Centro Federal de Educação Tecnológica de Minas Gerais
Departamento de Computação
Engenharia de Computação

Mariane Raquel Silva Gonçalves

COMPARAÇÃO DE ALGORITMOS PARALELOS DE ORDENAÇÃO EM MAPREDUCE

Orientadora: Prof[a]. Dr[a]. Cristina Duarte Murta

Belo Horizonte
2012

Sumário

1 Introdução

Fazer aqui uma introdução geral da área do conhecimento à qual o tema escolhido está ligado.

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 [Lin e Dyer 2010]. 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 [Asanovic et al. 2009].

O MapReduce é um modelo de programação paralela desenvolvido pela Google para processamento de grandes volumes de dados distribuídos em *clusters* [Dean e Ghemawat 2008]. Esse modelo propõe simplificar a computação paralela, escondendo detalhes da paralelização do desenvolvedor e utilizando duas funções principais - map e reduce. Uma das implementações mais conhecidas e utilizadas do modelo é o Hadoop [White 2009], 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 [Satish et al. 2009, Amato et al. 1998]. 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.

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 este tema e busca continuar a análise, com a implementação do algoritmo Quicksort, no mesmo ambiente, bem como a análise de escalabilidade e comparação do desempenho dos algoritmos.

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. Não é fácil medir o volume total de dados armazenados digitamente, mas uma estimativa da IDC [Gantz 2008] calculou o tamanho do universo digital em 0,18 zettabytes em 2006, e previa um crescimento dez vezes até 2011, chegando a 1,8 zettabytes. *The New York Stock Exchange* gera cerca de um terabyte de novos dados comerciais por dia. O Facebook armazena aproximadamente 10 bilhões de fotos, que ocupam mais de um petabyte. *The Internet Archive* armazena aproximadamente 2 petabytes de dados, com aumento de 20 terabytes por mês [White 2009]. Estima-se que dados não estruturados são a maior porção e a de mais rápido crescimento dentro das empresas, o que torna o processamento de tal volume de dados muitas vezes inviável.

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 em tais condições. 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 [Manferdelli et al. 2008].

Além disso, as tendências atuais estão redirecionando o foco da computação, do tradicional modelo de processamento científico para o processamento de grandes volumes de dados. Arquiteturas de memória distribuída estão cada vez mais frequentes, suprindo a necessidade de substituir a computação tradicional por computação distribuída eficiente, cujo foco sejam os dados e que forneça computação de alto desempenho [Bryant 2011].

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 [Ranger et al. 2007].

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 [Ranger et al. 2007]. 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 [Cherkasova 2011].

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. Estima-se que a ordenação seja responsável por aproximadamente 80% dos ciclos de processamento. Mesmo com o grande processamento empregado em interfaces gráficas, visualização e jogos a ordenação continua a ser uma parte considerável da computação.

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 [Kale e Solomonik 2010]. 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 [Amato et al. 1998].

Assim, estudos experimentais assumem uma crescente importância para a avaliação e seleção de algoritmos apropriados para multiprocessadores. É preciso que mais estudos sejam realizados para que determinado algoritmo pode ser recomendado em certa arquitetura com alto grau de confiança.

1.3 Objetivos

Os objetivos deste trabalho são:

- Estudar a programação paralela aplicada à algoritmos de ordenação;
- Implementar um ou mais algoritmos de ordenação paralela no modelo MapReduce, com o software Hadoop;
- Comparar duas ou mais implementações de algoritmos paralelos de ordenação.

Este projeto busca continuar o estudo sobre ordenação paralela feito no trabalho 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 feita em ambiente Hadoop. A análise busca compará-los com relação à quantidade de dados a serem ordenados, variabilidade dos dados de entrada e número máquinas utilizadas.

