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, disponíveis) 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. Muitas aplicações dependem de ordenações eficientes como base para seu próprio desempenho. A ordenação é um problema que abrange desde sistemas de banco de dados à computação gráfica, e muitos outros algoritmos podem ser descritos em termos de ordenação [Satish et al. 2009, Amato et al. 1998].

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 no ambiente Hadoop o algoritmo de Ordenação por Amostragem, 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 em indústrias, empresas e até mesmo em âmbito pessoal aumenta a cada ano. O desenvolvimento de soluções capazes de lidar com tais volumes de dados é uma das preocupações atuais, tendo em vista a quantidade de dados processados diariamente, e o rápido crescimento desse volume de dados. 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 deu-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, e 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. Cria-se assim 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].

A técnicas tradicionais de programação paralelas - 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 e 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 maior benefício desse modelo é a simplicidade. 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 amplamente utilizadas no processamento de terabyte e petabyte de dados [Ranger et al. 2007]. MapReduce e sua implementação *open source* 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].

// texto conector

A ordenação é um dos problemas fundamentais da ciência da computação e algoritmos paralelos para ordenação têm sido estudados desde o início da computação paralela. Os algoritmos ótimos existentes em arquitetura sequencial, como Quick Sort e Heap Sort necessitam de um tempo mínimo *(n log n)* para ordenar uma sequência de n elementos [Aho et al. 1974].

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.

O trabalho desenvolvido por Pinhão (2011) apresentou 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.

Este projeto busca continuar o estudo sobre ordenação paralela feito no trabalho citado, com a análise de desempenho dos algoritmos de ordenação ordenação por amostragem e quick sort. 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 a serem seguidos durante o desenvolvimento. Os resultados preliminares obtidos até a entrega do projeto são apresentados no Capítulo 4. As conclusões obtidas até o momento e os próximos passos para a conclusão do projeto estão no Capítulo 5.

2 Referencial Teórico

2.1 Processamento Distribuído ? / Computação Paralela ?

Com o avanço tecnólogico da última década, o volume crescente de dados sendo gerado, coletado e armazenado tornou o processamento dos dados inviável para 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. 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 homogêneas [**?**].

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 interligadas, garantindo grande desempenho e melhor custo-benefício que supercomputadores, o que apresenta grande vantagem. Outro ponto importante é que novas máquinas podem ser facilmente incorporadas ao *cluster*, tornando-o uma solução mais flexível, principalmente por ser formado por maquinas de capacidade de processamento similar.

Esta linha de pesquisa envolve vários outros conceitos relacionados à infraestrutura, como comunicação entre os nós, balanceamento de carga e outros discutidos nas próximas seções.

2.1.1 Princípios de processamento em cluster? distribuído? de grande volumes de dados?

O processamento de grandes volumes de dados em *clusters* deve suportar alguns princípios para garantir a escalabilidade e o bom desempenho [Bryant 2011]:

Tratamento de dados intrínsecos A coleta e manutenção dos dados deve ser funções do sistema e não tarefa dos usuários. O sistema deve recuperar informações atualizadas através de rede e realizar cálculos derivados como tarefas em segundo plano. Os usuários devem ser capazes de usar consultas ricos com base no conteúdo e identidade para acessar os dados. Mecanismos de confiabilidade, como replicação e de correção de erros devem ser incorporados como parte do sistema, de modo a garantir integridade e disponibilidade dos dados.

Modelo de programação paralelo de alto nível O desenvolvedor da aplicação deve fazer uso de primitivas de programação de alto nível, capazes de expressar formas naturais de paralelismo, que não incluam cofigurações específicas de uma máquina. O trabalho de mapear essas computações para a máquina de forma eficiente deve ficar a cargo do sistema, compilador e runtime.

Acesso interativo Os usuários devem ser capazes de executar programas de forma interativa, com variação dos requisitos de computação e armazenamento O sistema deve responder a consultas e cálculos simples rapidamente, e responder aos complexos sem degradar o desempenho geral. Para suportar a computação interativa, deve haver oferta de recursos. O custo consequente do aumento dos recursos ofertados pode ser justificado com base no aumento da produtividade dos usuários do sistema.

