UNIVERSIDADE FEEVALE

MARCELO CORREIA FLORES

COMPARAÇÃO DE DESEMPENHO ENTRE ALGORITMOS DE ORDENAÇÃO SEQUENCIAIS E PARALELOS

Novo Hamburgo

2015

MARCELO CORREIA FLORES

COMPARAÇÃO DE DESEMPENHO ENTRE ALGORITMOS DE ORDENAÇÃO SEQUENCIAIS E PARALELOS

Trabalho de Conclusão de Curso
apresentado como requisito parcial
à obtenção do grau de Bacharel em
Ciência da Computação pela
Universidade Feevale

Orientador: Juliano Varella de Carvalho

Novo Hamburgo
2015

AGRADECIMENTOS

Gostaria de agradecer a todos aqueles que de alguma forma, contribuíram para a realização desse trabalho de conclusão.

Agradeço aos amigos, família e professores que me apoiaram até mesmo nos momentos mais difíceis deste trabalho.

RESUMO

Manipular grandes massas de informações ainda é um desafio para as ferramentas atuais de manipulação de dados. As gigantes IBM e SAP estão envolvidas neste segmento de mercado com ferramentas robustas que prometem auxiliar a manipulação deste grande volume de dados denominado de *Big Data*. Nascido na Google, o MapReduce é um modelo de programação simples que promete auxiliar o segmento de *Big Data*. O Apache Hadoop é uma ferramenta que implementa este modelo de programação. Nele, é possível desenvolver algoritmos que executam em ambientes distribuídos. Desta forma, este trabalho visa comparar os tradicionais algoritmos de ordenação, levando em consideração o seu desempenho em um ambiente convencional (sequencial), em relação ao desempenho destes em um ambiente MapReduce com Apache Hadoop.

Palavras-chave: *Big Data*. MapReduce. Apache Hadoop. Algoritmos de ordenação.

ABSTRACT

Manipulate large masses of information is still a challenge for the current tools of data manipulation. IBM and SAP giants are involved in this market segment with robust tools that promise to help the handling of this large volume of data called Big Data. Born in Google, MapReduce is a simple programming model that promises to aid the Big Data segment. Apache Hadoop is a tool that implements this programming model. In it, you can develop algorithms that run in distributed environments. Thus, this study aims to compare the traditional sorting algorithms, taking into account their performance in a conventional environment (sequential), regarding the performance of these on a MapReduce environment with Apache Hadoop.

Key words: Big Data. MapReduce. Apache Hadoop. Sorting algorithms.

LISTA DE FIGURAS

Figura 1.1 - Modelo do algoritmo do contador de palavras ______ 17
Figura 2.1 - Algoritmo de ordenação *Bubble Sort* ______ 23
Figura 2.2 - Exemplo de ordenação com *Bubble Sort* ______ 23
Figura 2.3 - Exemplo de ordenação com *Selection Sort* ______ 24
Figura 2.4 - Exemplo de ordenação com *Insertion Sort* ______ 24
Figura 2.5 - Exemplo de ordenação com *Merge Sort* ______ 25
Figura 2.6 - Exemplo de ordenação com *Quick Sort* ______ 26
Figura 2.7 - Exemplo de ordenação com *Shell Sort* ______ 27
Figura 2.8 - Tempo para ordenação de 10^8 dados variando o número de partições ______ 29
Figura 2.9 - Resultados para variação do número de máquinas de processamento ______ 29
Figura 3.1 - Trecho do arquivo utilizado para ordenação ______ 31
Figura 3.2 - Algoritmo sequencial *Merge Sort* ______ 33
Figura 3.3 - Algoritmo sequencial *Insertion Sort* ______ 34
Figura 3.4 - Função Map() no contexto de ordenação no *Map* ______ 35
Figura 3.5 - Função Reduce() no contexto de ordenação no *Map* ______ 36
Figura 3.6 - Função *Map*() no contexto de ordenação no *Reduce* ______ 36
Figura 3.7 - Função *Reduce*() no contexto de ordenação no *Reduce* ______ 37
Figura 4.1 - Amostra de arquivo ordenado ______ 39
Figura 4.2 - Interface WEB do Apache Hadoop ______ 42
Figura 4.3 - Gráfico de tempos de execução do *Insertion Sort* ______ 44
Figura 4.4 - Gráfico de tempos de execução do *Merge Sort* ______ 48
Figura 4.5 - Gráfico de tempos de execução do *Merge Sort* na função *Map* ______ 49
Figura 4.6 - Gráfico do tempo de execução do arquivo de 1 GB ______ 52
Figura 4.7 – Lista de arquivos de saída com várias tarefas *Reduce* ______ 53
Figura 4.8 – Exemplos de arquivos de saída com várias tarefas *Reduce* ______ 54
Figura 4.9 - Processo MapReduce para ordenação no *Reduce* ______ 55
Figura 4.10 - Arquivos de saída da algoritmo de ordenação paralelo no *Reduce* ______ 56
Figura 4.11 - Comparação entre os algoritmos de ordenação no arquivo de 1 MB ______ 58
Figura 4.12 - Comparação entre os algoritmos de ordenação no arquivo de 100 MB ______ 59
Figura 4.13 - Comparação entre os algoritmos de ordenação no arquivo de 1 GB ______ 59

LISTA DE TABELAS

Tabela 4.1 - Detalhes dos resultados do *Insertion Sort* sequencial ____________________ 41

Tabela 4.2 - Configuração do *Hadoop* no *Insertion Sort* MapReduce__________________ 42

Tabela 4.3 - Detalhes dos resultados do *Insertion Sort* MapReduce ___________________ 43

Tabela 4.4 - Comparação entre a versão sequencial e paralela do *Insertion Sort* _________ 44

Tabela 4.5 - Detalhes dos resultados do *Merge Sort* sequencial ______________________ 46

Tabela 4.6 - Configuração do *Hadoop* no *Merge Sort* MapReduce____________________ 46

Tabela 4.7 - Detalhes dos resultados do *Merge Sort* MapReduce _____________________ 47

Tabela 4.8 – Comparação entra a versão sequencial e paralela do *Merge Sort* ___________ 47

Tabela 4.9 - Comparação entre *Merge Sort* sequencial e MapReduce sem o *Reduce* ______ 49

Tabela 4.10- Resultados do *Insertion Sort* MapReduce com 23 *Reduces* _______________ 50

Tabela 4.11 - Resultados do *Merge Sort* MapReduce com 23 *Reduces*_________________ 50

Tabela 4.12 - Configuração do *Hadoop* no *Insertion Sort* MapReduce_________________ 51

Tabela 4.13 - Configuração do *Hadoop* no *Merge Sort* MapReduce___________________ 51

Tabela 4.14 - Tempos de análise de desempenho geral _____________________________ 58

LISTA DE ABREVIATURAS E SIGLAS

CDH	Cloudera Hadoop
CPU	Central Processing Unit
DaaS	Database as a Service
GB	Gibabytes
HDFS	Hadoop Distributed File System
IaaS	Infrastructure as a Service
JAR	Java Archive
MB	Megabytes
RAM	Random Acess Memory
TB	Terabytes
YARN	Yet Another Resource Negotiator

SUMÁRIO

INTRODUÇÃO **11**

1 MAPREDUCE **14**

1.1 Big Data 14

1.2 O modelo de programação MapReduce 16

1.3 Apache Hadoop 18

1.3.1 HDFS 18

1.3.2 MapReduce dentro do Apache Hadoop 19

1.3.3 Versões e diferenças 19

1.3.4 Cloudera 20

1.4 Considerações finais do capítulo 21

2 ORDENAÇÃO **22**

2.1 Algoritmos de ordenação 22

2.1.1 Bubble Sort 22

2.1.2 Selection Sort 23

2.1.3 Insertion Sort 24

2.1.4 Merge sort 25

2.1.5 Quick Sort 25

2.1.6 Shell Sort 26

2.2 Ordenação paralela 27

2.3 Ordenação com MapReduce 28

2.4 Considerações finais do capítulo 29

3 EXPERIMENTOS **30**

3.1 Laboratório Experimental 30

3.2 Testes realizados 31

3.3 Implementação dos Algoritmos 32

3.3.1 Algoritmo *Merge Sort* Sequencial 33

3.3.2 Algoritmo *Insertion Sort* Sequencial 34

3.3.3 Algoritmos de Ordenação Paralelos Aplicados no Contexto do *Map* 35

3.3.4 Algoritmos de Ordenação Paralelos Aplicados no Contexto do *Reduce* 36

3.4 Considerações Finais do Capítulo 37

4 RESULTADOS **39**
4.1 Experimento com o algoritmo *Insertion Sort* sequencial 40
4.2 Experimento com o algoritmo *Insertion Sort* paralelo 41
4.3 Comparação entre o *Insertion Sort* sequencial e paralelo 43
4.4 Experimento *Merge Sort* sequencial 45
4.5 Experimento *Merge Sort* paralelo 46
4.6 Comparação entre o *Merge Sort* sequencial e paralelo 47
4.7 MapReduce com ordenação no *Map* com mais de um *Reduce* 50
4.8 Ordenação com MapReduce com ordenação no *Reduce* 54
4.9 Considerações Finais do Capítulo 57
CONCLUSÃO **60**
REFERÊNCIAS BIBLIOGRÁFICAS **63**

INTRODUÇÃO

A informação se faz necessária para a sociedade atual. Não só no meio corporativo se produz massas de informações, mas também no meio pessoal, por exemplo, indivíduos em suas rotinas diárias através de seus dispositivos eletrônicos. “Estima-se que até 2020 serão em torno de 30 bilhões de dispositivos móveis conectados à internet” (VELLOSO, 2014). Com o aumento constante da produção de dados, torna-se cada vez mais desafiador obter determinadas informações com eficácia.

