1 Introdução

Esse capítulo mostra o contexto da computação que torna relevante o estudo e a comparação de algoritmos de ordenação paralela no modelo MapReduce, sobretudo na implementação Hadoop; o crescimento de dados que contribui para a necessidade de mudança da computação sequencial para a computação paralela e a importância dos algoritmos de ordenação, utilizados em diversas aplicações. Também busca delimitar o tema a ser tratado, os objetivos do trabalho e a estrutura de capítulos.

1.1 Definição do Problema

Na última década, a quantidade de dados de trabalho utilizada pelos sistemas [elaborar mais] aumentou várias ordens de grandeza, fazendo do processamento dos dados um desafio para a computação sequencial. Como resultado, torna-se crucial substituir a computação tradicional por computação distribuída eficiente (1). A mudança no modelo de programação sequencial para paralelo é um fato inevitável e ocorre gradualmente, desde que a indústria declarou que seu futuro está em computação paralela (2).

O MapReduce é um modelo de programação paralela desenvolvido pela Google para processamento de grandes volumes de dados distribuídos em *clusters* (3). Esse modelo propõe simplificar a computação paralela, escondendo do desenvolvedor os detalhes da paralelização e utilizando duas funções principais - map e reduce. Uma das implementações mais conhecidas e utilizadas do modelo é o Hadoop (4), ferramenta de código aberto, desenvolvida por Doug Cutting em 2005 e apoiada pela Yahoo!.

A ordenação é um dos problemas fundamentais da ciência da computação e um dos problemas algorítmicos mais estudados. Suas aplicações vão desde sistemas de banco de dados à computação gráfica, além de muitos outros algoritmos que podem ser descritos em termos de ordenação (5, 6). Muitas aplicações dependem de ordenações eficientes como base para seu próprio desempenho e o uso crescente de computação paralela em sistemas computacionais gera a necessidade de algoritmos de ordenação inovadores, desenvolvidos para dar suporte a essas aplicações. Isso significa desenvolver rotinas eficientes de ordenação em arquiteturas paralelas e distribuídas.

Com a constante evolução das arquiteturas de computadores há uma necessidade contínua de explorar técnicas de ordenação em arquiteturas emergentes. Nesse sentido, o trabalho proposto por Pinhão (2011) apresentou uma avaliação da escalabilidade de algoritmos de ordenação paralela no modelo MapReduce. Para tal, foi desenvolvido o algoritmo de Ordenação por Amostragem, no ambiente Hadoop, e seu desempenho foi avaliado em relação à quantidade de dados de entrada e ao número de máquinas utilizadas.

Considerando esse contexto, o presente trabalho segue o tema e busca continuar o estudo com a implementação e análise de escalabilidade, em relação à quantidade de dados a ser ordenada e de número de máquinas utilizadas, do algoritmo Quicksort, no modelo MapReduce e ambiente Hadoop. Outro ponto é a comparação do desempenho dos dois algoritmos - Ordenação por Amostragem e Quicksort - em diferentes cenários.

1.2 Motivação

O volume de dados que é produzido e tratado diariamente em indústrias, empresas e até mesmo em âmbito pessoal teve um rápido crescimento nos últimos anos, tornando o desenvolvimento de soluções capazes de lidar com tais volumes de dados uma das grandes preocupações atuais. Estima-se que dados não estruturados são a maior porção e a de mais rápido crescimento dentro das empresas. Não é fácil medir o volume total de dados armazenados digitalmente, mas uma estimativa da *International Data Corporation* (IDC) calculou o tamanho do universo digital em 0,18 zettabytes em 2006, prevendo um crescimento de dez vezes até 2011, chegando a 1,8 zettabytes (7). Em 2008, o Facebook armazenava aproximadamente 10 bilhões de fotos, que ocupavam mais de um petabyte. *The Internet Archive* armazenava aproximadamente 2 petabytes de dados, com acréscimo de 20 terabytes por mês e a Bolsa de Valores de Nova Iorque gerava cerca de um terabyte de novos dados comerciais por dia. (4).

Mesmo para os computadores atuais, é um desafio conseguir lidar com quantidades de dados tão grandes. É preciso buscar soluções escaláveis, que apresentem bom desempenho mesmo com aumento significativo no número de recursos e na carga de trabalho. Nos últimos 40 anos, o aumento no poder computacional deveu-se, largamente, ao aumento na capacidade do hardware. Atualmente, o limite físico da velocidade do processador foi alcançado e arquitetos sabem que o aumento no desempenho só pode ser alcançado com o uso de computação paralela. Com isso, a indústria têm recorrido cada vez mais a arquiteturas paralelas para continuar a fazer progressos (8).

As tendências atuais estão redirecionando o foco da computação, do tradicional modelo de processamento científico para o processamento de grandes volumes de dados. Nesse sentido, arquiteturas paralelas, como as de memória

distribuída, estão cada vez mais frequentes, suprindo a necessidade de uma computação distribuída eficiente, que forneça alto desempenho no processamento de dados (9).

As técnicas tradicionais de programação paralela - como passagem de mensagens e memória compartilhada - em geral são complexas e de difícil entendimento para grande parte dos desenvolvedores. Em tais modelos é preciso gerenciar localidades temporais e espaciais; lidar explicitamente com concorrência, criando e sincronizando *threads* através de mensagens e semáforos. Dessa forma, não é uma uma tarefa simples escrever códigos paralelos corretos e escaláveis para algoritmos não triviais (10).

O MapReduce surgiu como uma alternativa aos modelos tradicionais, com o objetivo de simplificar a computação paralela. O foco do programador é a descrição funcional do algoritmo e não as formas de paralelização. Nos últimos anos o modelo têm se estabelecido como uma das plataformas de computação paralela mais utilizada no processamento de terabytes e petabytes de dados (11). MapReduce e sua implementação código aberto Hadoop oferecem uma alternativa economicamente atraente através de uma plataforma eficiente de computação distribuída, capaz de lidar com grandes volumes de dados e mineração de petabytes de informações não estruturadas (12).

Mesmo com o grande processamento empregado em interfaces gráficas, visualização e jogos, a ordenação continua a ser uma parte considerável da computação e estima-se que seja responsável por aproximadamente 80% dos ciclos de processamento(11). O uso de algoritmos paralelos de ordenação em tais aplicações melhora o tempo de execução do algoritmo e torna viável o processamento de grande quantidades de dados.

Na computação paralela, os algoritmos paralelos para ordenação têm sido objeto de estudo desde seu princípio, uma vez que a ordenação é um dos problemas fundamentais da ciência da computação. Um grande número de aplicações possui uma fase de computação intensa, na qual uma lista de elementos deve ser ordenada com base em algum de seus atributos. Um exemplo é o algoritmo de Page Rank (13) da Google: as páginas de resultado de uma consulta são classificadas de acordo com sua relevância, e depois precisam ser ordenadas de maneira eficiente (14).

Na ordenação paralela, fatores como movimentação de dados, balanço de carga, latência de comunicação e distribuição inicial das chaves são considerados ingredientes chave para o bom desempenho, e variam de acordo com o algoritmo escolhido como solução (14). No exemplo do Page Rank, o número de páginas a serem ordenadas é enorme, e elas são recolhidas de diversos servidores da Google; é uma questão fundamental escolher algoritmo paralelo com o melhor desempenho dentre as soluções possíveis. Dado o grande número de algoritmos de ordenação paralela e grande variedade de arquiteturas paralelas, é uma tarefa difícil escolher o melhor algoritmo para uma determinada máquina e instância do problema.

Além disso, não existe um modelo teórico conhecido que pode ser aplicado para prever com precisão o desempenho de um algoritmo em arquiteturas diferentes (6). Assim, estudos experimentais assumem uma crescente importância para a avaliação e seleção de algoritmos apropriados para ambientes paralelos. É preciso que estudos sejam realizados para que determinado algoritmo pode ser recomendado em certa arquitetura com alto grau de confiança.