Mecanismos escaláveis para garantir alta confiabilidade e disponibilidade 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.

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. 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.

Código 2.1: Pseudocódigo do exemplo de contagem de palavras com o MapReduce

```
1 Function Map (Integer chave, String valor):
2 #chave: numero 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 funcao 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 2.1 ilustra o fluxo de execução para este exemplo. A entrada é um arquivo contendo as linhas "Exemplo conta palavras"e "Hadoop exemplo palavras".

Figura 2.1: Fluxo simplificado da contagem de palavras com o MapReduce

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 é inclui um sistema de arquivos distribuídos, onde ficam armazenados os dados de entrada e intermediários.

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 do particionamento do conjunto intermediário de pares em R partes. O número de partições R pode ser definido pelo usuário.

A Figura 2.2 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.

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 em um cluster de máquinas.

2. Uma das cópias do programa é especial: o mestre (*master*). Os demais são trabalhadores (escravos, slaves) 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. 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;

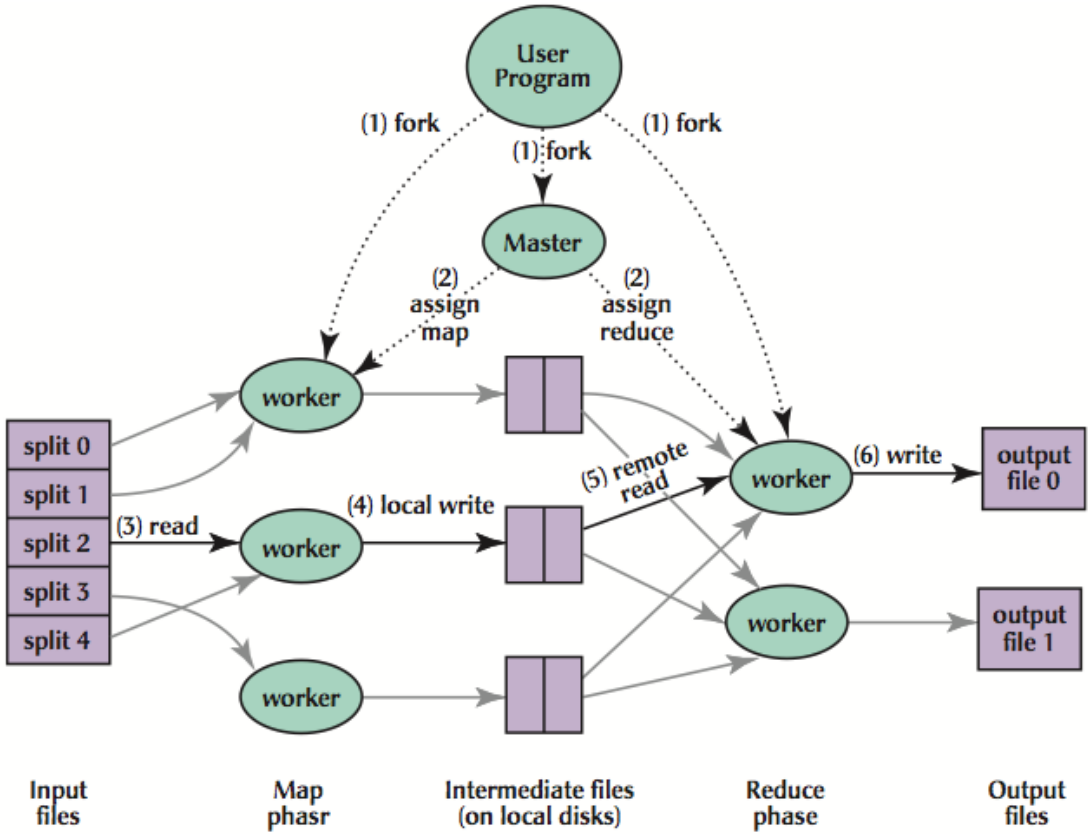

Figura 2.2: Visão geral do funcionamento do modelo MapReduce.

5. 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;

6. 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;

7. O trabalhador Reduce itera sobre os dados intermediários ordenados e, para cada chave intermediária única encontrada, passa a chave e o conjunto corres-

pondente 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;

8. Quando todas as tarefas Map e Reduce são concluídas, o mestre acorda o programa do usuário. Neste ponto, a chamada MapReduce no programa do usuário retorna para o código do usuário.

2.2.1 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 sua capacidade de lidar com falhas, sejam de disco, processos, ou de nós, e permitir 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 seu estado. 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].

HDFS

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 suporta diversos sistemas de arquivos distintos, como Amazon S3 (Native e Block-based), CloudStore, HAR, sistemas mantidos por servidores FTP e HTTP, Local (destinado a unidades de armazenamento conectadas localmente), 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 2.3 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.