Desta forma, se têm uma quantidade muito grande de dados na qual nem sempre é possível manipular a informação pelos meios convencionais. Tal quantidade de dados leva ao conceito de *Big Data*. “Em suma, o termo *Big Data* aplica-se a informações que não podem ser processadas ou analisadas por meio de processos ou ferramentas tradicionais” (ZIKOPOULOS; EATON; DEROOS, 2012).

Big Data é a simples constatação prática que o imenso volume de dados gerados a cada dia excede a capacidade das tecnologias atuais de os tratarem adequadamente. [...] *Big Data* = volume + variedade + velocidade. Hoje adiciona-se mais dois “V”s: veracidade e valor. (TAURION, 2012)

Com a crescente demanda de ferramentas de manipulação de *Big Data*, algumas companhias já estão investindo em produtos para suprir esta necessidade, a exemplo de grandes empresas como IBM[1] e SAP[2] que possuem um portfólio de ferramentas desta linha. A Cloudera[3] investe em uma ferramenta baseada no *framework open source* Apache Hadoop, que possibilita uma interface administrativa para acesso de escalabilidade e flexibilidade para diferentes tipos de dados. (RUSSOM; IBM, 2011)

Apesar da evolução constante da tecnologia, manipular *Big Data* ainda é um desafio. Devido à complexidade da distribuição de dados e do tamanho, que gira em torno dos *petabytes* (NESELLO, P.; FACHINELLI, 2014). Apesar dos diferentes métodos que ajudam a indexar os dados e da evolução de hardware, até mesmo as grandes corporações enfrentam dificuldades para manipular *Big Data*. “A IBM investiu, nos últimos cinco anos, mais de 14 bilhões de dólares na compra de 24 companhias para reforçar as capacidades analíticas de suas tecnologias” (OLIVEIRA, 2012).

---

[1] http://www-01.ibm.com/software/data/bigdata
[2] http://go.sap.com/solution/big-data.html
[3] http://www.cloudera.com/content/cloudera/en/about/hadoop-and-big-data.html

Para ajudar na manipulação de *Big Data*, foi desenvolvido pela Google um modelo de programação denominado MapReduce, que inicialmente estava implementado para análise de pesquisas *web*. “MapReduce é um modelo de programação associado ao processamento e geração de grandes conjuntos de dados” (DEAN; GHEMAWAT, 2008). Atualmente grandes companhias como Facebook, Yahoo, Amazon, IBM, entre outras, vem utilizando este modelo como ferramenta para manipulação de dados em implementações sobre *Cloud Computing* (ANJOS, 2012).

Embora surgido na Google, foi desenvolvido pela Apache Software Foundation uma implementação de MapReduce chamada Hadoop. Trata-se de um software de código aberto, projetado para permitir o processamento de grandes volumes de dados em *clusters*. Trata-se de um *framework* que dá suporte MapReduce capaz de detectar e tratar falhas, juntamente com um módulo responsável por escalonar tarefas e recursos do *cluster* (APACHE, 2015).

Visando oferecer um demonstrativo de desempenho na manipulação de grandes conjuntos de dados, este trabalho propõe o desenvolvimento de algoritmos tradicionais de ordenação, para posterior comparação com os desempenhos dos mesmos algoritmos em um ambiente multiprocessado, usando o *framework* Apache Hadoop. Desta forma, será possível medir em quais situações, os algoritmos MapReduce são ou não mais eficientes que os tradicionais.

Algoritmos de ordenação, ainda são considerados um dos problemas fundamentais da computação. O objetivo da ordenação, é facilitar o acesso aos dados, organizando-os de forma sequencial, geralmente em ordem alfabética (OLIVEIRA; SOUZA, 2008). Alguns algoritmos de ordenação são propícios de serem paralelizados, é o que trata o trabalho Algoritmos Paralelos de Ordenação (OLIVEIRA, 2008), que visa melhorar a performance dos já existentes, que rodam sequencialmente.

Os experimentos realizados com os algoritmos de ordenação paralelo, utilizando o modelo de programação MapReduce, foram executados em um *cluster* Apache Hadoop. O algoritmo de ordenação *Insertion Sort,* de complexidade do exponencial, apresentou um ganho de desempenho em comparação a sua versão sequencial. Para o cenário abordado neste trabalho, o algoritmo de ordenação com complexidade logarítmica, *Merge Sort*, obteve o menor tempo de execução na versão sequencial.

Este trabalho está dividido em quatro capítulos, o primeiro aborda *Big Data*, trazendo conceitos através de estudos bibliográficos, sua aplicação e estudo de casos oriundos de trabalhos correlatos. Explica o modelo de programação MapReduce, trazendo exemplos de

algoritmos, detalhando seu funcionamento e também aborda o Apache Hadoop, importante *framework* ligado ao tema. O capítulo seguinte trata de algoritmos de ordenações, suas complexidades e funcionamento. O terceiro capítulo abordando os algoritmos que serão utilizados no decorrer deste trabalho e o ambiente que serão rodados. O último capítulo traz os resultados dos experimentos realizados, assim com uma avaliação de desempenho entre os algoritmos tratados neste trabalho.

1 MAPREDUCE

O MapReduce é um modelo de programação paralela, que permite manipular e gerenciar dados em um conjunto de dispositivos. Com o objetivo de auxiliar a processar grandes massas de informações, este modelo de programação trabalha para que seja possível rodar paralelamente, algoritmos de manipulação de dados, principalmente em ambiente de *clusters*. Consiste em um sistema de programação de fácil manipulação, proporcionando que programadores inexperientes implementem programação paralela. Este modelo provê tolerância a falhas, distribuição de dados e balanceamento de carga (WHITE, 2010).

1.1 Big Data

Segundo a IBM, estima-se que 15 *petabytes* de dados, estruturados ou não, são produzidos por dia. Esta quantidade de dados, deve-se principalmente ao aumento do uso de dispositivos móveis, e-mails, planilhas, textos, etc (OLIVEIRA, 2012). Entretanto, o conceito de *Big Data* não diz respeito somente ao armazenamento de grandes massas de dados, mas também de toda a estrutura de acesso a estes dados, contribuindo para a manipulação de informações que os meios tradicionais, como bancos relacionais, têm dificuldade de processar.

Sendo assim, é possível destacar dois pontos importantes nas tecnologias que envolvem *Big Data*. A parte analítica, tendo o *Hadoop* e o MapReduce como principais tecnologias, e a infraestrutura responsável pelo armazenamento das grandes massas de dados para o uso posterior (TAURION, 2012).

O conceito de *Big Data* ainda envolve em sua estrutura os chamados 5Vs, correspondentes a: volume, variedade, velocidade, veracidade e valor (TAURION, 2012). O volume, se refere às grandes massas de dados presentes neste conceito. Este volume, vem acompanhado de uma variedade de informações, que aumenta a complexidade e o cuidado na hora de manipular *Big Data*.

Desta forma, a velocidade é um diferencial muito importante, sem ela teria pouca diferença dos métodos tradicionais pouco eficientes neste caso. As grandes massas de dados, produzidas pela sociedade atual, ainda podem ser melhor exploradas, tendo em vista a veracidade e o valor destas informações. É possível fragmentar estas informações e dar um objetivo cujo mercado ou a sociedade possa aproveitar de uma melhor maneira.

Existe uma variedade muito grande de aplicações no contexto de *Big Data*: científicas, engenharias, medicina, biologia, financeira, redes de sensores e nos últimos tempos as redes sociais. “Somente o Twitter gera mais de 7 *terabytes* (TB) de dados todos os dias, o Facebook 10 TB, e algumas empresas geram *terabytes* de dados a cada hora todos os dias do ano.” (ZIKOPOULOS; EATON; DEROOS, 2012)

Neste contexto, a análise de dados trabalha, usualmente, na casa dos *terabytes*, envolvendo dados soltos ou puramente conceituais, sendo estes de baixo valor, que são processados e transformados em dados mais representativos, de maior valor, o que envolve um processo de limpeza, amostragem e relacionamento continuo de dados. A grande quantidade dados, envolvida em *Big Data*, não agrega valor significativo em termos de qualidade, dentro do contexto de uma análise, pois a massa de informações por si só, não é autoexplicativa. Na análise de dados científicos, sendo da área de genoma, física, ambientais ou simulações numéricas, uma margem de imprecisão é tolerada. Entretanto, mesmo não exigindo precisão, a análise destas áreas têm a necessidade de ser processada de forma rápida ou em tempo real, o que é muito difícil nos métodos tradicionais, como em bancos relacionados (VIEIRA et al, 2012).