1.3 Objetivos

Este projeto busca continuar o estudo sobre ordenação paralela desenvolvido por Pinhão (2011), com a análise de desempenho dos algoritmos de ordenação: Ordenação por Amostragem e Quicksort. No citado trabalho, foi feito um estudo sobre a computação paralela e algoritmos de ordenação no modelo MapReduce, através da implementação do algoritmo de Ordenação por Amostragem em ambiente Hadoop. No presente trabalho busca-se comparar os algoritmos Ordenação por Amostragem e Quicksort em relação à quantidade de dados a serem ordenados, variabilidade dos dados de entrada e número máquinas utilizadas.

Desse modo, os objetivos deste trabalho são:

- Estudar a programação paralela aplicada a algoritmos de ordenação;
- Implementar o algoritmo de ordenação Quicksort no modelo MapReduce, com o *framework* Hadoop;
- Comparar as implementações dos algoritmos de ordenação paralela: Ordenação por Amostragem e Quicksort.

1.4 Organização do Texto

Esse projeto está organizado em seis capítulos. O próximo capítulo apresenta o referencial teórico para o desenvolvimento do trabalho, com conceitos de computação paralela e do modelo MapReduce. O Capítulo 3 complementa o referencial teórico e apresenta os conceitos da ordenação de dados e os algoritmos de ordenação paralela. O Capítulo 4 descreve a metodologia de pesquisa, indicando os passos seguidos durante o desenvolvimento. Os resultados preliminares obtidos até a entrega do projeto são apresentados no Capítulo 5. As conclusões e os próximos passos para a finalização do projeto estão no Capítulo 6.

2 Computação Paralela

Esse capítulo aborda os principais conceitos envolvidos no trabalho, como computação paralela, o modelo MapReduce e sua implementação de código aberto Hadoop.

2.1 Definições

A computação paralela constitui-se de uma coleção de elementos de processamento que se comunicam e cooperam entre si e com isso resolvem um problema de maneira mais rápida (15). Mesmo com o avanço tecnológico das últimas décadas, as arquiteturas de Von Neumann demostram deficiências quando utilizadas por aplicações que necessitam de grande poder computacional. Essa deficiência impulsionou a utilização da computação paralela, com o objetivo de aumentar a capacidade de processamento das máquinas.

No estudo de computação paralela, é importante diferenciar os conceitos de paralelismo e concorrência, pois ambos tratam de programação e execução de tarefas em múltiplos fluxos, implementados com o objetivo de resolver um único problema. Concorrência consiste em diferentes tarefas serem executadas ao mesmo tempo, de forma a produzir um resultado particular mais rapidamente. Isso não implica necessariamente na existência de vários elementos de processamento; a concorrência pode ocorrer tanto com um único processador quanto com múltiplos processadores. Por outro lado, o paralelismo exige a execução de diversas tarefas simultaneamente, com a necessidade de vários elementos de processamento. Se há apenas um elemento de processamento não há paralelismo, pois apenas uma tarefa será executada a cada instante, mas pode haver concorrência, pois o processador pode ser compartilhado pelas tarefas em execução (10).

Comparada à computação sequencial, a computação paralela apresenta alto desempenho e soluções mais naturais para problemas intrinsecamente paralelos, contudo sua utilização também inclui desvantagens. O desenvolvimento de soluções paralelas apresenta maior dificuldade na programação, pois há mais detalhes e diversidades na implementação, uma vez que um programa paralelo envolve múltiplos fluxos de execução simultâneos e é preciso coordenar todos os fluxos para completar uma dada computação. Além disso há a necessidade de sincronismo e de balanceamento de cargas (16). Assim, o desenvolvimento de software paralelo introduz três principais desafios: assegurar confiabilidade de software, minimizar o tempo de desenvolvimento e conquistar bom desempenho na aplicação (17).

Manter a confiabilidade do sistema é essencial, pois ao se introduzir paralelismo à aplicação, ela se torna vulnerável às condições de corrida e dependendo da ordem de execução das tarefas pode se comportar de forma diferente. Mesmo se nenhuma alteração for feita no hardware ou nos arquivos de entrada, execuções consecutivas da mesma aplicação podem produzir resultados diferentes. Lidar com situações como essa é particularmente desafiante, pois tais erros são assíncronos e ocorrem eventualmente, o que torna difícil evitá-los e encontrá-los durante testes.

Outro desafio é minimizar o tempo de desenvolvimento, já que muitas vezes o desenvolvimento paralelo é mais complexo que o sequencial e demanda, do desenvolvedor, conhecimento prévio de paralelismo. Em geral, é preciso maior tempo para escrever o código e a depuração é mais trabalhosa, diversos testes devem ser feitos no sistema, o que pode dispender maior tempo.

O bom desempenho da aplicação é um dos objetivos centrais da paralelização, mas pode ser comprometido por comunicação excessiva ou balanceamento irregular de carga. O balanceamento de carga busca atingir um aproveitamento ótimo dos recursos do sistema, alocando tarefas de forma a obter o mesmo nível de esforço em todos os processadores. A comunicação e sincronização de tarefas devem ser minimizadas pelo desenvolvedor, pois são tipicamente as maiores barreiras para se atingir grande desempenho em programas paralelos. Após o desenvolvimento de uma aplicação paralela é importante avaliar se os tempos de execução paralelos são menores que os sequenciais e quão menores são esses tempos, através de indicadores de desempenho. Para avaliar o desempenho de algoritmos paralelos as principais métricas são o *speedup*, a eficiência e a lei de Amdahl.

O *speedup* (S_p) é uma métrica que informa quão mais rápida é a aplicação paralela, em comparação com sua versão sequencial. Para isso, determina a relação existente entre o tempo gasto para executar um algoritmo em um único processador ($T_{sequencial}$) e o tempo gasto para executá-lo em p processadores ($T_{paralelo}$):

$$S_p = \frac{T_{sequencial}}{T_{paralelo}}$$

Em uma situação ideal o *speedup* é igual a p, o que indicaria que o aumento da capacidade de processamento é diretamente proporcional ao número de processadores. Contudo, o *speedup* é diretamente afetado por fatores como comunicação entre processos, granulosidades inadequadas e partes não paralelizáveis de programas.

A eficiência (E_p) é outro parâmetro utilizado para medir o desempenho na computação paralela. Ela relaciona

o *speedup* ao número de processadores, identificando a taxa de utilização do processador. Pode ser calculada por:

$$E_p = \frac{S_p}{p}$$

Enquanto o *speedup* relaciona tempos de execução, a eficiência avalia o quão bem estão sendo utilizados os recursos do sistema. Em uma paralelização ideal a eficiência tem valor 1, significando que os processadores têm utilização total.

A Lei de Amdahl é outra importante métrica, que fornece um limite superior para o valor do *speedup* que pode ser atingido em um sistema. Para aplicar a lei de Amdahl é preciso estimar o percentual de tempo que a aplicação vai executar em paralelo e o percentual que executará sequencialmente. Ela demonstra que o ganho de desempenho obtido com a paralelização do sistema é limitado pela fração de tempo em que o programa executa código sequencial e é um bom indicativo do potencial para *speedup* de uma aplicação.

2.1.1 Modelos de programação paralela

Um modelo de programação paralela descreve um sistema de computação paralela em termos da semântica da linguagem ou do ambiente de programação. Seu objetivo é fornecer um mecanismo com o qual o programador pode especificar programas paralelos. Os tradicionais modelos de programação paralela são: memória compartilhada, memória distribuída e paralelismo de dados e de tarefas (*threads* e *multithreads*).

No ambiente de memória compartilhada, múltiplos processadores compartilham o espaço de endereçamento de uma única memória. A comunicação entre os processos é implícita, pois a memória é acessível diretamente por todos os processadores.

O paralelismo de dados é o modelo de programação no qual as várias tarefas realizam operações em elementos distintos de dados, simultaneamente, e então trocam dados globalmente. No ambiente *multithread*, múltiplas *threads* podem ser executadas dentro de um único processo. Cada *thread* possui seu próprio conjunto de registradores e pilha, porém compartilha o mesmo espaço de endereçamento, temporizadores e arquivos, de forma natural e eficiente com as demais *threads* do processo.