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 DataNodes que pertencem a um mesmo *rack*, e distribuir as réplicas de maneira mais inteligente, aumentando a performance e confiabilidade do sistema. A outra 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 significante de nós, a maioria dos dados são lidos localmente e o consumo de banda é mínimo.

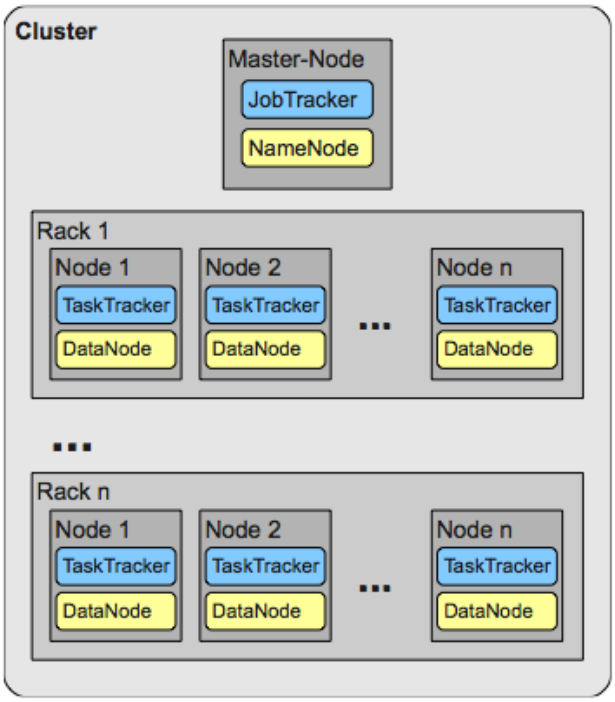

Figura 2.3: Visão abstrata do cluster.

2.3 Ordenação

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 a ordenação é feita em um grande conjunto de dados, que não podem ser armazenados em memória principal a ordenação é chamada 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 é feito em discos, cujo tempo de acesso é muito 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].

2.4 Algoritmos de Ordenação Paralela

Condições de implementação de algoritmos paralelos de ordenação

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 da 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.

2.4.1 Sample Sort

2.4.2 Quick Sort

3 Desenvolvimento

3.1 Metodologia

A primeira fase do projeto será destinado ao estudo mais detalhado da computação paralela, em especial os algoritmos de ordenação paralela, dos fatores que influenciam o desempenho de tais algoritmos, o modelo MapReduce e a plataforma Hadoop. O passo seguinte é conhecer detalhadamente o algoritmo paralelo a ser implementado e definir as estratégias para sua implentação ambiente Hadoop. O algoritmo implementado deve ser cuidadosamente avaliado para verificar um funcionamento adequado com diferentes entradas e número de máquinas.

Em seguida, serão realizados experimentos para testes de desempenho dos algoritmos com relação à quantidade de máquinas, quantidade de dados e conjunto de dados. Os resultados obtidos serão analisados e permitirão comparar a desempenho dos algoritmos em cada situação.

3.2 Infraestrutura

A infra estrutura necessária ao desenvolvimento do projeto será 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 serão utilizadas 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 são compostos de duas partes: geração da carga de dados, seguida da ordenação.

3.3.1 Testes com 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 permite 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, descrito na Figura **??** (incluir figura!). O registro é formado por uma chave, um id e um valor.

Chave as chaves são caracteres aleatórios do conjunto ' ' .. ' '.

Id um valor inteiro

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. Nos testes realizados, foram gerados dois arquivos, cada um contendo 50 mil linhas.

O TeraSort lê tais arquivos 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 Testes com o Algoritmo Ordenação por Amostragem

(A escrever)

