arquivos nas máquinas que possuem os maiores períodos médios de disponibilidade, enquanto realizam o balanceamento das réplicas entre as máquinas que participam do sistema. Além disso, os resultados mostram que as heurísticas reduzem o impacto das transferências de dados no desempenho da aplicação uma vez que elas buscam aumentar as chances das aplicações realizarem leituras locais em máquinas ociosas.

Como trabalhos futuros, seria interessante investigar a utilização de técnicas de virtualização para controlar a quantidade de recursos que as aplicações podem consumir das estações de trabalho. Além disso, considerando que as máquinas com vários núcleos estão se tornando cada vez mais comuns, poderia-se averiguar a possibilidade de se executar mais de

uma tarefa em uma mesma máquina. Isto poderia permitir a redução do número de réplicas ou possibilitar a execução simultânea de um maior número de aplicações. Além disso, espera-se que a implementação dessas heurísticas no BeeFS ajude a indentificar aspectos que necessitam ser refinados.

Referências Bibliográficas

[AD76] Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 562–570, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[Bac86] Maurice J. Bach. *The design of the UNIX operating system*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[BDET00] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *SIGMETRICS '00: Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 34–43, New York, NY, USA, 2000. ACM.

[CDG+06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[CFS02] Lustre: A scalable, high-performance file system. Cluster File Systems Inc. white paper, version 1.0, November 2002. http://www.lustre.org/docs/whitepaper.pdf.

[CPC+03] Walfredo Cirne, Daniel Paranhos, Lauro Costa, Elizeu Santos-Neto, Francisco Brasileiro, Jacques SauvÃ©, FabrÃcio A. B. Silva, Carla O. Barros,

and Cirano Silveira. Running bag-of-tasks applications on computational grids: The mygrid approach. *Parallel Processing, International Conference on*, 0:407, 2003.

[Cro09] Richard Crowley. "building opendns stats"at velocity. `http://rcrowley.org/2009/06/23/building-opendns-stats-at-velocity.html`, June 2009.

[DB99] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. In *SIGMETRICS '99: Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 59–70, New York, NY, USA, 1999. ACM.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DH07] Thomas H. Davenport and Jeanne G. Harris. *Competing on Analytics: The New Science of Winning*. Harvard Business School Press, Boston, MA, USA, March 2007.

[FHC08] Gilles Fedak, Haiwu He, and Franck Cappello. Bitdew: a programmable environment for large-scale data management and distribution. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[Fre09] Erik Frey. Mapreduce bash script. `http://blog.last.fm/2009/04/06/mapreduce-bash-script`, April 2009.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS - Operating Systems Review*, 37(5):29–43, 2003.

[GNA+98] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. *SIGOPS Oper. Syst. Rev.*, 32(5):92–103, 1998.

[Had10] Hadoop. `http://hadoop.apache.org`, March 2010.

[HDF10] Hadoop distributed file system (hdfs). `http://hadoop.apache.org/hdfs`, March 2010.

[HFL+08] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269, New York, NY, USA, 2008. ACM.

[HTT09] Anthony J. G. Hey, Stewart Tansley, and Kristin M. Tolle. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, Redmond, Washington, 2009.

[JB09] Chao Jin and Rajkumar Buyya. Mapreduce programming model for .net-based cloud computing. In Henk Sips, Dick Epema, and Hai-Xiang Lin, editors, *Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, volume 5704, pages 417–428, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[KTI+04] Derrick Kondo, Michela Taufer, Charles L. Brooks III, Henri Casanova, and Andrew A. Chien. Characterizing and evaluating desktop grids: An empirical study. *Parallel and Distributed Processing Symposium, International*, 1:26b, 2004.

[LO08] Huan Liu and Dan Orban. Gridbatch: Cloud computing for large-scale data-intensive batch applications. In *CCGRID '08: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, pages 295–305, Washington, DC, USA, 2008. IEEE Computer Society.

[ORS+08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international con-*

ference on Management of data, pages 1099–1110, New York, NY, USA, 2008. ACM.

[PDGQ05] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with sawzall. *Sci. Program.*, 13(4):277–298, 2005.

[PJS+94] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3 design and implementation. In *Proceedings of the Summer USENIX Conference*, pages 137–152, 1994.

[Pow10] Applications powered by hadoop. `http://wiki.apache.org/hadoop/PoweredBy`, March 2010.

[PSSB10] Thiago Emmanuel Pereira, Alexandro S. Soares, Jonhnny Weslley Silva, and Francisco Vilar Brasileiro. Beefs: A cheaper and naturally scalable distributed file system for corporate environments. Technical report, Federal University of Campina Grande - Distributed Systems Lab, 2010.

[RF02] Kavitha Ranganathan and Ian Foster. Decoupling computation and data scheduling in distributed data-intensive applications. In *HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, page 352, Washington, DC, USA, 2002. IEEE Computer Society.

[RRP+07] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, volume 0, pages 13–24, Washington, DC, USA, February 2007. IEEE Computer Society.

[SC08] S. W. Schlosser S. Chen. Map-reduce meets wider varieties of applications. Technical report, Intel Research Pittsburgh, 2008.

[SKK+90] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere. Coda: a highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[SNAKA+08] Elizeu Santos-Neto, Samer Al-Kiswany, Nazareno Andrade, Sathish Gopalakrishnan, and Matei Ripeanu. enabling cross-layer optimizations in storage systems with custom metadata. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, pages 213–216, New York, NY, USA, 2008. ACM.

[SNCBL04] Elizeu Santos-Neto, Walfredo Cirne, Francisco Brasileiro, and Aliandro Lima. Exploiting replication and data reuse to efficiently schedule data-intensive applications on grids. In *JSSPP*, pages 210–232, 2004.

[SPSB09] Alexandro S. Soares, Thiago Emmanuel Pereira, Jonhnny Weslley Silva, and Francisco Vilar Brasileiro. Um modelo de armazenamento de metadatados tolerante a falhas para o DDGfs (in portuguese) . In *WSCAD-SSC 2009: Proceedings of the 10th Computational Systems Symposium*, October 2009.

[SS96] Jim Smith and Santosh K. Shrivastava. A system for fault-tolerance execution of data and compute intensive programs over a network of workstations. In *Euro-Par '96: Proceedings of the Second International Euro-Par Conference on Parallel Processing*, pages 487–495, London, UK, 1996. Springer-Verlag.

[TW06] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems Design and Implementation*. Prentice Hall, 3rd edition, January 2006.

[Twe00] S. Tweedie. EXT3, Journaling File System. In *Ottawa Linux Symposium*, pages 24–29, 2000.

[VMF+05] Sudharshan S. Vazhkudai, Xiaosong Ma, Vincent W. Freeh, Jonathan W. Strickland, Nandan Tammineedi, and Stephen L. Scott. Freeloader: Scavenging desktop storage resources for scientific data. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 56, Washington, DC, USA, 2005. IEEE Computer Society.

[WBM+06] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In

OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation, pages 22–22, Berkeley, CA, USA, 2006. USENIX Association.