O modelo de memória distribuída é composto por várias unidades de processamento com memória fisicamente distribuída, chamadas nós, e por uma rede de interconexão que os conecta e transfere dados entre eles. Cada nó é uma unidade independente, com processador e memória próprios, como representado na Figura 1. Nesse modelo, as tarefas compartilham dados por meio de comunicação com o envio e recebimento de mensagens, que pode ser realizada por meio de bibliotecas como a MPI (*Message Passing Interface*) e a PVM (*Parallel Virtual Machine*). Quando o modelo de memória distribuída consiste em conjuntos de computadores completos com rede de intercomunicação dedicada é denominado *cluster*. Geralmente *clusters* são baseados em computadores e topologias de rede padrão, programados como uma única unidade (16).

[trim=0cm 1cm 0cm 0cm, width=0.5]figuras/Arquitetura.pdf

Figura 1: Modelo de arquitetura distribuída

2.1.2 Computação paralela em *clusters*

Com o avanço tecnólogico da última década, o volume crescente de dados gerados, coletados e armazenados tornou o processamento dos dados inviável a um único computador. A quantidade de dados atualmente processados cria a necessidade de computação de alto desempenho, cujo foco sejam os dados. Como resultado, torna-se crucial substituir a computação tradicional por computação distribuída eficiente, e é um caminho natural para o processamento de dados em larga escala o uso de *clusters* (1).

Clusters são conjuntos de máquinas, ligadas em rede, que comunicam-se através do sistema, trabalhando como se fossem uma única máquina de grande porte. Dentre algumas características observadas em um *cluster*, é possível destacar: o baixo custo se comparado a supercomputadores; a proximidade geografica dos nós; altas taxas de transferência nas conexões entre as máquinas e o uso de máquinas em geral homogêneas (18).

Apesar dos computadores em um *cluster* não precisarem processar necessariamente a mesma aplicação, a grande vantagem de tal organização é a habilidade de cada nó processar individualmente uma fração da aplicação, resultando em desempenho que pode ser comparado ao de um supercomputador. Em geral os computadores de *clusters* são de baixo custo, o que permite que um grande número de máquinas seja interligado, garantindo desempenho e melhor custo-benefício que os supercomputadores, o que apresenta outra vantagem. Além disso, novas máquinas podem ser facilmente incorporadas ao *cluster*, tornando-o uma solução mais flexível, principalmente por ser formado por máquinas de capacidade de processamento similar.

O bom desempenho das aplicações em *clusters* envolve conceitos relacionados à infraestrutura, principalmente comunicação entre os nós e balanceamento de carga. Para que o processamento do *cluster* possa ser utilizado de maneira eficiente, é importante que os dados a serem processados sejam transferidos suficientemente rápidos, através de redes de alta velocidade, para evitar que os processadores fiquem ociosos (16).

2.1.3 Computação paralela em grandes volumes de dados

O processamento em *clusters* é uma tarefa cujo desempenho é dependente de diversos fatores, como descrito anteriormente. O processamento de grandes volumes de dados também é uma tarefa desafiadora, que tem sido objeto de vários estudos. Os sistemas utilizados para processar grandes volumes de dados devem se basear em alguns princípios para garantir a escalabilidade e o bom desempenho.

A coleta e manutenção dos dados deve ser funções do sistema e não tarefa dos usuários. O sistema deve prover tratamento intrínseco dos dados e os usuários devem ter facilidade para acessá-los. Mecanismos de confiabilidade, como replicação e correção de erros devem ser incorporados como parte do sistema de modo a garantir integridade e disponibilidade dos dados. O uso de modelos de programação paralela de alto nível também deve ser incentivado. O desenvolvedor deve utilizar programação de alto nível que não inclua configurações específicas de uma máquina. O trabalho de distribuir a computação entre as máquinas de forma eficiente deve ficar a cargo do sistema, e não do desenvolvedor.

Além disso, um sistema para computação de grandes volumes de dados deve implementar mecanismos de confiabilidade, no qual os dados originais e intermediários são armazenados de forma redundante. Isso permite que no caso de falhas de componente ou dados seja possível refazer a computação. Além disso, a máquina deve identificar e desativar automaticamente componentes que falharam, de modo a não prejudicar o desempenho do sistema e se manter sempre disponível (9).

Grandes empresas de serviços de Internet - como Google, Yahoo!, Facebook e Amazon - buscam soluções para processamento de dados em grandes conjuntos de máquinas que atendam as características descritas, pois com um software que provê tais características é possível alcançar alto grau de escalabilidade e custo-benefício.

Dentre as principais propostas está o modelo MapReduce e sua implementação Hadoop, que são soluções escaláveis, capazes de processar grandes volumes de dados, com alto nível de abstração para distribuir a aplicação e mecanismos de tolerância a falhas. A próxima seção apresenta com mais detalhes o modelo e suas características.

2.2 MapReduce

O MapReduce é um modelo de programação paralela criado pela Google para processamento de grandes volumes de dados em *clusters*. Esse modelo propõe simplificar a computação paralela e ser de fácil uso, abstraindo conceitos complexos da paralelização - como tolerância a falhas, distribuição de dados e balanceamento de carga - e utilizando duas funções principais: Map e Reduce. As do desenvolvimento paralelo não é vista pelo desenvolvedor, que pode se ocupar em desenvolver a solução proposta (3).

Esse modelo de programação é inspirado em linguagens funcionais, tendo como base as primitivas Map e Reduce. Os dados de entrada são específicos para cada aplicação e descritos pelo usuário. A função Map é aplicada aos dados de entrada e produz uma lista intermediária de pares (chave, valor). Todos os valores intermediários associados a uma mesma chave são agrupados e enviados à função Reduce. A função Reduce então combina esses valores para formar um conjunto menor de resultados. Tipicamente há apenas zero ou um valores de saída em cada função Reduce. A saída de cada função é agrupada e forma um conjunto de pares no formato (chave, valor).

O pseudocódigo a seguir apresenta um exemplo de uso do MapReduce, cujo objetivo é contar a quantidade de ocorrências de cada palavra em um documento. A função Map recebe como valor uma linha do documento, e como chave o número da linha. Para cada palavra encontrada na linha recebida, a função emite a palavra e a contagem de uma ocorrência. A função Reduce, recebe como chave uma palavra, e uma lista dos valores emitidos pela função Map, associados com a palavra questão. As ocorrências da palavra são agrupadas e a função retorna palavra e seu total de ocorrências.

Listing 2.1: some-code

```
1 Function Map (Integer chave, String valor):
2   #chave: número da linha no arquivo.
3   #valor: texto da linha correspondente.
4   listaDePalavras = split (valor)
5   for palavra in listaDePalavras:
6     emit (palavra, 1)
7 Function Reduce (String chave, Iterator valores):
8   #chave: palavra emitida pela função Map.
9   #valores: conjunto de valores emitidos para a chave.
```

```
10    total = 0
11    for v in valores:
12      total = total + 1
13    emit (palavra, total)
```

A Figura 2 ilustra o fluxo de execução para este exemplo. A entrada é um arquivo contendo as linhas "hadoop conta", "conta palavras" e "exemplo hadoop".

[trim=0cm 9cm 0cm 1cm, width=]figuras/MapReduceExemplo.pdf

Figura 2: Fluxo simplificado da contagem de palavras com o MapReduce

2.2.1 Arquitetura do MapReduce

O MapReduce é constituído de uma arquitetura com dois tipos principais de nós: *Master* e *Worker*. O nó mestre tem como função atender requisições de execução dos usuários, gerenciá-las, criar tarefas e distribuí-las entre os nós trabalhadores, que executam as tarefas com base nas funções Map e Reduce definidas pelo usuário. A arquitetura também inclui um sistema de arquivos distribuídos, onde ficam armazenados os dados de entrada e intermediários.

2.2.2 Visão geral do fluxo de execução

As chamadas da função Map são distribuídas automaticamente entre as diversas máquinas através do particionamento dos dados de entrada em M conjuntos. Cada conjunto pode ser processado em paralelo por diferentes máquinas. As chamadas da função Reduce são distribuídas pelo particionamento do conjunto intermediário de pares em R partes. O número de partições R pode ser definido pelo usuário.