1.4 Organização do Texto

Esse projeto está organizado em cinco capítulos. O próximo capítulo apresenta o referencial teórico para o desenvolvimento do trabalho. O Capítulo 3 descreve a metologia de pesquisa, indicando os passos seguidos durante o desenvolvimento. Os resultados preliminares obtidos até a entrega do projeto são apresentados no Capítulo 4. As conclusões e os próximos passos para a finalização do projeto estão no Capítulo 5.

2 Referencial Teórico

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

2.1 Computação Paralela

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 [Almasi e Gottlieb 1994]. Mesmo com o avanço tecnológico das últimas décadas, as arquiteturas sequenciais de Von Neumann ainda demostram deficiências quando utilizadas por aplicações que necessitam de grande poder computacional. Essa necessidade de maior poder computacional causou o surgimento da computação paralela, com a proposta de aumentar o poder computacional das máquinas.

No estudo de computação paralela, é importante diferenciar os conceitos de paralelismo e concorrência, que podem ser confundidos, pois ambos tratam de programação e execução de tarefas em múltiplos fluxos, implementados com o objetivo de resolver um único problema. Concorrência consiste de diferentes tarefas serem executadas ao mesmo tempo, de forma a produzir um resultado particular mais rapidamente. Isso não implica na existência em múltiplos 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 várias 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 [Breshears 2009].

Comparada à computação sequencial, a computação paralela apresenta alto desempenho e soluções mais naturais para problemas intrinsecamente paralelos, mas sua utilização também inclui algumas desvantagens. Há muito mais detalhes e diversidades no desenvolvimento, 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, o desenvolvimento de soluções paralelas apresenta maior dificuldade na programação, necessidade de balanceamento de cargas, sincronismo e intensa sobrecarga de comunicação [Rauber e Rünger 2010] .

O desenvolvimento de software paralelo introduz três principais desafios: assegurar confiabilidade de software, minimizar o tempo de desenvolvimento e conquistar bom desempenho na aplicação [Leiserson e Mirman 2008].

Manter a confiabilidade do sistema é essencial, pois ao se introduzir paralelismo a aplicação se torna vulnerável às condições de corrida, e dependendo da ordem de execução das tarefas o software 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 os torna difíceis de evitar e de encontrar durante testes.

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

O bom desempenho da aplicação é um objetivo central 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 são tipicamente as maiores barreiras para se atingir grande desempenho em programas paralelos, e devem ser minimizadas pelo desenvolvedor. Para avaliar o desempenho de algoritmos paralelos, algumas métricas definidas são largamente utilizadas. A lei de Amdahl e a eficiência são os principais indicadores de desempenho de algoritmos paralelos.

A lei de Amdahl demonstra que o ganho de desempenho que pode ser obtido melhorando uma parte do sistema é limitado pela fração de tempo em que essa parte é

utilizada pelo sistema. Pode ser utilizada para calcular o ganho de desempenho de uma máquina, conhecido como *speedup* [Amdahl 1967]. O *speedup* determina a relação existente entre o tempo gasto para executar um algoritmo de maneira sequencial em um único processador ($T_{sequencial}$) e o tempo gasto para executá-lo em p processadores ($T_{paralelo}$): $speedup = T_{sequencial}/T_{paralelo}$. O *speedup* ideal é aquele igual a p, que significaria um aumento da capacidade de processamento é diretamente proporcional ao número de processadores. O resultado do *speedup* pode ser diretamente afetado por fatores como comunicação entre processos, granulosidades inadequadas e partes não paralelizáveis de programas.

A eficiência é outro parâmetro utilizados para medir o desempenho quando se utiliza computação paralela. Ela relaciona o *speedup* ao número de processadores, identificando a taxa de utilização do processador. Pode ser calculada por: $E_p = speedup/p$. Em uma paralelização ideal, a eficiência tem valor 1, significando que os processadores têm utilização total.

2.1.1 Modelos de programação paralela

Um modelo de programação 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, passagem de mensagens, paralelismo de dados e de tarefas (*threads*/*multithreads*).