Recentemente, *Big Data* está presente em ambientes de computação na nuvem (*cloud computing*) e principalmente usado para o gerenciamento de dados. O foco principal está em duas tecnologias, DaaS e IaaS, respectivamente chamados de Bases de dados como Serviço (*Database* as a *Service*) e Infraestrutura como Serviço (*Infrastructure* as a *Service*) (ZIKOPOULOS; EATON; DEROOS, 2012). Grandes empresas como a Oracle estão investindo nesta linha.

Encontram-se aplicações de *Big Data* também no setor de energia. Por exemplo, em uma plataforma de petróleo, existem em torno de 20 a 40 mil sensores enviando dados sobre a “saúde” da plataforma, qualidade de operações, dentre outros. Estima-se que 90% das informações recebidas dos sensores são ruídos, este é um típico cenário onde a informação deve ser lida em tempo real, podendo separar o que realmente é informação daquilo que é ruído (ZIKOPOULOS; EATON; DEROOS, 2012).

Outro caso de estudo, é da empresa dinamarquesa chamada Vestas, líder global em energia eólica. Seu negócio gira em torno de venda, instalação e manutenção de turbinas eólicas, o que requer uma grande precisão para a instalação, uma falha pode gerar um prejuízo milionário. A determinação para o melhor posicionamento de uma turbina depende de um

grande número de fatores climáticos que devem ser considerados, tais como temperatura, precipitação, velocidade do vento, umidade, pressão atmosférica, entre muitos outros.

O sistema de modelagem da Vestas esperava inicialmente gerar 2600 TB de dados, e quando a engenharia das turbinas eólicas começou a desenvolver suas próprias previsões e gravações de dados reais para cada instalação, as suas necessidades foram projetadas a aumentar para 6000 TB (ZIKOPOULOS; EATON; DEROOS, 2012). O modelo de dados legado da empresa já não comportava manipular esta massa de dados, um cenário típico de *Big Data*.

Desta forma, a solução encontrada para manipular esta grande quantidade de informações gerada pelas turbinas, foi através de uma ferramenta analítica empresarial da IBM para plataforma *Big Data* baseada em *Hadoop*, que engloba componentes de código, chamada *BigInsights InfoSphere*, que roda em servidores IBM System X. Após a implementação desta ferramenta, a empresa Vestas se tornou capaz de gerenciar e analisar dados meteorológicos que antes não eram possíveis.

1.2 O modelo de programação MapReduce

Com o objetivo de auxiliar a indexação para a análise de pesquisas na *web*, o MapReduce foi inicialmente proposto na Google em 2004 por Jeffrey Dean e Sanjay Ghemawat, na época funcionários da empresa. A Google segue usando a metodologia MapReduce devido aos ganhos de performance em buscas e indexações e atualmente é usado para diferentes propósitos (DEAN, GHEMAWAT, 2008).

A principal característica do MapReduce é a capacidade de separar os dados e distribuí-los juntamente ao algoritmo que será executado entre os diversos nós do *cluster*. Com origens da programação funcional, o MapReduce foi denominado pela junção de duas funções presentes na linguagem Lisp, o *Map* e o *Reduce* (FILHO, 2011). Estas duas funções, permitem que os dados sejam separados e processados de forma distintas pelos nodos e depois unidos novamente, para gerar um retorno único.

Sendo assim, a função *Map* tem o objetivo de produzir, a partir dos dados de entrada, uma lista intermediária de pares chave e valor, também chamada de blocos. O *Reduce* utiliza estes blocos, e associa os valores intermediários por uma mesma chave, no qual combina esses valores para retornar um conjunto de resultados oriundos de todos os nós executados pelo *Map* (DEAN, GHEMAWAT, 2008). Internamente, entre estas duas funções, é realizado

um processo nos valores intermediários, onde são ordenados e agrupados por nós do *cluster*, esta etapa é chamada de *Shuffle*.

Para explicar melhor o funcionamento das funções *Map* e *Reduce* na prática, este trabalho utilizou o exemplo encontrado no artigo MapReduce*: Simplified Data Processing on Large Clusters* (DEAN, GHEMAWAT, 2008). Este exemplo, tem como finalidade contar as ocorrências de palavras repetidas encontradas em um arquivo de entrada fornecido.

Neste caso, a função *Map* tem por objetivo, indexar as palavras, formando os pares de chave-valor, onde a chave é cada palavra encontrada e ao valor é atribuído o número 1. A etapa *shuffle* agrupa e ordena as chaves iguais. A função *Reduce* deve, por sua vez, somar os valores atribuídos nas chaves iguais, preparadas pela etapa anterior, o que resultaria como saída do número de ocorrências de cada palavra.

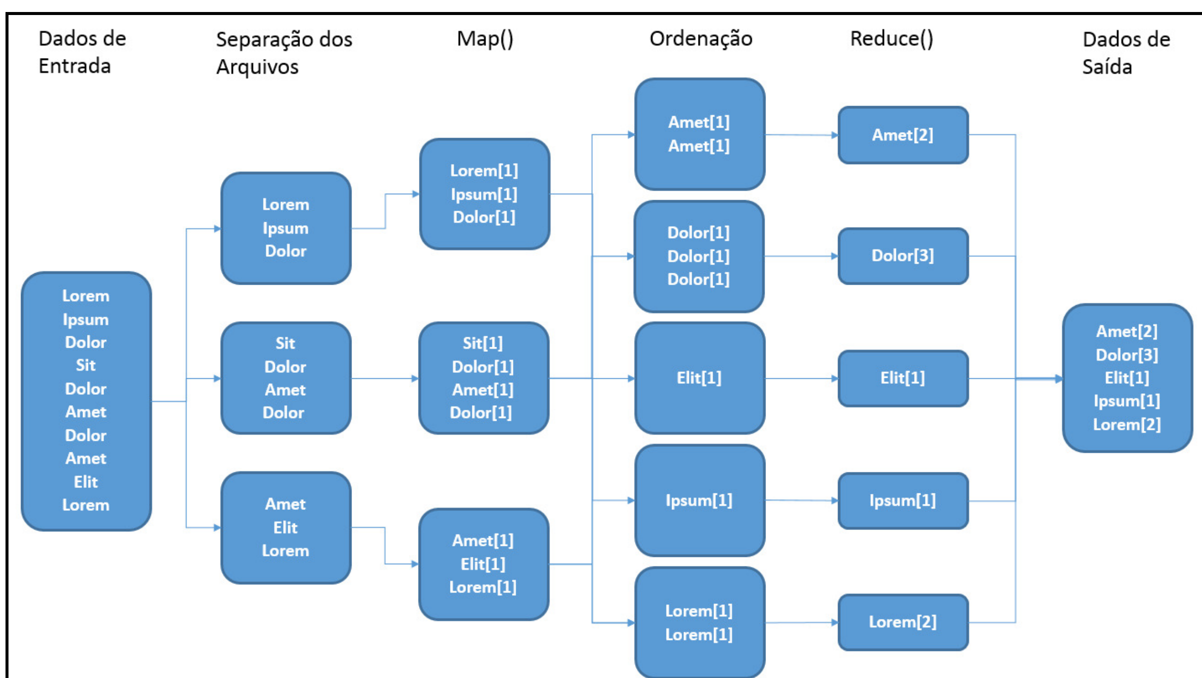

Figura 1.1 - Modelo do algoritmo do contador de palavras

Fonte: do Autor

A figura 1.1, ilustra o exemplo do algoritmo do contador de palavras de forma mais detalhada, adicionando as etapas que o MapReduce executa internamente. Neste caso, os dados de entrada, ou arquivo de entrada, seriam as palavras que devem ser contadas. A etapa de separação dos arquivos, é executada internamente pelo MapReduce, separando em blocos de forma que rode nos nodos do *cluster*, podendo sofrer interferência do programador ou não.

Neste contexto, a função *Map*, destacada na figura 1.1, forma os pares de chave-valor, onde a chave é cada palavra encontrada e o valor é o número 1. Após este processo, a etapa de *shuffle* ordena e agrupa as chaves geradas, o que acaba facilitando a execução da próxima etapa de *Reduce*. Esta etapa, irá associar as chaves iguais e somar os valores correspondentes a ela. Por fim, o resultado trazido no arquivo de saída, retorna as palavras do arquivo de entrada de forma distinta, com a quantidade de vezes que aquela palavra foi encontrada.