A Figura 3 ilustra o fluxo de uma execução do modelo MapReduce (3). A sequência de ações descrita a seguir explica o que ocorre em cada um dos passos. A numeração dos itens a seguir corresponde à numeração da figura.

[trim=0cm 2cm 0cm 1cm, width=]figuras/MapReduceOverflow.pdf

Figura 3: Visão geral do funcionamento do modelo MapReduce.

1. A biblioteca MapReduce, no programa do usuário, divide os arquivos de entrada em M pedaços. Em seguida, iniciam-se muitas cópias do programa para o conjunto de máquinas;
2. Uma das cópias do programa é especial: o mestre (*Master*). Os demais são trabalhadores (*Workers*) cujo trabalho é atribuído pelo mestre. Existem M tarefas Map e R tarefas Reduce a serem atribuídas. O mestre atribui aos trabalhadores ociosos uma tarefa Map ou uma tarefa Reduce;
3. Um trabalhador que recebe uma tarefa Map lê o conteúdo do fragmento de entrada correspondente. Ele cria pares (chave, valor) a partir dos dados de entrada e encaminha cada par para a função Map definida pelo usuário. Os pares (chave, valor) intermediários, produzidos pela função Map, são colocados no *buffer* de memória;
4. Periodicamente, os pares colocados no *buffer* são gravados no disco local, divididos em R regiões pela função de particionamento. As localizações desses pares bufferizados, no disco local, são passadas de volta para o mestre que é responsável pelo encaminhamento desses locais aos trabalhadores Reduce;
5. Quando um trabalhador Reduce é notificado pelo mestre sobre essas localizações, ele usa chamadas de procedimento remoto para ler os dados dos discos locais dos trabalhadores Map. Quando um trabalhador Reduce tiver lido todos os dados intermediários da sua partição, ele a ordena pelas chaves intermediárias, de forma que todas as ocorrências da mesma chave estejam agrupadas. Se a quantidade de dados intermediários é muito grande para caber na memória, um tipo de ordenação externa é usado;
6. O trabalhador Reduce itera sobre os dados intermediários ordenados e, para cada chave encontrada, repassa a chave e o conjunto correspondente de valores intermediários para função Reduce do usuário. A saída da função Reduce é anexada a um arquivo de saída final para essa partição Reduce;

Após todas as tarefas Map e Reduce concluídas, o mestre acorda o programa do usuário, retornando, neste ponto, a chamada MapReduce para o código do usuário.

2.3 Hadoop

O Hadoop é uma das implementações mais conhecidas modelo MapReduce. Ele provê o gerenciamento de computação distribuída, de maneira escalável e confiável. É uma implementação código aberto em Java desenvolvida por Doug Cutting em 2005 e mantida pela Apache Software Foundation (4).

Um dos principais benefícios do Hadoop é permitir o processamento em conjunto de centenas de máquinas de maneira transparente, o que significa que o usuário não deve se preocupar com mecanismos de tolerância a falhas, que é provido pelo sistema. Facebook, Yahoo! e eBay utilizam o ambiente Hadoop em seus *clusters*, para processar diariamente terabytes de dados e logs de eventos para detecção de *spam*, *business intelligence* e diferentes tipos de otimização (12).

O mecanismo de tolerância a falhas implementado pelo sistema, permite que o trabalho do usuário possa ser concluído mesmo que ocorra alguma falha de disco, de processo ou de nó. Periodicamente, o nó mestre envia mensagens aos demais nós para verificar seus estados. Se nenhuma resposta é recebida, o mestre identifica que houve falha neste nó e o substitui. As tarefas que não foram executadas são reescalonadas para os demais nós. O mecanismo de replicação garante que sempre haja um número determinado de cópias dos dados, e caso um dos nós de armazenamento seja perdido, os demais se encarregam de realizar uma nova replicação (4).

2.3.1 Sistema de Arquivos do Hadoop

O *Hadoop Distributed File System* (HDFS) é um sistema de arquivos distribuído desenvolvido para armazenar grandes conjuntos de dados e ser altamente tolerante a falhas (4). A plataforma Hadoop fornece o HDSF como sistema de arquivos padrão, mas é compatível com diversos sistemas de arquivos distintos, como Amazon S3 (Native e Block-based), CloudStore, HAR, Local (destinado a unidades de armazenamento conectadas localmente) e sistemas mantidos por servidores FTP e HTTP.

A arquitetura do HDFS também é do tipo mestre-escravos. O nó mestre consiste em um *JobTracker* para as tarefas MapReduce e um *NameNode* responsável por manter e controlar todos os metadados do sistema de arquivos e gerenciar a localização dos dados. O nó mestre também é responsável por outras atividades, como por exemplo, balanceamento de carga, *garbage collection* e atendimento a requisições dos clientes. Os nós escravos são formados por um *TaskTracker* para as tarefas MapReduce e por um *DataNode* responsável por armazenar e transmitir os dados aos usuários que os requisitarem.

A Figura 4 ilustra a arquitetura do sistema de arquivos distribuídos. O *NameNode* gerencia e manipula todas as informações dos arquivos, tal como a localização e o acesso. Os *DataNodes* se encarregam da leitura e escrita das informações nos sistemas de arquivo cliente. Os *JobTracker* e *TaskTracker* são responsáveis por executar as tarefas MapReduce.

[trim=2cm 12cm 2cm 2cm, width=0.6]figuras/HadoopCluster.pdf

Figura 4: Visão abstrata do cluster.

O HDFS incorpora funcionalidades que têm grande impacto no desempenho geral do sistema. Uma delas é conhecida como *rack awareness*. Com esse recurso, o sistema de arquivos é capaz de identificar os nós escravos que pertencem a um mesmo *rack*, e distribuir as réplicas de maneira mais inteligente, aumentando o desempenho e a confiabilidade do sistema. Outra funcionalidade é a distribuição das tarefas considerando localização dos dados nos nós. O sistema de arquivos procura manter um balanceamento na ocupação das unidades de armazenamento, e o *framework* busca atribuir tarefas a um escravo que possua, em sua unidade de armazenamento local, os dados que devem ser processados. Assim, quando executa-se grandes operações MapReduce com um número significativo de nós, a maioria dos dados são lidos localmente e o consumo de banda é mínimo.

3 Ordenação de dados

A ordenação é o processo de organizar elementos de uma sequência em determinada ordem, e é um dos problemas fundamentais da computação devido à sua importância teórica e prática. A ordenação é utilizada por um grande número de aplicações computacionais, como compiladores e sistemas operacionais, que usam extensivamente a ordenação para lidar com tabelas e listas [Rajasekaran e Reif 1989]. A ordenação participa de aplicações de computação gráfica, compressão de dados, para determinar a duplicidade de elementos, encontrar o maior valor, realizar busca contínua e encontrar the convex hull. É também realizada internamente por operações SQL, em criação de índices e buscas binárias, portanto todos os sistemas que que usam banco de dados se beneficiam de uma rotina eficiente de ordenação. [Martin, 1971] (W. A. Martin. Sorting. ACM Comp Surv., 3(4):147–174, 1971.)

De forma geral, a ordenação pode ser dividida em dois grupos: a ordenação interna e externa. A ordenação em memória interna é caracterizada pelo armazenamento de todos os registros na memória principal, onde seus acessos são feitos diretamente pelo processador. Essa ordenação é possível apenas quando a quantidade de dados é pequena o suficiente para ser armazenada em memória.

Quando é preciso ordenar uma base de dados muito grande, que não cabe na memória principal, um outro modelo faz-se necessário, a ordenação externa. Apesar do problema nos dois casos ser o mesmo - rearranjar os registros de um arquivo em ordem ascendente ou descendente - não é possível usar as mesmas estratégias da ordenação interna, pois o acesso aos dados precisa ser feito em memória secundária, como discos, cujo tempo de acesso é superior ao da memória principal.