No ambiente de memória compartilha, 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 consiste em vários processadores, cada um com sua própria memória, interconectados por uma rede de comunicação, como repre-

sentado na Figura 2.1. Nesse modelo as tarefas compartilham dados por meio de comunicação de envio e recebimento de mensagens. Essa passagem de mensagens pode ser realizada por meio de bibliotecas como a MPI (*Message Passing Interface*) e a PVM (*Parallel Virtual Machine*) [Rauber e Rünger 2010].

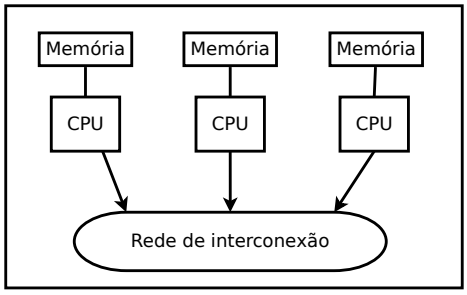

Figura 2.1: Modelo de arquitetura distribuída

2.1.2 Computação paralela em *clusters*

Com o avanço tecnólogico da última década, o volume crescente de dados sendo gerado, coletado e armazenado 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* [Lin e Dyer 2010].

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 [Toth 2008].

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 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 para evitar que os processadores fiquem ociosos, através de redes de alta velocidade [Rauber e Rünger 2010].

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. O processamento deve ser baseado 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 acessar os dados. 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 paralelo 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 [Bryant 2011].

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-desempenho.

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 balanço de carga - e utilizando duas funções principais: Map e Reduce. A complexidade do algoritmo paralelo não é vista pelo desenvolvedor, que pode se ocupar em desenvolver a solução proposta [Dean e Ghemawat 2008].

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 saída é um conjunto de pares no formato (chave, valor). 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 é então aplicada para todos os pares intermediários com a mesma chave. A função combina esses valores para formar um conjunto menor de resultados. Tipicamente, há apenas zero ou um valores de saída em cada função Reduce.

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 texto, 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 2.2 ilustra o fluxo de execução para este exemplo. A entrada é um arquivo contendo as linhas "hadoop conta", "conta palavras"e "exemplo hadoop".

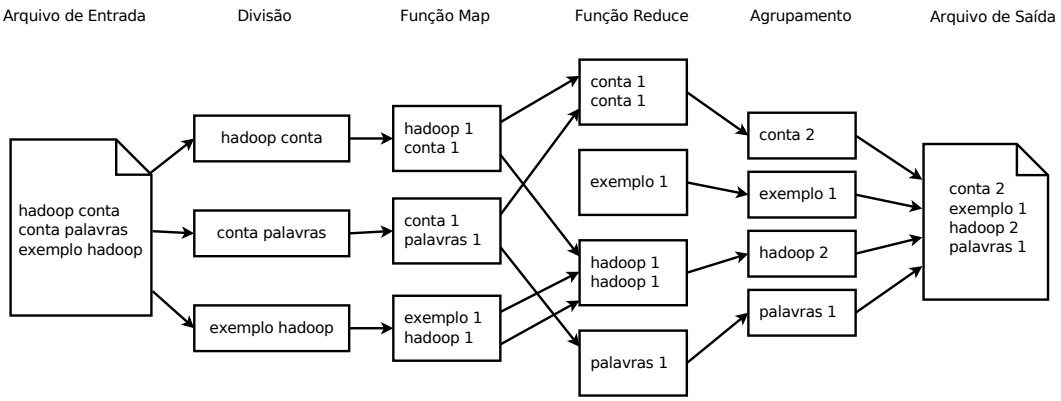

Figura 2.2: Fluxo simplificado da contagem de palavras com o MapReduce

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 execu-

tam 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 2.3 ilustra uma o fluxo de uma execução do modelo MapReduce [Dean e Ghemawat 2008]. 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.