1.3 Apache Hadoop

O Apache Hadoop é um framework, que usa o modelo de programação MapReduce, que permite o processamento em larga escala de conjuntos de dados em *clusters* de computadores. Ele é projetado para escalonar, a partir de um único servidor ou para milhares de máquinas, e cada nó oferece processamento e armazenamento local. A biblioteca está preparada para detectar e tratar falhas na camada de aplicação, ao invés de confiar esta tarefa ao hardware.

O projeto inclui, além do modelo de programação MapReduce, outros módulos que auxiliam o funcionamento do *framework*: O *Hadoop Common*, é um módulo que dá suporte a outros módulos do *Hadoop*; O *Hadoop Distributed File System* (HDFS) é um sistema de arquivos distribuído que fornece acesso de alto rendimento para os dados do aplicativo; O módulo YARN é um *framework* para escalonamento de tarefas e gerenciamento de recursos de *cluster* de computadores (APACHE, 2015).

1.3.1 HDFS

O *Hadoop Distributed File System* (HDFS) é um dos fatores que contribuem para que seja possível escalonar tarefas em um *cluster Hadoop,* para centenas ou milhares de nós. Os dados em um *cluster Hadoop* são divididos em pedaços menores, chamados de blocos e distribuídos por todo o *cluster*. Desta forma, as funções *Map* e *Reduce* podem ser executadas em subconjuntos menores e isto proporciona a escalabilidade que é necessária para o processamento de grande volume de dados.

O objetivo do *Hadoop* é usar servidores comumente disponíveis em um grande *cluster*, em que cada servidor tenha um conjunto de unidades de disco internas de baixo custo. Para um melhor desempenho, o MapReduce tenta atribuir cargas de trabalho para esses

servidores, onde os dados a serem processados são armazenados. Isto é conhecido como dados de localização (COSS, 2012).

A responsabilidade do HDFS se encontra em dividir os dados em blocos e fazer o balanceamento destes no *cluster*. O HDFS deve garantir que os dados estejam íntegros nos blocos, por isso foi construído com recursos de compensação de tolerância a falhas. Para conseguir a disponibilidade dos dados quando um dos componentes falhar, o módulo replica esses blocos, por padrão em dois servidores adicionais, esta redundância oferece vários benefícios, o principal é garantir a integridade dos dados. Também é sua responsabilidade, manter o acesso a estes blocos, de forma eficiente garantindo a execução do MapReduce em grandes massas de dados (APACHE, 2015).

1.3.2 MapReduce dentro do Apache Hadoop

O MapReduce, além de ser um modelo de programação capaz de realizar o processamento paralelo, tem uma função muito importante dentro do Apache Hadoop. Mesmo não havendo falhas causadas pelo *hardware*, a maioria das tarefas de análise precisam ser capazes de combinar dados em certo momento, de algum modo. Os dados lidos a partir de um disco podem precisar ser combinados com os dados de qualquer outro disco do *cluster*. Vários sistemas distribuídos permitem que os dados sejam combinados, porém fazer este tipo de combinação corretamente é notoriamente difícil e de certa forma custoso ao gerenciador de arquivos (WHITE, 2010).

Sendo assim, a principal função de MapReduce, dentro do contexto de gerenciamento de arquivos do *Hadoop*, está ligado ao fato de que este modelo de programação lê e escreve no disco, formando um conjunto de dados com chaves e valores, tendo assim um índice, onde é possível identificar as partes que devem ser combinadas. O processamento de combinar os arquivos que foram processados por diferentes nós do *cluster*, fica a cargo do modelo de programação e não do sistema operacional, melhorando o desempenho do gerenciador de arquivos (WHITE, 2010). Em suma, o *Hadoop* fornece um sistema de armazenamento, fornecido por HDFS, e índice de mapeamento de dados, fornecido pelo MapReduce.

1.3.3 Versões e diferenças

Existe uma série de desafios que o Apache Hadoop deve enfrentar ao realizar o processamento de dados em paralelo, como uma máquina ou um processo parando de

responder, falhas de rede, entre outros. Para que o algoritmo MapReduce seja executado, a tarefa de entrada tem de ser dividida em pequenas tarefas e atribuída a várias máquinas no *cluster*. A saída parcial destas máquinas tem que ser consolidada, de alguma forma, para chegar ao resultado desejado.

Sendo assim, este contexto é contemplado por todas as versões do Apache Hadoop, com algumas diferenças, principalmente na parte de arquitetura, entre as suas versões 1 e 2. A versão 1, possui basicamente um processo no nodo *master* chamado *JobTracker*, que é responsável por gerenciar os recursos do *cluster* e distribuir tarefas. O outro processo, que ocorre nos nodos *slaves,* é chamado de *TaskTracker* e é encarregado de executar somente as tarefas *Map* e *Reduce* (APACHE, 2015).

O Apache Hadoop versão 2, introduziu um conceito de recipiente (*container*), que é genérico e roda em qualquer tipo de tarefa, fazendo com que os processos *master* e *slave* se distribuam de forma que os mesmos nodos possam assumir ambos os papeis. Este novo *framework*, também é responsável por gerenciamento dos recursos do *cluster*, e recebe o nome de YARN (*Yet Another Resource Negotiator*).

O YARN é a principal diferença entra as duas versões do Apache Hadoop. Devido principalmente a este *framework*, a versão 2 é superior a versão 1. Além de escalonar melhor as tarefas, a versão 2 ainda traz melhorias no sistema de arquivos HDFS. Os algoritmos MapReduce que rodavam na primeira versão são compatíveis com esta, entretanto devem ser compilados adequadamente com os conjuntos de arquivos JAR (APACHE, 2015).

1.3.4 Cloudera

A Cloudera é uma empresa que trabalha com uma grande variedade de ferramentas de *Big Data*, onde utiliza em todas as suas soluções o Apache Hadoop (CLOUDERA, 2015). Para dar credibilidade ao *framework*, o site da empresa detalha o funcionamento da ferramenta, trazendo conceitos de MapReduce, HDFS e YARN. As ferramentas fornecidas pela empresa são *open source*, porém nem todas são *freeware*.

A empresa apresenta algumas soluções, como no caso da máquina virtual com a instalação do CDH (Cloudera Hadoop) versão 5.4.0, que é uma distribuição de software de código aberto que consiste em Apache Hadoop e importantes projetos de código aberto.

Ligado ao CDH, a Cloudera disponibiliza a ferramenta *Cloudera Manager,* que possui uma versão *express* e uma *enterprise*. Esta ferramenta permite implantar, gerenciar, monitorar e realizar diagnósticos no *cluster* CDH. A versão corporativa possui um período de

teste, e oferece recursos avançados de gestão e suporte, incluindo atualizações, backup e recuperação.

1.4 Considerações finais do capítulo

O MapReduce, desde seu surgimento na Google, vem sendo utilizado por empresas como IBM e SAP, que possuem um portfólio de ferramentas no segmento de *Big Data*. A Cloudera, apresenta diversas máquinas virtuais com versões do Apache Hadoop, hibridas com ferramentas próprias, instaladas e prontas para aplicação em *cluster*, focado também em atender soluções de *Big Data*. Pode-se concluir, que o modelo de programação MapReduce vem sendo amplamente utilizado quando o assunto é *Big Data*.

Um dos motivos da aderência do MapReduce para se trabalhar com *Big Data*, é o fato de que aplicações desta área tendem a ser muito custosas ao *hardware*. Sendo que, uma das soluções para mitigar o custo de se trabalhar com *Big Data*, é explorar a programação paralela. O modelo de programação tratado neste capítulo é um ótimo meio para se trabalhar desta forma. Por ser um modelo de fácil entendimento, programadores inexperientes podem desenvolver aplicações em *cluster*.

O algoritmo detalhado neste capítulo, tem o propósito de contar o número de palavras em uma grande quantidade de dados, de forma paralela, tendo em vista ser executado em um *cluster* de computadores. Desta mesma forma, é possível que outros algoritmos possam ser implementados para explorar a programação paralela e obter ganhos de performance com a utilização do MapReduce. O capítulo seguinte, irá tratar sobre algoritmos de ordenação já conhecidos da literatura da computação e implementados de forma linear. Entretanto, abordará de quais formas poderia haver melhoria de desempenho com a aplicação do paralelismo nestes algoritmos.

2 ORDENAÇÃO

O processo de ordenação é um dos mais executados na área da computação. Isto se deve ao fato de que na maior parte dos casos, manipular a informação de forma ordenada é mais fácil e menos custosa. “Ordenar corresponde ao processo de rearranjar um conjunto de objetos em uma ordem ascendente ou descendente. O objetivo principal da ordenação é facilitar a recuperação posterior de itens do conjunto ordenado” (ZIVIANI, 2004).