Na ordenação externa, os itens que não estão na memória principal devem ser buscados em memória secundária e trazidos para a memória principal, para assim serem comparados. Esse processo se repete inúmeras vezes, o que o torna lento, uma vez que os processadores ficam grande parte do tempo ociosos à espera da chegada dos dados à memória principal para serem processados. Por esse motivo, a grande ênfase de um método de ordenação externa deve ser na minimização do número de vezes que cada item é transferido entre a memória interna e a memória externa. Além disso, cada transferência deve ser realizada de forma tão eficiente quanto as características dos equipamentos disponíveis permitam (19).

3.1 Ordenação Paralela

Diversas aplicações possuem uma fase de processamento intenso, na qual é preciso ordenar uma lista de elementos. Mesmo algoritmos de ordenação sequenciais ótimos, como o QuickSort e o HeapSort, apresentam custo mínimo O(n x log n) para ordenar uma sequência de n chaves [Aho e Hopcroft 1974]. Isso significa que com o crescimento do número de elementos a ser ordenado o tempo para realizar a ordenação aumenta de maneira não linear, o que pode ser um entrave ao processamento. A fim de resolver tal problema, com o surgimento do processamento paralelo, foram apresentadas versões paralelas dos algoritmos de ordenação sequenciais, com o intuito de diminuir consideravelmente o tempo de execução [Rajasekaran e Reif 1989].

A ordenação paralela é o processo de ordenação feito em múltiplas unidades de processamento, que trabalham em conjunto para ordenar uma sequência de entrada. O conjunto inicial é dividido em subconjuntos disjuntos, que são associados a uma única unidade de processamento. A sequência final ordenada é obtida a partir da composição dos subconjuntos ordenados. É um ponto fundamental do algoritmo de ordenação paralela que a distribuição dos dados a serem ordenados por cada processo individual seja feita de tal forma que todas as unidades de processamento estejam trabalhando e que o custo de redistribuição de chaves entre os processadores seja minimizado.

Diversas soluções de ordenação podem ser consideradas ao implementar um algoritmo de ordenação em ambiente paralelo. Cada uma delas atende um cenário, tipo de entrada, plataforma ou arquitetura particulares. Dessa forma, ao implementar algoritmos de ordenação paralela, é importante considerar certas condições que interferem no desempenho final do algoritmo, relacionadas tanto ao ambiente de implementação, quanto ao conjunto de dados que deve ser ordenado. As principais questões a serem analisadas são (14):

- **Habilidade de explorar distribuições iniciais parcialmente ordenadas:** Alguns algoritmos podem se beneficiar de cenários nos quais a sequência de entrada dos dados é mesma, ou pouco alterada. Nesse caso, é possível obter melhor desempenho ao realizar menos trabalho e movimentação de dados. Se a alteração na posição dos elementos na sequência é pequena o suficiente, grande parte dos processadores mantém seus dados iniciais e precisa se comunicar apenas com os processadores vizinhos.
- **Movimentação dos dados:** A movimentação de dados entre processadores deve ser mínima durante a execução do algoritmo. Em um sistema de memória distribuída, a quantidade de dados a ser movimentada é um ponto crítico, pois o custo de troca de dados pode dominar o custo de execução total e limitar a escalabilidade.

- **Balanceamento de carga:** O algoritmo de ordenação paralela deve assegurar o balanceamento de carga ao distribuir os dados entre os processadores. Cada processador deve receber uma parcela equilibrada dos dados para ordenar, uma vez que o tempo de execução da aplicação é tipicamente limitada pela execução do processador mais sobrecarregado.
- **Latência de comunicação:** A latência de comunicação é definida como o tempo médio necessário para enviar uma mensagem de um processador a outro. Em grandes sistemas distribuídos, reduzir o tempo de latência se torna muito importante.
- **Sobreposição de comunicação e computação:** Em qualquer aplicação paralela, existem tarefas com focos em computação e comunicação. A sobreposição de tais tarefas permite que sejam feitas tarefas de processamento e ao mesmo tempo operações de entrada e saída de dados, evitando que os recursos fiquem ociosos durante o intervalo de tempo necessário para a transmissão da carga de trabalho.

Além das condições relacionadas à implementação do algoritmo em ambiente paralelo, existem outras condições necessárias, relacionadas principalmente ao conjunto de elementos a ser ordenado. Considerando um conjunto de n chaves e p processadores, durante a execução de qualquer algoritmo de ordenação paralela é preciso que o conjunto de chaves seja particionado em p subconjuntos mutualmente exclusivos, sem nenhuma chave duplicada. É necessário ainda, que todas as chaves da sequência inicial sejam mantidas, ou seja, que não se perca nenhuma chave durante a distribuição entre os processadores.

Após o conjunto estar ordenado, é preciso verificar se todas as chaves da sequência inicial foram preservadas, se todas as chaves de cada processador estão ordenadas em ordem crescente, se a maior chave no processador p_i é inferior ou igual à menor chave no processador p_{i+1} e se a saída resultante é uma sequência de chaves totalmente ordenada.

3.1.1 Fluxo geral de execução da ordenação paralela

Na execução de um algoritmo de ordenação paralela podem ser identificadas algumas tarefas principais, que todos os algoritmos precisam executar em algum momento, normalmente realizadas de forma sequencial (14). A primeira tarefa é a ordenação local, na qual as chaves em cada processador são ordenadas inicialmente ou ordenadas em grupos. Existe também uma fase de agrupamento, pois muitas vezes é necessário colocar as chaves em grupos, a fim de enviá-las a outros processadores ou calcular histogramas. Por fim, é preciso realizar a intercalação das chaves ordenadas em subsequências em uma sequência completa.

De forma geral, todos os algoritmos de ordenação paralela executam tarefas similares que podem ser definidas, superficialmente, como se segue:
1. Realizar processamento local;
2. Coletar informações relevantes de distribuição de todos os processadores;
3. Em um único processador, inferir uma divisão de chaves a partir das informações coletadas;
4. Transmitir aos outros processadores a divisão dos elementos;
5. Realizar processamento local;
6. Mover os dados de acordo com os elementos de divisão;
7. Realizar processamento local;
8. Se a divisão de chaves foi incompleta, retornar ao passo 1;

De acordo com essa generalização é possível identificar pontos que se relacionam diretamente com as condições que limitam o desempenho dos algoritmos de ordenação paralela, e fornecem ideias para a análise de eficiência da comunicação dos algoritmos. Primeiro, há duas tarefas principais de comunicação: descobrir um vetor de divisão global e enviar os dados para os processadores adequados. Em segundo lugar, a maioria dos algoritmos têm múltiplos estágios de computação local e pode ser muito vantajoso sobrepor este processamento local e a comunicação. A fração de processamento local que pode ser sobreposta à comunicação necessária em um algoritmo é um bom indicativo para comparação da escalabilidade dos algoritmos de ordenação paralela.

3.2 Algoritmos de Ordenação Paralela

Esse seção apresentará os algoritmos de ordenação paralela objetos desse trabalho: algoritmo Ordenação por Amostragem ou *SampleSort*, *QuickSort* e as aplicações de ordenação em Hadoop *Terasort* e *Sort*.

3.2.1 Ordenação por Amostragem

O algoritmo *SampleSort* ou Ordenação por Amostragem é um método de ordenação baseado na divisão do arquivo de entrada em subconjuntos, de forma que as chaves de um subconjunto i sejam menores que as chaves do

subconjunto $i + 1$. Após a divisão, cada subconjunto é enviado a um processador, que ordena os dados localmente. Ao final, todos os subconjuntos são concatenados e formam um arquivo globalmente ordenado.

Nesse algoritmo, o ponto chave é dividir as partições de maneira balanceada, para que cada processador receba aproximadamente a mesma carga de dados. Para isso, é preciso determinar o número de elementos que devem ser destinados a uma certa partição, o que é feito através da amostragem das chaves do arquivo original. Essa estratégia baseia-se na análise de um subconjunto de dados denominado amostra, ao invés de todo o conjunto, para estimar a distribuição de chaves e construir partições balanceadas.