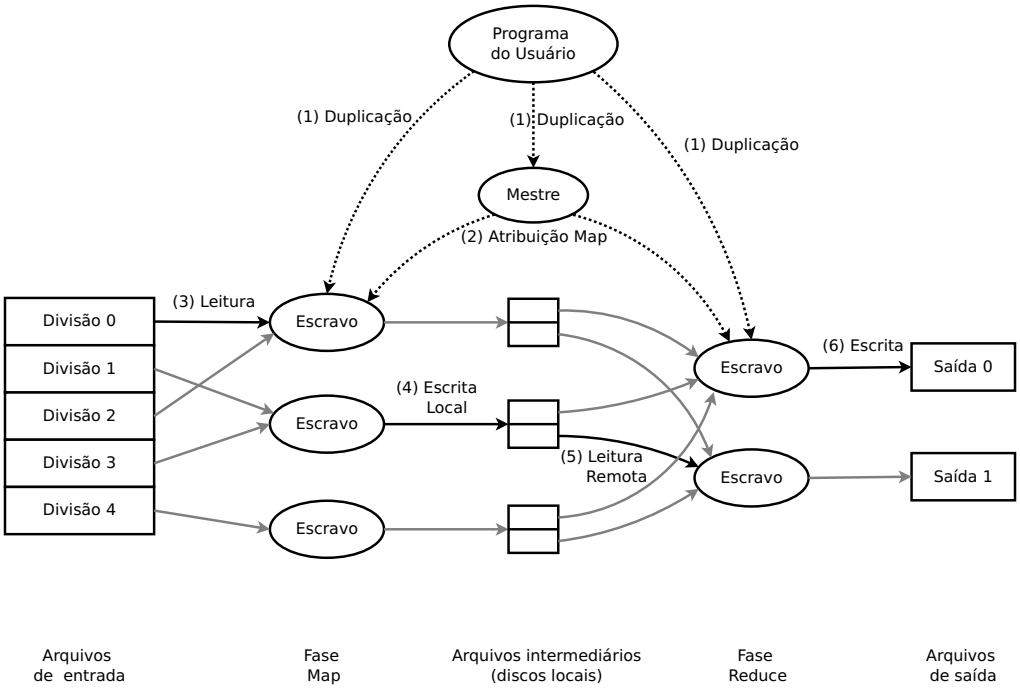

Figura 2.3: Visão geral do funcionamento do modelo MapReduce.

1. A biblioteca MapReduce, no programa do usuário, primeiro 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 analisa 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 regiões R 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 no *buffer*, a partir dos discos locais dos trabalhadores Map. Quando um trabalhador Reduce tiver lido todos os dados intermediários para sua partição, ela é ordenada pela chave intermediária para que todas as ocorrências da mesma chave sejam 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 intermediária única encontrada, passa 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. Neste ponto, a chamada MapReduce no programa do usuário retorna para o código do usuário.

2.3 Hadoop

Uma das implementações mais conhecidas do MapReduce é o Hadoop, desenvolvido por Doug Cutting em 2005 e mantido pela Apache Software Foundation. O

Hadoop é uma implementação código aberto em Java do modelo criado pela Google, que provê o gerenciamento de computação distribuída, de maneira escalável e confiável [White 2009].

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 [Cherkasova 2011].

O modelo MapReduce foi criado para permitir o processamento em conjuntos de centenas de máquinas de maneira transparente, o que significa que o usuário não deve ser preocupar com mecanismos de tolerância a falhas, que deve ser provido pelo sistema [Dean e Ghemawat 2008]. Um dos principais benefícios do Hadoop é a implementação desse mecanismo de tolerância a falhas, sejam elas de disco, processos, ou de nós, o que permite que o trabalho do usuário possa ser concluído.

O sistema é capaz de verificar e substituir nós quando ocorre alguma falha. O nó mestre envia mensagens periódicas aos demais nós para verificar seus estados. Se nenhuma resposta é recebida, o mestre identifica que houve uma falha neste nó. 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 [White 2009].

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 [White 2009]. A plataforma Hadoop 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, mas fornece o HDSF como sistema de arquivos padrão.