Pode-se dizer que algoritmos de ordenação ainda são um dos problemas fundamentais da área de computação (ANJOS, 2012). Devido a isto, existem vários tipos de algoritmos, com diversos níveis de complexidade, que visam melhor desempenho em determinadas situações. Estes algoritmos podem ser divididos em duas categorias: algoritmos baseados em comparações e algoritmos não baseados em comparações (GONDA, 2004).

Algoritmos não baseados em comparações, como por exemplo o *counting sort*, utilizam-se de outros elementos a partir do conteúdo a ser ordenado, porém não compara conteúdo a conteúdo da lista a ser ordenada. Este algoritmo, compara os elementos da lista e cria uma lista auxiliar, no qual os elementos da lista original correspondem aos índices da lista auxiliar. Neste caso, como os índices já estão ordenados, é possível gerar uma nova lista com os elementos da lista auxiliar, existentes na lista original e resultar em uma lista ordenada.

2.1 Algoritmos de ordenação

Existem diversos algoritmos para ordenação, suas complexidades variam em $O(n^2)$ e $O(n \log n)$. Alguns exemplos de algoritmos com complexidades $O(n^2)$: *Bubble Sort, Quick Sort e Selection Sort*. Com complexidade $O(n \log n)$ existem: *Heap Sort, Merge Sort* e *Quick Sort*, entre outros. Os algoritmos citados têm suas complexidades iguais nos seus melhores e piores casos, exceto o *Quick Sort*, que possui no melhor caso $O(n \log n)$ e no pior $O(n^2)$ (BARBOSA, TOSCANI, RIBEIRO, 2000)

2.1.1 Bubble Sort

O *Bubble Sort* é um algoritmo de ordenação, que funciona percorrendo a lista que deve ser ordenada, por diversas vezes, comparando valor a valor, de forma sequencial, até que toda a lista esteja totalmente ordenada (ASTRACHAN, 2003). No melhor dos casos, o *Bubble*

Sort tem complexidade O(n), onde n é o número de elementos da lista, mas na maioria dos casos, incluindo o pior caso, ele tem complexidade O(n²).

```
void BubbleSort(Vector a, int n)
{
    for(int j=n-1; j > 0; j--)
        for(int k=0; k < j; k++)
            if (a[k+1] < a[k])
                Swap(a,k,k+1);
}
```

Figura 2.1 - Algoritmo de ordenação *Bubble Sort*

Fonte: ASTRACHAN, 2003

Dificilmente alguma lista a ser ordenada levará o tempo de uma complexidade O(n). Aplicações que buscam uma boa performance não costumam utilizar este algoritmo, pois ele tem complexidade quadrática O(n²).

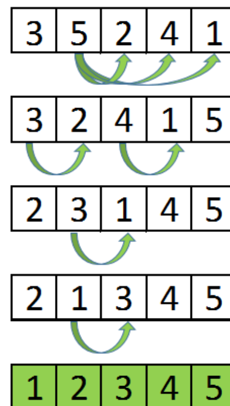

Figura 2.2 - Exemplo de ordenação com *Bubble Sort*

Fonte: do autor

2.1.2 Selection Sort

O algoritmo de ordenação por seleção, ou *Selection Sort* funciona através de um método de substituição por seleção. Ou seja, percorre-se a lista de elementos a ser ordenada, identifica-se o menor valor da lista, ou maior, dependendo do tipo de sequência da ordenação desejada (crescente ou decrescente) e realiza-se a substituição do valor da posição do primeiro (ou último) elemento da lista, pelo valor da posição do elemento identificado (LOPES, 2015).

Sendo assim, este procedimento deve ser realizado para cada posição da lista recebida, fazendo com que este algoritmo de ordenação tenha complexidade de $O(n^2)$. Outros algoritmos de ordenação possuem complexidade melhor $O(n \log n)$, fazendo deste um algoritmo de ordenação pouco utilizado.

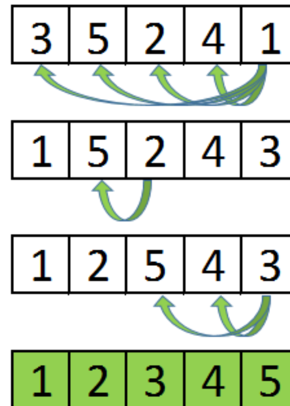

Figura 2.3 - Exemplo de ordenação com *Selection Sort*

Fonte: Do autor

2.1.3 *Insertion Sort*

O funcionamento do algoritmo de ordenação por inserção, *Insertion Sort*, é semelhante ao manuseio de um jogador de cartas organizando os números em sua mão. O algoritmo percorre os elementos a serem ordenados da esquerda para a direita, passando por cada elemento e inserindo este elemento à esquerda da lista, onde seu critério de ordenação é aceito. Ou seja, a cada elemento percorrido é verificado se algum elemento à esquerda dele é maior, se for, o elemento é deslocado para o lugar certo (SILVA, 2007).

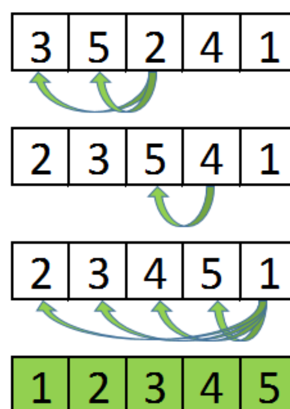

Figura 2.4 - Exemplo de ordenação com *Insertion Sort*

Fonte: do autor

Desta forma, à medida que a lista vai sendo percorrida, os elementos à esquerda vão ficando ordenados. Este é um algoritmo eficiente para pequenas quantidades de elementos, porém possui complexidade $O(n^2)$, não sendo um algoritmo de alto desempenho.

2.1.4 Merge sort

O funcionamento deste algoritmo utiliza a abordagem de "dividir para conquistar" a fim de ordenar os elementos. Seu funcionamento se dá de forma recursiva, com complexidade O(n log n) para todos os casos. Um ponto negativo deste algoritmo, tendo em vista grande massas de dados, é o alto consumo de memória.

Este algoritmo de ordenação se dá em dois momentos. O primeiro trata da etapa de divisão do vetor de entrada, de modo a reduzir estas entradas em dados menores e subsequentes. Partindo do princípio que esta primeira etapa retorna os dados previamente ordenados, a segunda etapa se encarrega de mesclar as entradas retornadas pela primeira parte. Existem na literatura diversas maneiras de ordenar e mesclar estas listas, inclusive de forma paralela (GARCIA, 2013).

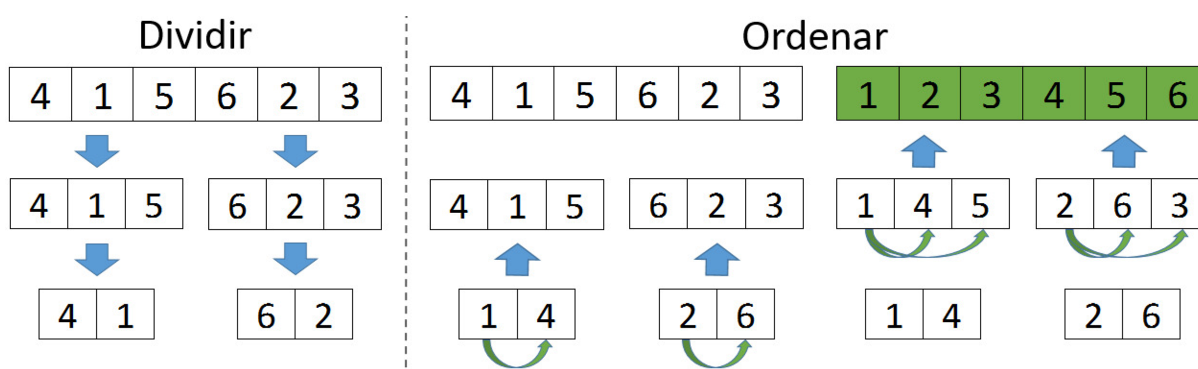

Figura 2.5 - Exemplo de ordenação com *Merge Sort*

Fonte: do autor

2.1.5 Quick Sort

O algoritmo de ordenação denominado *Quick Sort* foi inventado em 1960 por Charles Antony Richard Hoare e posteriormente, em 1962, foi publicado com uma série de refinamentos. É um algoritmo relativamente fácil de ser implementado e realiza a ordenação sem um vetor auxiliar. O tempo médio de execução é ótimo, com complexidade de O(n log n), mas sua complexidade no pior caso é $O(n^2)$, sendo um dos algoritmos de ordenação mais

utilizados nos dias de hoje. Ainda assim, sua implementação vem sofrendo tentativas de melhoria com diferentes versões otimizadas (PRADO, 2005).