Existem três tipos de estratégias de amostragem: *SplitSampler*, *IntervalSampler* e *RandomSampler*. O *SplitSampler* seleciona os n primeiros registros do arquivo para formar a amostra. O *IntervalSampler* cria a amostra com a seleção de chaves em intervalos regulares no arquivo. No *RandomSampler*, a amostra é constituída por chaves selecionadas aleatoriamente no conjunto. A melhor estratégia de amostragem depende diretamente dos dados de entrada. O *SplitSampler* não é recomendado para arquivos quase ordenados, pois as chaves selecionadas serão as iniciais, que não são representativas do conjunto como um todo. Nesse caso, a melhor escolha é o *IntervalSampler* pelo fato de selecionar chaves que representam melhor a distribuição do conjunto. O *RandomSampler* é considerado um bom amostrador de propósito geral [White 2009], e foi o amostrador escolhido na implementação do algoritmo Ordenação por Amostragem feito por Pinhão (2011), que foi utilizado neste trabalho.

Para criar a amostra, o *RandomSampler* necessita de alguns parâmetros, como a probabilidade de escolha de uma chave, o número máximo de amostras a serem selecionadas para realizar a amostragem e o número máximo de partições que podem ser utilizadas. O número máximo de partições é determinado pelo número de núcleos disponíveis por processador e pela quantidade de máquinas, de acordo com a equação: $particoes = nucleos$ x $maquinas$. Após a definição das amostras, são conhecidos os intervalos compreendidos por cada partição. As informações das partições são armazenadas em um arquivo e transmitidas para as demais máquinas por meio de cache distribuído.

Algoritmo em MapReduce

O algoritmo Ordenação por Amostragem, quando implementado no modelo MapReduce no ambiente Hadoop pode ser dividido nas fases Map e Reduce. Na fase Map os arquivos de entrada são lidos e são formados os pares (chave, valor) para cada registro presente no arquivo. Em seguida é definido o vetor contendo as amostras. A partir desse vetor de amostras os dados são divididos em partições. O número de partições é determinado pelo número de máquinas e núcleos de processamento. Por meio de cache distribuído, as informações das partições são transmitidas para as máquinas participantes e os dados particionados. Cada partição é atribuída a um processador, que executa a tarefa Reduce.

Na fase Reduce, cada processador ordena os dados localmente. Essa ordenação é realizada pelo próprio *framework*, que avalia a profundidade da árvore de recursão e escolhe entre os algoritmos QuickSort e HeapSort. Após a ordenação local os dados são enviados para a máquina mestre, na qual são concatenados e formam o conjunto final ordenado.

O balanceamento das partições, ou seja, a formação de partições com tamanhos aproximados é fundamental para o algoritmo de Ordenação por Amostragem, pois reduz a possibilidade de que um processador esteja ocioso, enquanto outro processador está sobrecarregado, situação que comprometeria o desempenho do algoritmo (4).

A Figura **??** apresenta um exemplo de como seria a execução do algoritmo implementado no Hadoop. Nesse exemplo, está representada a execução do algoritmo em duas máquinas com dois núcleos cada, totalizando 4 unidades de processamento. Primeiramente foram lidos os arquivos e formados os pares (chave, valor) (passos 1 e 2). Em seguida foram amostrados 3 valores com o *RandomSampler* (passo 3) para determinar os valores presentes nas 4 partições (passo 4). Após formadas as partições, os dados foram distribuídos para os escravos executarem a função Reduce. A função Reduce ordena localmente os dados (passo 5) e o mestre agrupa todos os valores, escrevendo o arquivo final (passo 6).

3.2.2 Quicksort

O QuickSort foi criado por Hoare em 1960 e utiliza uma estratégia de dividir para conquistar. Na implementação sequencial, a estratégia é o particionamento recursivo da sequência de entrada utilizando um elemento como pivô. Após a escolha do elemento pivô, a lista é divida em duas sublistas, uma contendo elementos maiores que o pivô, e outra contendo elementos menores e igual. Em cada sublista é escolhido um novo pivô, o processo se repete, até que cada lista contenha apenas um elemento. Ao final obtém-se uma lista com elementos ordenados.

Em geral, a complexidade do algoritmo QuickSort é O(n x log n)Mas caso a entrada de dados seja um conjunto ordenado ou quase ordenado o desempenho do QuickSort será comprometido, se o pivô for escolhido nas extremidades, pois uma das das partições diminuirá o tamanho da outra. Nesse situação a complexidade do algoritmo pode chegar a O(n2). O melhor caso será quando os conjuntos de dados tiverem tamanhos próximos após o particionamento.

A estratégia de dividir para conquistar do QuickSort é naturalmente extensível à paralelização. Existem várias

implementações paralelas do QuickSort como: Quinn's QuickSort, Hyper QuickSort, Sanders QuickSort e Grama QuickSort. Apesar das variações, todas as implementações tem como entrada um conjunto de processadores, um mecanismo de escolha do pivô e uma lista de chaves para nelas operarem. A saída dos algoritmos é uma sequencia de chaves globalmente ordenadas.

A paralelização do QuickSort é feita pelo uso de pivôs para realizar o particionamento recursivo do conjunto de processadores que interagem. A questão do balanceamento de carga, fundamental ao desempenho do algoritmo é mantida pela semântica da seleção do pivô [Kale e Solomonik 2010].

Na versão paralela do QuickSort , inicialmente tem-se um conjunto de p processadores e um pivô. Após a escolha desse pivô, o mestre envia, valor do pivô para os demais processadores do conjunto. Cada processador divide suas chaves em dois grupos: elementos maiores que o pivô e elementos menores que o pivô. São contabilizados a quantidade de elementos maiores e menores que o pivô em todos os processadores. Com essa informação o mestre é capaz de definir o número de processadores que deve receber chaves menores que o pivô e o número de processadores que deve receber chaves maiores que o pivô, a fim de manter o balanceamento.

Além disso é conhecido o número médio de chaves que cada processador deve receber.

No QuickSort paralelo todas as chaves serão movidas entre os processadores durante a execução do algoritmo. No entanto a latência de comunicação das mensagens no Quicksort paralelo aumentam apenas com o crescimento do número de processadores. Com isso a versão paralela do QuickSort está sujeita a menor overhead de latência de comunicação que as versões paralelas de RadixSort e Ordenação por Amostragem. A complexidade da versão paralela do QuickSort é O((n/p)log(n/p)), pois o conjunto de n elementos é divido entre os p processadores.

O Quicksort apresenta vantagem em relação a outros algoritmos de ordenação paralela, pois não necessita de sincronização. Cada sublista gerada é associada a um único processo, que não precisa se comunicar com os demais porque seus dados são independentes.

3.2.3 Ordenação no ambiente Hadoop

A ordenação de dados é uma das cargas de trabalho mais consideradas pelos *benchmarks* em geral, que buscam, a partir de uma entrada desordenada, obter uma saída ordenada e avaliar o desempenho do algoritmo que realizou a ordenação.

O *Sort* é um *benckmark* criado por por Jim Gray em 1998, e hoje é um dos mais conhecido na ordenação de dados (**??**). Consiste em um conjunto de seis *benchmarks*, cada um com as suas regras, que medem os tempos para ordenar diferentes números de registros e se diferem principalmente nas métricas de avaliação. As principais categorias dos *benchmarks Sort* são a *MinuteSort* e a *GraySort*. A categoria *MinuteSort* deve ordenar a maior quantidade dos dados em um minuto e a *GraySort* deve ordenar mais que 100 terabytes em pelo menos uma hora (4). Ainda existem as categorias *PennySort*, *JouleSort*, e os descontinuados *DatamationSort* e *TeraByte Sort*. Em cada categoria de ordenação, existem duas classificações, de acordo com o tipo de registro a ser ordenado: *Daytona* e *Indy*. O participantes da categoria *Daytona* são códigos de ordenação de propósito geral, e os participantes da *Indy* devem ordenar apenas registros de 100 bytes, sendo os primeiros 10 bytes de um registro a chave e o restante o valor do elemento a ser ordenado (**??**).