A arquitetura do HDFS também é do tipo mestre-escravos. O nó mestre (*NameNode*) é responsável por manter e controlar todos os metadados do sistema de arquivos e gerenciar a localização dos dados. Também é responsável por outras atividades, como por exemplo, balanceamento de carga, *garbage collection*, e atendimento a requisições dos clientes. Os nós escravos (*DataNode*) são responsáveis por armazenar e transmitir os

dados aos usuários que os requisitarem.

A Figura 2.4 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 arquivos cliente. Os conceitos de nó *Master* e *Worker* do MapReduce, são respectivamente denominados *JobTracker* e *TaskTracker* no Hadoop.

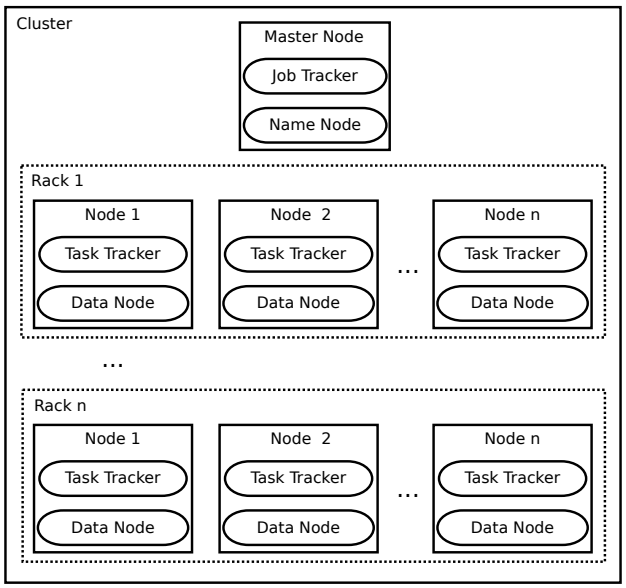

Figura 2.4: Visão abstrata do cluster.

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. A outra funcionalidade é a distribuição dos dados. O sistema de arquivos busca manter um balanceamento na ocupação das unidades de armazenamento, e o *framework* busca atribuir tarefas a um *worker* 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.

2.4 Ordenação

A ordenação é o processo de organizar elementos de uma sequência em determinada ordem, e é um dos problemas fundamentais na computação devido à sua importância teórica e prática. 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 [Ziviani 2007].

2.5 Algoritmos de Ordenação Paralela

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 ordenação feita por cada processo individual seja organizada de tal forma que todas

as unidades de processamento estejam trabalhando, enquanto o custo de redistribuição de chaves entre os processadores é 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 [Kale e Solomonik 2010]:

- **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 opera-

ções 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 às propriedades do conjunto de elementos a ser ordenado. Considerando um conjunto de elementos $\tau = k_1, k_2, ..., k_n$ distribuído entre p processadores, é preciso que durante a execução de qualquer algoritmo de ordenação paralela todas as chaves da sequência inicial sejam preservadas, ou seja, que não se perca nenhuma chave durante a distribuição entre os processadores. É necessário ainda que o conjunto de chaves seja particionada em p subconjuntos mutualmente exclusivos, sem nenhuma chave duplicada e que o conjunto de todas as chaves satisfaça as propriedades de um conjunto parcialmente ordenado (Adicionar propriedades do conjunto parc ord).

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 ao menor chave no processador p_{i+1} e se a saída resultante é uma sequência de chaves totalmente ordenada.

2.5.1 Fluxo geral de execução da ordenação paralela

Na execução de um algoritmo de ordenação paralela, podem ser identificadas algumas tarefas principais, normalmente realizadas de forma sequencial, que todos os algoritmos precisam executar em algum momento [Kale e Solomonik 2010]. 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, de maneira superficial, 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 funções 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. O custo da comunicação necessária em um algoritmo (para determinar a divisão e mover os dados) e o custo do processamento local que pode ser sobreposto à essa comunicação é um bom indicativo para comparação da escalabilidade dos algoritmos de ordenação paralela.