O funcionamento deste método consiste em, primeiramente, determinar um valor pivô na lista. Ele recebe este nome, pois fará o direcionamento e ficará entre as duas listas geradas posteriormente. Em seguida, todos os itens da lista devem ser divididos de forma que valores menores que o pivô, fiquem antes dele e, consequentemente, os valores maiores fiquem depois dele. Sendo assim, o pivô já estará no lugar dele, e os demais deverão ser ordenados. Este procedimento deve ser aplicado de forma recursiva nas sub-listas geradas, até que toda a lista seja ordenada (OLIVEIRA, 2008).

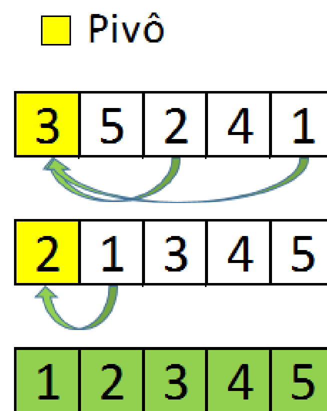

Figura 2.6 - Exemplo de ordenação com *Quick Sort*

Fonte: do autor

2.1.6 Shell Sort

O algoritmo de ordenação *Shell Sort*, possui complexidade quadrática O(n²). Foi criado em 1959 por Donald Shell, este algoritmo possui a melhor eficácia entre os algoritmos de ordenação com a mesma complexidade. Semelhante ao método de ordenação por inserção, *Insertion Sort*, este algoritmo divide a lista de entrada de elementos a serem ordenados em pequenos grupos e então é aplicado o mesmo algoritmo de ordenação utilizado no *Insertion Sort* para realizar a ordenação destes pequenos grupos (DA ROSA, 2014). O processo de ordenação passa pela lista várias vezes criando os subgrupos e aplicando a ordenação necessária quantas vezes for preciso até a lista estar totalmente ordenada.

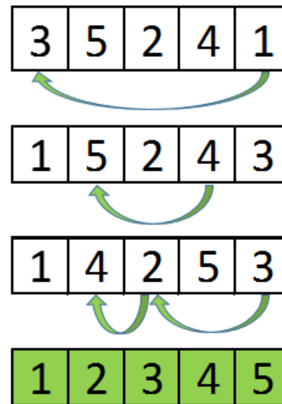

Figura 2.7 - Exemplo de ordenação com *Shell Sort*

Fonte: do autor

2.2 Ordenação paralela

Os algoritmos de ordenação sequenciais executam suas tarefas uma instrução de cada vez, não executando uma próxima instrução sem a anterior terminar. Entretanto, existem casos de algoritmos, onde estas instruções podem ser processadas paralelamente, levando em conta arquiteturas de computadores e o número de processadores para executar as instruções ao mesmo tempo. Estes algoritmos dividem os problemas em subproblemas, repassando os subproblemas para os processadores disponíveis e juntando os seus resultados em um único resultado final. Tal conceito é base para a programação paralela (OLIVEIRA, 2008).

O trabalho “Algoritmos Paralelos De Ordenação” (OLIVEIRA, 2008) tem o objetivo de estudar os algoritmos de ordenação paralelos, tendo em vista seu desempenho e consistência de dados. Foi realizado uma comparação entre suas versões sequenciais e paralelas dos algoritmos de ordenação *Bubble Sort, Merge Sort, Quick Sort, Rank Sort, Counting Sort* e *Radix Sort*. O resultado final deste trabalho correlato evidencia o desempenho superior dos algoritmos de ordenação paralela. Entretanto a ordenação paralela exige outros desafios como concorrência e consistência de dados.

Sendo assim, a maior dificuldade deste trabalho relacionado, foi manter os dados íntegros, tratando as falhas dos nodos e a concorrência, problemas que não ocorrem quando o algoritmo é executado sequencialmente. Os problemas descritos foram mitigados com auxílio de *frameworks* para o processamento em *cluster*. A proposta do MapReduce com Apache Hadoop tende a resolver o problema de integridade de dados.

2.3 Ordenação com MapReduce

Outra implementação de MapReduce é encontrada no artigo "Implementação e Avaliação de Algoritmos de Ordenação Paralela em MapReduce" (MURTA et al., 2013). Este trabalho se propôs a desenvolver dois algoritmos de ordenação utilizando o modelo de programação MapReduce em um ambiente Apache Hadoop e realizar uma comparação de desempenho. Foram apresentados o *Quick Sort* Paralelo e a Ordenação por Amostragem, sobre os quais foram realizados testes para medição de seus desempenhos.

O algoritmo de Ordenação por Amostragem, assim como o *Quick Sort,* é baseado na divisão da lista de informações que devem ser ordenadas. A divisão da lista, ocorre de acordo com a quantidade de processadores disponíveis, esta lista é dividida de forma que os subconjuntos estejam em faixas diferentes ordenação, ou seja, no caso de 2 subconjuntos, os itens do primeiro devem ser menores que os itens do segundo subconjunto. Sendo assim, a lista pode ser agrupada resultando em uma lista ordenada. A seleção dos subconjuntos, pode ser feita através de diferentes estratégias, tendo em vista o arquivo de dados de entrada.

O algoritmo *Quick Sort* Paralelo se mostrou superior ao de Ordenação por Amostragem ao ser utilizado em ordenações com menores quantidades de dados, sendo que ao definir as partições, é realizado um processamento extra. Entretanto, para quantidades maiores de dados, o *Quick Sort* Paralelo apresenta desempenho inferior, devido ao reduzido número de partições e uma divisão de trabalho menor entre os processadores.

A avaliação dos algoritmos propostos demonstrou que são estáveis, com tempos de execução similares e a distribuição dos dados não influencia no tempo de ordenação. Os resultados apontaram de maneira conclusiva o melhor desempenho e escalabilidade do algoritmo Ordenação por Amostragem, que apresentou os menores tempos de ordenação exibido na figura 2.8 e uma boa distribuição de trabalho entre as máquinas presentes no *cluster,* conforme figura 2.9. Estes gráficos mostram o tempo de ordenação de cada um dos algoritmos, tendo em vista o número de partições, que correspondem a quantidade de subconjuntos gerados pelo algoritmo, e em relação ao número de máquinas existentes no *cluster* em que os algoritmos foram análisados.

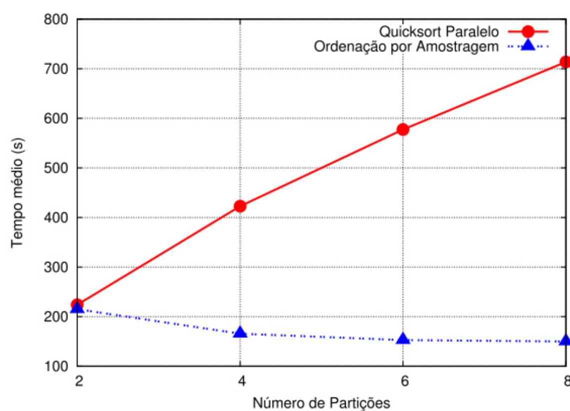

Figura 2.8 - Tempo para ordenação de 10^8 dados variando o número de partições

Fonte: MURTA et al., 2013

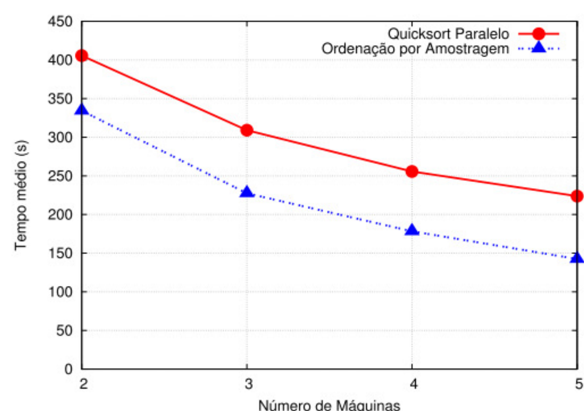

Figura 2.9 - Resultados para variação do número de máquinas de processamento

Fonte: MURTA et al., 2013

O artigo de MURTA et al. (2013) aqui evidenciado, não realizou uma análise de desempenho entre os algoritmos MapReduce com a sua tradicional implementação sequencial. Este trabalho de conclusão se propõe a realizar este estudo, trazendo resultados dos experimentos realizados, tendo assim, uma avaliação de desempenho entre os algoritmos sequenciais e suas implementações MapReduce.

2.4 Considerações finais do capítulo

Os algoritmos de ordenação quando executados sequencialmente são processados instrução a instrução, onde cada uma deve esperar a anterior terminar para que uma nova seja executada. Entretanto, existem algoritmos que podem ser processados em paralelo, usufruindo de computadores com vários núcleos, ou até mesmo em estrutura de *cluster*, para que o processamento seja feito ao mesmo tempo em diferentes instruções.

Sendo assim, algoritmos de ordenação paralela dividem suas tarefas em sub-tarefas e as distribuem pelos nós ou núcleos de processadores que estiverem disponíveis, juntando no final os resultados das tarefas executadas. Este conceito é a base para a programação paralela de maneira geral e também é a base do modelo de programação MapReduce.