No Hadoop, o *Sort* é uma aplicação MapReduce, composta de três etapas: gerar dados aleatórios, realizar a ordenação, e validar os resultados. A geração de dados aleatórios é feita com o programa *RandomWriter*. Ele executa 10 tarefas MapReduce por nó, e cada função Map gera aproximadamente 1GB, totalizando 10GB de dados binários aleatórios. É possível alterar o número de dados e as configurações para os tamanhos das chaves e valores, alterando algumas configurações do *RandomWriter*. No segundo passo é realizada uma ordenação parcial dos dados de entrada, e o resultado é escrito em um diretório de saída. O passo final é validar os resultados obtidos pela ordenação dos dados realizada pelo *Sort*, através do programa *SortValidator*, que realiza uma série de verificações nos dados ordenados e nos não ordenados para confirmar se a ordenação foi realizada corretamente. Esse programa é considerado muito útil para verificar o desempenho do sistema como um todo, uma vez que todo o conjunto de dados é transferido através da aplicação.

O *TeraSort* é outra aplicação de destaque para ordenação de dados com Hadoop, criada por Owen O' Malley [O'Malley e Murthy 2009], com o intuito de participar da competição *Sort* [Gray 1998]. Em 2009, o *TeraSort* foi o campeão dessa competição em duas categorias: *MinuteSort* ao ordenar 500 GB em 59 segundos, utilizando um *cluster* com 1.406 nodos; e *GraySort* ordenando 100 TB em 173 minutos em um *cluster* com 3.452 nodos. A escalabilidade da solução foi provada pela ordenação de 1 PB em 975 minutos (equivalente a 16,25 horas) em 3.658 nodos. O *TeraSort* consiste de três algoritmos, que são responsáveis pela geração dos dados, ordenação e validação.

Teragen é o programa padrão para geração dados para a ordenação com *Terasort*. Nele o número de registros gerados é um parâmetro definido pelo usuário, assim como o número de tarefas Map a serem realizadas. O programa divide o número desejado de registros pelo número de tarefas Map, e atribui a cada tarefa Map um intervalo de chaves para a geração de um arquivo. Cada tarefa Map corresponde a um arquivo de saída, assim os dados gerados são divididos em diversos arquivos. Deste modo, se existem 2 tarefas Map, serão escritos 2 arquivos, cada um contendo metade das

chaves geradas. Os registros gerados têm um formato específico: uma chave, um id e um valor. As chaves são caracteres aleatórios do conjunto ' ' .. ''. O id é um valor inteiro que representa a linha, e o valor consiste de 70 caracteres de 'A' a 'Z'.

TeraSort é uma espécie MapReduce padrão, mas apresenta um particionador personalizado que usa uma lista ordenada de $N-1$ chaves amostradas que definem a faixa de chaves para cada função Reduce. Em particular, todas as chaves tal que $amostra[i-1] \leq chave < amostra[i]$ são enviadas para a função i. Isto garante todas as chaves da saída i sejam menores que as da saída $i+1$. Há também um formato de entrada e saída, *TeraOutputFormat*, que é utilizado por todas as 3 aplicações para ler e gravar os arquivos de texto no mesmo formato.

TeraValidate garante que a saída está totalmente ordenada. A aplicação cria uma função Map para cada arquivo no diretório de saída, que se certifica que cada chave é menor ou igual à anterior. O Map também gera registros com a primeira e última chave de cada arquivo. Em seguida, a função Reduce lê tais registros e garante que a primeira chave de um arquivo é maior do que a última chave do arquivo anterior. Caso alguma chave seja encontrada fora de ordem, ela é escrita em um arquivo de saída do Reduce.

4 Desenvolvimento

Esse capítulo trata do desenvolvimento do trabalho, indicando a metodologia utilizada, a infraestrutura disponibilizada para realização dos testes e ainda o cronograma das atividades propostas. Descreve também os experimentos realizados, com a apresentação dos algoritmos e distribuição dos dados da carga de trabalho.

4.1 Metodologia

A primeira fase do projeto foi destinada ao estudo mais detalhado da computação paralela, em especial dos algoritmos de ordenação paralela, do modelo MapReduce e da plataforma Hadoop. Foram executados testes com os exemplos disponibilizados pelo Hadoop e com os *benchmarks* TeraSort e Sort. Após esses testes, foram realizados experimentos com o algoritmo Ordenação por Amostragem, com dados em distribuição uniforme, mesma carga de dados utilizada no trabalho de Pinhão (2011). Além dessa carga de dados, foram feitos testes incluindo duas novas distribuições: Normal e Pareto.

Os testes buscavam mensurar o desempenho dos algoritmos com relação à quantidade de máquinas, quantidade de dados e conjunto de dados, uma vez que cada algoritmo implementado deve ser cuidadosamente avaliado para verificar um funcionamento adequado com diferentes entradas e número de máquinas. Os resultados obtidos foram analisados a fim de permitir comparar o desempenho dos algoritmos em cada situação.

O passo seguinte foi conhecer detalhadamente o algoritmo paralelo a ser implementado, o Quicksort. No próximo semestre serão definidas as estratégias para sua implementação em ambiente Hadoop e realizados os testes.

4.2 Infraestrutura

A infraestrutura necessária ao desenvolvimento do projeto foi fornecida pelo Laboratório de Redes e Sistemas (LABORES) do Departamento de Computação do Centro Federal de Educação Tecnológica de Minas Gerais (DECOM). O laboratório possui um *cluster* formado por cinco máquinas Dell Optiplex 380, que estão sendo utilizados na realização dos testes dos algoritmos.

Cada máquina do *cluster* apresenta as seguintes características:

- Processador Intel Core 2 Duo de 3.0 GHz
- Disco rígido SATA de 500 GB 7200 RPM
- Memória RAM de 4 GB
- Placa de rede Gigabit Ethernet
- Sistema operacional Linux Ubunbu 10.04 32 bits
- Sun Java JDK 1.6.0 19.0-b09
- Apache Hadoop 1.0.2

4.3 Descrição dos experimentos

A primeira parte dos experimentos consitiu em reproduzir os resultados já encontrados no trabalho de referência: testes de ordenação com os *benchmarks* TeraSort e Sort, e com o algoritmo Ordenação por Amostragem. Em todos os casos, os testes foram compostos de duas partes: geração da carga de dados e ordenação.

4.3.1 Benchmarks: TeraSort e Sort

Para compreender o funcionamento dos algoritmos de ordenação e do ambiente Hadoop foram executados, primeiramente, testes com os *benchmarks* TeraSort e Sort. A importância desses testes consiste no fato de tais aplicações serem conhecidas e consolidados no ambiente Hadoop. O objetivo dos testes foi verificar se as condições do ambiente estavam similares aos apresentados no trabalho de Pinhão (2011). Para tal, os parâmetros dos algoritmos e número de máquinas utilizadas foram similares.

O TeraSort consiste de três algoritmos, que são responsáveis pela geração dos dados, ordenação e validação, conforme descrito na seção **??**. Os testes com o Terasort foram feitos em duas máquinas. Foram gerados pelo TeraGen

Figura 5: Distribuição Uniforme

Figura 6: Distribuição Normal com média 5 e desvio padrão 1

dois arquivos contendo 50 mil linhas cada e o algoritmo foi executado 10 vezes.

Para os testes realizados com o Sort foram utilizados dados gerados pelo algoritmo RandomWriter. Para cada máquina do *cluster*, foram escritos 10 arquivos de 1GB cada em formato binário, totalizando 10GB. Os testes foram feitos em 4 máquinas, com 10 execuções.

4.3.2 Ordenação por Amostragem

O algoritmo Ordenação por Amostragem foi implementado em Java, no ambiente Hadoop.

A parte inicial dos experimentos com o algoritmo Ordenação por Amostragem foi reproduzir os testes realizados no trabalho de referência. Foram feitos três tipos de experimentos com o algoritmo, com alterações no número de arquivos ou máquinas e carga de dados com distribuição uniforme, semelhante à utilizada no trabalho de Pinhão (2011). O objetivo de cada experimento era avaliar o algoritmo em situações diversas, confirmando sua escalabilidade e eficiência.

Os experimentos podem ser divididos em três categorias. O primeiro experimento manteve constante o tamanho do arquivo a ser ordenado e o número de máquinas utilizadas na ordenação. O segundo experimento manteve constante o número de máquinas utilizadas e variou o tamanho do arquivo a ser ordenado. Já o terceiro experimento manteve constante o número de dados e alterou a quantidade de máquinas utilizadas.