2.5.2 Sample Sort

O algoritmo *Sample Sort*, 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 estimar o número de elementos que devem ser destinados a uma certa partição, que é feita 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, assim, 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 seleciona-

das 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 utilizado na implementação do algoritmo feito por Pinhão (2011) e 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.

// descrever o algoritmo em Map Reduce

2.5.3 Quick Sort

2.5.4 Ordenação no ambiente Hadoop

// sort e terasort

3 Desenvolvimento

3.1 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 realizados testes de ordenação com os *benchmarks* TeraSort e Sort, e com os exemplos disponibilizados pelo Hadoop. Após esses testes, foram realizados testes com o algoritmo Ordenação por Amostragem do trabalho de Pinhão (2011), mas com com a inclusão das distribuições Normal e Pareto. O passo seguinte foi conhecer detalhadamente o algoritmo paralelo a ser implementado. No próximo semestre serão definidas as estratégias para sua implementação em ambiente Hadoop e realizados os testes.

O algoritmo implementado deve ser cuidadosamente avaliado para verificar um funcionamento adequado com diferentes entradas e número de máquinas. Dessa forma, foram realizados experimentos para testes de desempenho dos algoritmos com relação à quantidade de máquinas, quantidade de dados e conjunto de dados. Os resultados obtidos foram analisados a fim de permitir comparar a desempenho dos algoritmos em cada situação.

3.2 Infraestrutura

A infraestrutura necessária ao desenvolvimento do projeto foi fornecida pelo Laboratório de Redes e Sistemas (LABORES) do Departamento de Computação (DECOM). O laboratório possui um *cluster* formado por cinco máquinas Dell Optiplex 380, que foram utilizados na realização dos testes dos algoritmos. Os algoritmos serão desenvolvidos em linguagem Java, de acordo com o modelo MapReduce, no ambiente Hadoop.

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

3.3 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.

3.3.1 Benchmarks: TeraSort e Sort

Os *benchmarks* TeraSort e Sort foram os primeiros testes de ordenação realizados. O uso de algoritmos conhecidos e consolidados na ordenação no ambiente Hadoop permitiu compreender o funcionamento dos algoritmos e do ambiente dos testes.

Terasort

O TeraSort consiste de três algoritmos, que são responsáveis pela geração dos dados, ordenação e validação. A geração dos dados é feita pelo algoritmo TeraGen. Os registros gerados têm um formato específico, formado por 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’. O número de registros gerados é um parâmetro definido pelo usuário e os dados gerados são divididos em dois arquivos. O TeraSort lê os arquivos gerados e realiza a ordenação. Após a ordenação, os dados são validados pelo TeraValidade. Caso haja algum erro na ordenação, o algoritmo escreve um arquivo informando quais foram as chaves com erros.

Sort

Sort é um dos *benchmarks* de ordenação de dados mais conhecidos para Hadoop. Ele é uma aplicação MapReduce, que realiza uma ordenação dos dados de entrada. Além da ordenação, é fornecido um programa padrão para geração de dados aleatórios de entrada, o RandomWriter. Os dados utilizados para os testes de ordenação com o Sort foram gerados pelo algoritmo RandomWriter. Para cada máquina do *cluster*, são escritos 10 arquivos de 1GB cada em formato binário, totalizando 10GB.

3.3.2 Ordenação por Amostragem

Para o algoritmo ordenação por amostragem foram feitos três tipos de experimentos, com alterações no número de arquivos ou máquinas. 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. E o terceiro experimento manteve constante o número de dados e alterou a quantidade de máquinas utilizadas.

Cada um dos experimentos foi realizado com três distribuições diferentes: 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.

É parte fundamental do algoritmo de Ordenação por Amostragem a definição de parâmetros que resultem 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 (2 máquinas); 6 (3 máquinas); 8 (4 máquinas); 10 (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.

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.