O objetivo deste trabalho é, fazer uma comparação de desempenho entre algoritmos implementados de forma sequencial e usando o modelo de programação MapReduce. Serão utilizados um algoritmo de complexidade $O(n^2)$, *Insertion Sort,* e o *Merge Sort* de complexidade $O(n \log n)$. Os resultados das comparações de complexidades, poderão ajudar aplicações voltados para a área manipulação de grandes massas de dados.

3 EXPERIMENTOS

A etapa de realização de experimentos deste trabalho consiste em executar os algoritmos de ordenação *Insertion Sort* e *Merge Sort* de forma sequencial e no formato do modelo de programação MapReduce e realizar uma comparação de desempenho entre ambos. Os algoritmos serão executados no mesmo *hardware*, porém em ambientes computacionais diferentes. Para que o algoritmo MapReduce rode é necessário ter um ambiente Apache Hadoop configurado, diferentemente do algoritmo de ordenação sequencial que é necessário apenas a máquina virtual Java. Os próximos subcapítulos trarão mais detalhes sobre os algoritmos aqui descritos e do ambiente utilizado para execução dos mesmos, assim como os detalhes das etapas da análise de desempenho.

3.1 Laboratório Experimental

A comparação de desempenho entre os algoritmos de ordenação sequencial e MapReduce será realizado em um laboratório contendo um *cluster* composto por três computadores em rede. As três máquinas possuem processadores Intel Xeon CPU E5345 2.33GHz x 8. O servidor 1 possui 16 GB de memória RAM e 126 GB de espaço em disco, o servidor 2 possui 8 GB de memória RAM e 63 GB de espaço em disco e o servidor 3 possui 32 GB de memória RAM e 38 GB de espaço em disco.

Os algoritmos de ordenação *Marge Sort* e *Insertion Sort* sequenciais serão executados em um único servidor, neste caso o servidor 3, o mais robusto do *cluster*. Não há necessidade de utilizar mais de um nodo do *cluster* para executar estes algoritmos, pois eles não são projetados para executar em rede ou utilizando os recursos de um *cluster*, mas sim em um único computador, no momento de executar estes algoritmos os demais nodos do *cluster* permanecerão ociosos.

A execução dos algoritmos de ordenação paralelos, desenvolvidos no modelo de programação MapReduce, serão realizados no *cluster*, nos nodos descritos acima. O *cluster* possui um sistema operacional Linux Ubuntu 14.04.3 64 *bits* com o Apache Hadoop instalado e configurado para processar os algoritmos nos 3 nodos presentes. Os resultados gerados resultarão na comparação de desempenho proposta por este trabalho. A próxima seção irá tratar dos testes realizados a partir dos algoritmos.

3.2 Testes realizados

A comparação de desempenho entre os algoritmos sequenciais e paralelos se dá a partir de dois arquivos com conteúdo semelhantes e tamanhos diferentes. O arquivo maior possui 1 GB de dados e o menor 100 MB, ambos com codificação UTF-8. O conteúdo dos arquivos possui frases e diversas linhas, conforme o exemplo da figura 3.1. Os arquivos tratam-se de uma concatenação de mais de dois mil livros provenientes do Projeto Gutenberg (http://www.gutenberg.org/), que tem por objetivo disponibilizar livros (*eBooks*) livre para *downloads*. O propósito de ter dois arquivos com tamanhos diferentes é mapear se o tempo de execução possui variação de desempenho, para melhor ou pior, de acordo com o tamanho do arquivo processado.

```
Title: The Declaration of Independence
The United States Declaration of Independence was the first Etext
in an emailed instruction set which required a tape or diskpack be
hand mounted for retrieval.  The diskpack was the size of a large
cake in a cake carrier, cost $1500, and contained 5 megabytes, of
which this file took 1-2%.  Two tape backups were kept plus one on
paper tape.  The 10,000 files we hope to have online by the end of
2001 should take about 1-2% of a comparably priced drive in 2001.
This file was never copyrighted, Sharewared, etc., and is thus for
all to use and copy in any manner they choose.  Please feel free to
In my research for creating this transcription of our first Etext,
I have come across enough discrepancies [even within that official
documentation provided by the United States] to conclude that even
"facsimiles" of the Declaration of Indendence will NOT going to be
all the same as the original, nor of other "facsimiles."  There is
a plethora of variations in capitalization, punctuation, and, even
where names appear on the documents [which names I have left out].
The resulting document has several misspellings removed from those
parchment "facsimiles" I used back in 1971, and which I should not
[JT, Apr 05: "Brittish", however, is spelled as in the original.]
The Declaration of Independence of The United States of America
The unanimous Declaration of the thirteen united States of America
When in the Course of human events, it becomes necessary for
```

Figura 3.1 - Trecho do arquivo utilizado para ordenação

Fonte: do autor

Sendo assim, os algoritmos de ordenação *Merge Sort* e *Insertion Sort*, em seus formatos sequenciais e MapReduce processarão individualmente cada arquivo proposto, gerando os resultados de desempenho quanto ao tempo de execução de cada um.

A comparação dos resultados obtidos, ocorrerá de forma em que cada tipo de algoritmo de ordenação seja comparada entre a sua versão sequencial e paralelo, por arquivo processado. Serão apresentados gráficos para mostrar o desempenho individual dos algoritmos e o comparativo quanto aos tamanhos dos arquivos dos algoritmos sequenciais e MapReduce.

A avaliação de desempenho comtempla também etapas que existem no algoritmo como funções de comparação de palavras e métodos de extração de palavras de trechos de caracteres. Como todos os algoritmos possuem estes trechos de código, não haverá favorecimento para mais ou para menos, em nenhum dos algoritmos de ordenação. Serão realizados 3 execuções de cada algoritmo, para dar mais solidez ao teste.

3.3 Implementação dos Algoritmos

O desenvolvimento do algoritmo dos algoritmos de ordenação foi baseado em trabalhos correlatos. Para desenvolver o algoritmo de ordenação sequencial *Merge Sort* foi utilizado informações do trabalho “Analisando o Desempenho da Paralelização no Algoritmo de Ordenação Mergesort In-place” (GARCIA, 2013). O trabalho “Realimentação de Relevância via Algoritmos Genéticos aplicada a Recuperação de Imagens” (SILVA, 2007) serviu de base para auxiliar o desenvolvimento do algoritmo de ordenação *Insertion Sort* de forma sequencial.

Através de testes exploratórios e pesquisas realizadas, foram encontradas duas formas de inserir os algoritmos de ordenação no modelo de programação MapReduce para que estes rodem de forma paralela. Uma das formas foi inserir os algoritmos de ordenação na função *Map* e a outra dentro da função *Reduce*. Em nenhum dos casos, foi necessário modificar os algoritmos de ordenação. Sua diferença para o formato sequencial é o contexto em que foi inserido, ou na função *Map* ou na *Reduce*. Este trabalho, vai explorar as duas abordagens de execução de algoritmos paralelos.

O objetivo dos algoritmos, sendo paralelos ou sequenciais, é ordenar as palavras contidas nos arquivos, sendo que eles são compostos por palavras e frases contidas em diversas linhas. Sendo assim, é necessário separar cada palavra do contexto da frase, resultando ao final da execução uma lista de palavras ordenadas. Os algoritmos não preveem nenhum agrupamento de palavras. Todos os algoritmos de ordenação, sendo sequenciais ou paralelos, possuem o mesmo método de extração de palavras e o mesmo método de comparação de caracteres.

3.3.1 Algoritmo *Merge Sort* Sequencial

A construção do algoritmo de ordenação *Merge Sort,* proposto por este trabalho, tem o objetivo de ordenar uma lista de conjuntos de caracteres alfanuméricos de forma ascendente. Este algoritmo recebe por parâmetro um *array* de *strings* e utiliza-se de métodos recursivos para realizar a ordenação dos subconjuntos de listas de conjuntos de caracteres gerados pelo algoritmo. Este algoritmo está ilustrado na figura 3.2, onde existem três métodos que são utilizados de forma recursivamente, para realizar a ordenação.

```
public static String[] MergeSort(String[] vetor) {
    String[] vetorTemp = new String[vetor.length];
    mergeSorter(vetor, vetorTemp, 0, vetor.length - 1);
    return vetor;
}
private static void mergeSorter(String[] vetor, String[] vetorTemp,
        int left, int right) {
    if (left < right) {
        int center = (left + right) / 2;
        mergeSorter(vetor, vetorTemp, left, center);
        mergeSorter(vetor, vetorTemp, center + 1, right);
        merge(vetor, vetorTemp, left, center + 1, right);
    }
}
private static void merge(String[] vetor, String[] vetorTemp, int left,
        int right, int rightEnd) {
    int leftEnd = right - 1;
    int k = left;
    int num = rightEnd - left + 1;

    while (left <= leftEnd && right <= rightEnd)
        if (Helper.GreaterThan(vetor[right], vetor[left]))
            vetorTemp[k++] = vetor[left++];
        else
            vetorTemp[k++] = vetor[right++];

    while (left <= leftEnd)
        vetorTemp[k++] = vetor[left++];

    while (right <= rightEnd)
        vetorTemp[k++] = vetor[right++];

    for (int i = 0; i < num; i++, rightEnd--)
        vetor[rightEnd] = vetorTemp[rightEnd];
}
```

Figura 3.2 - Algoritmo sequencial *Merge Sort*

Fonte: do autor

No algoritmo de figura 3.2, existe um método público chamado *MergeSort*, que se encarrega de chamar a função recursiva *mergeSorter*. Através desta função recursiva é possível dividir a lista de entrada, fazer a comparação e obter uma lista ordenada. O método *MergeSort* estará presente conforme apresentado nesta mesma figura em todas as vezes que for necessário chamar este algoritmo de ordenação. O método *merge* é responsável por realizar a ordenação propriamente dita, ela que realizará a comparação entre os itens e realizará a troca de posições quando necessário.