Além disso, cada um dos experimentos foi realizado com três conjuntos diferentes de dados. Os dados utilizados foram números gerados aleatoriamente em três distribuições: uniforme, normal e pareto. As distribuições foram geradas por um programa implementado em Java para geração de chaves aleatórias de ponto flutuante, contendo entre 10^6 (12MB) e 10^{10} (120GB) chaves.

As Figuras 5, 6 e 7 exibem o padrão de frequência das chaves das três distribuições: uniforme, normal e pareto, respectivamente. Pode-se observar que as três distribuições tem padrões de comportamento bastante distintos, que se refletem nos arquivos de entrada gerados. A distribuição pareto é a única que está apresentada em escala logarítmica, para melhor visualização. Na distribuição uniforme cada chave tem a mesma probabilidade de aparecimento, levando a chaves igualmente distribuídas no intervalo. A distribuição normal concentra grande parte dos valores próximos à media, e distribui poucas chaves em valores mais extremos. A distribuição pareto possui uma grande concentração de chaves em valores próximos de 0, e pequena quantidade de valores mais altos, com baixa frequência de aparecimento no intervalo restante.

Uma parte fundamental do algoritmo de Ordenação por Amostragem é a definição dos parâmetros de amostragem de chaves, para uma amostragem que resulte em partições balanceadas. Nos testes realizados, os parâmetros definidos foram a frequência máxima de amostras e o número de partições para cada caso. A frequência das amostras foi fixada 10 mil, e o número de partições foi função do número de máquinas utilizadas e núcleos dos processadores: 4 partições para 2 máquinas; 6 para 3 máquinas; 8 para 4 máquinas; 10 para 5 máquinas.

Quantidade de máquinas e dados constante

Os testes foram realizados em 4 máquinas, com arquivos de 10^6 chaves. Foram feitos testes com 10 conjuntos de dados diferentes, e para cada conjunto, o algoritmo foi executado 10 vezes, com os parâmetros de balanceamento descritos anteriormente. O objetivo era avaliar a influência dos valores gerados aleatoriamente no desempenho do algoritmo.

Variando a quantidade de dados

Os testes variando a quantidade de dados também foram executados em 4 máquinas, com conjuntos de dados das três distribuições diferentes. Cada distribuição gerou aleatoriamente uma quantidade de dados entre 10^6 e 10^{10}. O algoritmo foi executado três vezes em cada conjunto com os parâmetros descritos anteriormente. O objetivo foi avaliar a complexidade do algoritmo quando o conjunto de dados a serem ordenados aumenta.

Figura 7: Distribuição Pareto com parâmetro 0.9

Variando a quantidade de máquinas

Esses testes foram executados com tamanho constante do arquivo de entrada (10^8 chaves) em quantidades de máquinas que variaram de 2 a 5. Para cada quantidade de máquinas, foram gerados conjuntos com as distribuições diferentes e o algoritmo foi executado três vezes para cada conjunto, com os parâmetros de balanceamento descritos anteriormente. O objetivo foi avaliar a escalabilidade do algoritmo, com diminuição do tempo de ordenação quando se aumenta o número de máquinas.

4.4 Cronograma de trabalho

O cronograma de trabalho inclui as tarefas que devem ser realizadas e como elas devem ser alocadas durante as disciplinas TCC I e TCC II para que o projeto possa ser concluído com sucesso. As atividades a serem desenvolvidas são descritas a seguir:

1. Pesquisa bibliográfica sobre o tema do projeto e escrita da proposta.
2. Estudo mais detalhado dos algoritmos de ordenação paralela, modelo MapReduce e Hadoop.
3. Configuração do ambiente Hadoop no laboratório.
4. Testes com exemplos do Hadoop e com algoritmo Ordenação por Amostragem.
5. Escrita, revisão e entrega do projeto.
6. Implementação e testes do Quicksort.
7. Análise comparativa dos resultados.
8. Escrita e revisão do relatório final.
9. Entrega e apresentação.

Na Tabela 1 está descrito o cronograma esperado para o desenvolvimento do projeto. Na disciplina TCC I foram realizadas as atividades 1 a 5, e as demais atividades serão realizadas em TCC II. Cada atividade foi alocada para se adequar da melhor maneira ao tempo disponível, mas é possível que o cronograma seja refinado posteriormente, com a inclusão de novas atividades ou redistribuição das tarefas existentes.

Atividade	Fev	Mar	Abr	Mai	Jun	Jul	Ago	Set	Out	Nov
1	•	•								
2		•	•							
3			•							
4			•	•						
5				•	•					
6						•	•			
7								•		
8									•	
9										•

Tabela 1: Cronograma proposto para o projeto

Referências

1 LIN, J.; DYER, C. *Data-Intensive Text Processing with MapReduce*. University of Maryland, College Park, Maryland: Morgan & Claypool Publishers, 2010. (Synthesis Lectures on Human Language Technologies).

2 ASANOVIC, K. et al. A view of the parallel computing landscape. *Commun. ACM*, ACM, New York, NY, USA, v. 52, n. 10, p. 56–67, out. 2009.

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

4 WHITE, T. *Hadoop: The Definitive Guide*. 1. ed. Sebastopol, CA, USA: O'Reilly, 2009.

5 SATISH, N.; HARRIS, M.; GARLAND, M. Designing efficient sorting algorithms for manycore gpus. In: *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009. p. 1–10.

6 AMATO, N. M. et al. *A Comparison of Parallel Sorting Algorithms on Different Architectures*. College Station, TX, USA, 1998.

7 GANTZ, J. *The Diverse and Exploding Digital Universe: An Updated Forecast of Worldwide Information Growth Through 2011*. Framingham, MA, USA: International Data Corporation, 2008.

8 MANFERDELLI, J. L.; GOVINDARAJU, N. K.; CRALL, C. Challenges and opportunities in Many-Core computing. *Proceedings of the IEEE*, Redmond, WA, USA, v. 96, n. 5, p. 808–815, may 2008.

9 BRYANT, R. E. Data-Intensive Scalable Computing for Scientific Applications. *Computing in Science and Engineering*, IEEE Computer Society, Los Alamitos, CA, USA, v. 99, n. PrePrints, 2011.

10 BRESHEARS, C. P. *The Art of Concurrency - A Thread Monkey's Guide to Writing Parallel Applications*. Sebastopol, CA, USA: O'Reilly, 2009. I-XIII, 1-285 p.

11 RANGER, C. et al. Evaluating mapreduce for multi-core and multiprocessor systems. In: *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2007. (HPCA '07), p. 13–24.

12 CHERKASOVA, L. Performance modeling in mapreduce environments: challenges and opportunities. In: *Proceedings of the second joint WOSP/SIPEW international conference on Performance engineering*. New York, NY, USA: ACM, 2011. (ICPE '11), p. 5–6.

13 PAGE, L. et al. *The PageRank Citation Ranking: Bringing Order to the Web*. 1999.

14 KALE, V.; SOLOMONIK, E. Parallel sorting pattern. In: *Proceedings of the 2010 Workshop on Parallel Programming Patterns*. New York, NY, USA: ACM, 2010. (ParaPLoP '10), p. 10:1–10:12.

15 ALMASI, G. S.; GOTTLIEB, A. *Highly parallel computing (2. ed.)*. Redwood City, CA, USA: Addison-Wesley, 1994. I-XXVI, 1-689 p.

16 RAUBER, T.; RüNGER, G. *Parallel Programming for Multicore and Cluster Systems*. Berlin, Heidelberg: Springer Verlag, 2010.

17 LEISERSON, C. E.; MIRMAN, I. B. *How to Survive the Multicore Software Revolution*. [S.l.], 2008.

18 TOTH, D. M. *Improving the Productivity of Volunteer Computing*. Tese (Doutorado) — Worcester Polytechnic Institute, 2008.

19 ZIVIANI, N. *Projeto de Algoritmos com Implementações em Java e C++*. São Paulo, Brazil: Thomson Learning, 2007. 641 p.

20 PINHÃO, P. de M. *Ordenação Paralela no Ambiente Hadoop*. 2011.