3.4 Cronograma de trabalho

O cronograma de trabalho inclui as atividades 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 tarefas a serem desenvolvidas estão 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. Implementação e testes.
5. Escrita, revisão e entrega do relatório.
6. Análise comparativa entre os resultados.
7. Escrita e revisão do projeto final.
8. Entrega e apresentação.

Na Tabela 3.1 está descrito o cronograma esperado para o desenvolvimento do projeto. 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										•

Tabela 3.1: Cronograma proposto para o projeto

4 Resultados Preliminares

Neste capítulo são apresentados e analisados os resultados obtidos nessa fase do projeto, de acordo com os testes descritos anteriormente.

4.1 Benchmarks: TeraSort e Sort

A Tabela 4.1 apresenta os resultados obtidos para o *benchmarks* TeraSort.

Tabela 4.1: Resultados do benchmark TeraSort para execução em 2 máquinas

Algoritmo	Tempo (seg)	Tarefas Map	Tarefas Reduce
TeraGen	14	2	0
TeraSort	22	2	1
TeraValidade	22	1	1

A Tabela 4.2 apresenta os resultados obtidos para o *benchmarks* Sort.

Tabela 4.2: Resultados do benchmark Sort para execução em 4 máquinas

Algoritmo	Tempo (seg)	Tarefas Map	Tarefas Reduce
RandomWriter	234	40	0
Sort	2242	640	7

4.2 Algoritmo Ordenação por Amostragem

Os testes feitos com o algoritmo Ordenação por Amostragem tinham como objetivo reproduzir os resultados encontrados no trabalho feito por Pinhão (2011), e gerar resultados que serão utilizados posteriormente na comparação de desempenho dos algorit-

mos. O resultado dos experimentos está separado de acordo com o tipo de teste realizado, com variação do conjunto de dados, da quantidade de dados ordenada e da quantidade de máquinas utilizadas.

Diferentes quantidades de dados

A Tabela 4.3 apresenta os tempos médios de 6 execuções.

Tabela 4.3: Resultados da ordenação diferentes quantidade de dados 4 máquinas

Dados	Tamanho em bytes	Tempo Médio (seg)	
		Uniforme	Normal
10^6	12MB	39	41
10^7	120MB	51	56
10^8	1.2GB	221	231
10^9	12GB	1816	1893
10^{10}	120GB	17964	18165

Diferentes quantidades de máquinas

5 Conclusões e Propostas de Continuidade

// Conclusão

Como proposta de continuidade do projeto, está a implementação e teste do algoritmo Quicksort. Em seguida, serão comparados os desempenhos dos algoritmos nos cenários propostos, variando os conjuntos de dados, a quantidade de dados e a quantidade de máquinas utilizadas na ordenação. Nos diferentes cenários, serão feitas ordenações utilizando conjuntos com diferentes distribuições de chaves, para simular situações reais em que os dados nem sempre seguem uma distribuição uniforme.

Os resultados finais poderão auxiliar na escolha do melhor algoritmo para uma determinada situação, de acordo com o que se conhece dos dados a serem ordenados.

Referências Bibliográficas

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

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

[Amdahl 1967]AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In: *Proceedings of the April 18–20, 1967, spring joint computer conference—AFIPS '67 (Spring)*. New York, USA: ACM Press, 1967. p. 483–485.

[Asanovic et al. 2009]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.

[Breshears 2009]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.

[Bryant 2011]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.

[Cherkasova 2011]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.

[Dean e Ghemawat 2008]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.

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

[Kale e Solomonik 2010]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.

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

[Lin e Dyer 2010]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).

[Manferdelli et al. 2008]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.

[Pinhão 2011]PINHÃO, P. M. *Ordenação Paralela no Ambiente Hadoop*. 2011.

[Ranger et al. 2007]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.

[Rauber e Rünger 2010]RAUBER, T.; RÜNGER, G. *Parallel Programming for Multicore and Cluster Systems*. Berlin, Heidelberg: Springer Verlag, 2010.

[Satish et al. 2009]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.

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

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

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