(GeraDados) Programa implementado em Java para geração de chaves inteiras aleatórias. Foram gerados 10 conjuntos de chaves entre 10^6 (2MB) e 10^{10} (20GB) chaves inteiras. Anotar os tempos de execução de cada algoritmo.

Variando o conjunto de dados

(4 máquinas)

Objetivo: avaliar a influência dos valores gerados aleatoriamente no desempenho do algoritmo. Testes com 10 conjuntos de 10^6 dados. Para cada conjunto, executar 10 vezes com os parâmetros de balanceamento descritos anteriormente.

Variando a quantidade de dados

(4 máquinas)

Objetivo: avaliar a complexidade do algoritmo quando o conjunto de dados a serem ordenados aumenta. Testes com dados de 10^6 a 10^{10} gerados aleatoriamente.

Para cada quantidade de dados, o algoritmo foi executado três vezes com os parâmetros descritos anteriormente.

Variando a quantidade de máquinas

(2 a 5 máquinas)

Objetivo: avaliar a escalabilidade do algoritmo (diminuição do tempo de ordenação) Testes com o mesmo conjunto de 10^8 dados em diferentes quantidades de máquinas, de 2 a 5.

Para cada quantidade de máquinas, o algoritmo foi executado três vezes, com os parâmetros de balanceamento descritos anteriormente.

Parâmetros do algortimo

Frequência: Número max de amostras: 10 mil Núm max de partições: 4 (5 máquinas) 6 (5 máquinas) 8 (5 máquinas) 10 (5 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

Tabela 4.1: Tempos de Execução com diferentes quantidades de dados

4 Máquinas												
Dados		**EXECUÇÃO 1**				**EXECUÇÃO 2**				**EXECUÇÃO 3**		
		Tempo	Map	Reduce		Tempo	Map	Reduce		Tempo	Map	Reduce
10^6												

5 Conclusões e Propostas de Continuidade

Referências Bibliográficas

[Aho et al. 1974]Aho, A. V.; Hopcroft, J. E.; Ullman, J. D. *The Design and Analysis of Computer Algorithms.* : Addison-Wesley, 1974.

[Amato et al. 1998]Amato, N. M.; Iyer, R.; Sundaresan, S.; Wu, Y. *A Comparison of Parallel Sorting Algorithms on Different Architectures*. College Station, TX, USA, 1998.

[Asanovic et al. 2009]Asanovic, K.; Bodik, R.; Demmel, J.; Keaveny, T.; Keutzer, K.; Kubiatowicz, J.; Morgan, N.; Patterson, D.; Sen, K.; Wawrzynek, J.; Wessel, D.; Yelick, K. A view of the parallel computing landscape. *Commun. ACM*, ACM, New York, NY, USA, v. 52, n. 10, p. 56–67, out. 2009.

[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*. 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.

[Lin e Dyer 2010]Lin, J.; Dyer, C. *Data-Intensive Text Processing with MapReduce*. : 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*, , v. 96, n. 5, p. 808–815, may 2008.

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

[Ranger et al. 2007]Ranger, C.; Raghuraman, R.; Penmetsa, A.; Bradski, G.; Kozyrakis, C. 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.

[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.

[White 2009]White, T. *Hadoop: The Definitive Guide*. first edition. : O'Reilly, 2009.