3.3.2 Algoritmo *Insertion Sort* Sequencial

A implementação do algoritmo de ordenação *Insertion Sort,* proposta por este trabalho, tem em vista ordenar uma lista de conjuntos de caracteres alfanuméricos de forma ascendente. É recebido por parâmetro um *array* de *strings* e passado para um laço (*loop*) onde todo o *array* é percorrido. É usada uma variável do tipo *string* para armazenar o valor eleito para a comparação entre os itens da lista. Ao término da execução, o *array* inicial estará ordenado.

Conforme mostra a figura 3.3, existe um método público chamado *InsertionSort*, que que contém algoritmo de ordenação *Insertion Sort*. Este método estará presente, conforme o apresentado nesta mesma figura, em todas as vezes que for necessário chamar este algoritmo de ordenação.

```
public static String[] InsertionSort(String[] lista) {

    int i, j;
    String eleito;
    for (i = 1; i < lista.length; i++) {
        eleito = lista[i];
        j = i;
        while ((j > 0) && (Helper.GreaterThan(lista[j - 1], eleito))) {
            lista[j] = lista[j - 1];
            j = j - 1;
        }
        lista[j] = eleito;
    }

    return lista;
}
```

Figura 3.3 - Algoritmo sequencial *Insertion Sort*

Fonte: do autor

3.3.3 Algoritmos de Ordenação Paralelos Aplicados no Contexto do *Map*

A função *Map* do MapReduce funciona lendo o arquivo linha a linha. Desta forma, para utilizar a abordagem do algoritmo de ordenação, inserida na função *Map,* é necessário extrair as palavras do trecho lido e aplicar o algoritmo de ordenação neste momento, obtendo uma ordenação parcial do arquivo. A função *Map* é executada nos nodos do *cluster*, gerando assim a ordenação paralela, que irá devolver pilhas de chaves e valores ordenados.

Na figura 3.4, pode-se notar o método utilizado para extrair as palavras do trecho enviado ao *Map* por parâmetro, linha 3. Logo abaixo, linha 5, é utilizado o algoritmo de ordenação desejado, podendo ser o *Merge Sort* ou *Insertion Sort*. A partir da linha 9 desta mesma figura, começa a preparação para serem geradas as chaves e valores, neste caso, as chaves e valores são os mesmos, sendo a chave gerada com letras maiúsculas, linha 14, para que o *shuffle* ordene corretamente.

```
1  public void map(LongWritable key, Text value, Context context) {
2
3      String[] palavras = Helper.QuebrarLinha(value.toString());
4
5      String[] list = Ordenar.MergeSort(palavras);
6
7      Text result = new Text();
8
9      for (String val : list) {
10         Text textoValue = new Text();
11         textoValue.set(val);
12
13         Text textoKey = new Text();
14         textoKey.set(val.toUpperCase());
15
16         context.write(textoKey, textoValue);
17     }
18 }
```

Figura 3.4 - Função Map() no contexto de ordenação no *Map*

Fonte: do autor

Sendo assim, com as palavras ordenadas das frases do arquivo lido, o *shuffle* se encarrega de pegar todos os pedaços, previamente ordenados, gerados pelas funções *Maps* de cada nodo do *cluster,* e ordená-los totalmente, contemplando todos os pedaços, o que é o comportamento padrão do *shufle e* não é necessária qualquer interferência. A função *Reduce* tem a única responsabilidade de reunir os valores da lista para o retorno de uma lista única.

```java
public void reduce(Text key, Iterable<Text> values, Context context) {

    for (Text val : values) {
        context.write(new Text(), val);
    }
}
```

Figura 3.5 - Função Reduce() no contexto de ordenação no *Map*

Fonte: do autor

3.3.4 Algoritmos de Ordenação Paralelos Aplicados no Contexto do *Reduce*

O algoritmo de ordenação dentro do contexto da função *Reduce* tem uma função um pouco diferente do que no contexto do *Map*. Como a função *Map* precisa gerar chaves e valores para que a função *Reduce* possa ordenar, é necessário que as listas geradas fiquem particionadas. A maneira de mapear as chaves proposta por este trabalho, foi pegar a primeira letra de cada palavra e usar como chave, e no valor o restante da palavra. Sendo assim, a função do *shuffle* ficaria encarregada de ordenar a primeira letra de cada palavra e passar para o *Reduce* todas as palavras que comecem com uma determinada letra, que por sua vez se encarregará de ordenar a partir da segunda letra de cada palavra.

```java
public void map(LongWritable key, Text value, Context context){

    String[] palavras = Helper.QuebrarLinha(value.toString());

    int tamanhoChave = 1;

    for (int x = 0; x < palavras.length; x++) {

        String palavra = palavras[x].trim();

        if(palavra.length() == 0) continue;

        Text textoValue = new Text();
        textoValue.set(palavra.substring(tamanhoChave));

        Text textoKey = new Text();
        textoKey.set(palavra.substring(0, tamanhoChave).toUpperCase());

        context.write(textoKey, textoValue);
    }

}
```

Figura 3.6 - Função *Map*() no contexto de ordenação no *Reduce*

Fonte: do autor

A função *Map*, conforme figura 3.6, extrai as palavras da frase, linha 3, conforme a função *Map* da ordenação no contexto do *Map*. Entretanto, ao invés de chamar a função de ordenação como no contexto anterior, são geradas chaves a partir da primeira letra de cada palavra extraída da linha, conforme linha 17 da figura 3.5, sendo o valor o restante da palavra, linha 14 da mesma figura.

Neste contexto, a função do *shuffle* será a de agrupar as chaves iguais geradas pelo *Map* e repassar ao *Reduce*, por exemplo, agrupar todas as chaves "A" e todas as palavras que começa com "A", o *shuffle* neste momento não precisa identificar todas as palavras que começar com "A", o *Map* já atrelou todas as palavras que começam com "A" a chave "A". Este é o comportamento padrão do *shuffle* e não é necessária intervenção.

A função do *Reduce* neste caso, além de juntas os conjuntos de chave e valor gerados no *Map*, é de realizar a ordenação propriamente dita, como mostra a figura 3.7, linha 3. Nela poderiam ser inseridos os algoritmos de ordenação *Marge Sort* ou *Insertion Sort*, posteriormente, a partir da linha 7 da mesma figura, cabe ao ele também concatenar a chave, que representa a primeira letra de cada palavra ordenada, com cada palavra resultante da lista ordenada, tendo assim um arquivo de retorno.

```
1  public void reduce(Text key, Iterable<Text> values, Context context) {
2
3      String[] lista = Ordenar.InsertionSort(values);
4
5      Text result = new Text();
6
7      for (String val : lista) {
8          result.set(key.toString() + val);
9          context.write(new Text(), result);
10     }
11
12 }
```

Figura 3.7 - Função *Reduce*() no contexto de ordenação no *Reduce*

Fonte: do autor

3.4 Considerações Finais do Capítulo

O laboratório fornecido pela Universidade Feevale, na sala 204 do Prédio Verde, proporciona ao Apache Hadoop um *cluster* com três nodos, onde é possível realizar testes utilizando o processamento paralelo. Da mesma forma, também proporciona o hardware apropriado para os testes dos algoritmos sequenciais.

A proposta de manter o mesmo algoritmo de ordenação para as formas sequenciais e paralela garantem a integridade dos testes, podendo de fato mostrar a eficácia ou não do MapReduce para grandes quantidades de dados em comparação ao seu modelo tradicional. Para todos os algoritmos de ordenação, é usada a mesma função de comparação, este método recebe dois conjuntos de caracteres e retorna qual deles é o maior.

Os arquivos de entrada que os testes vão utilizar, exploram a capacidade de processamento dos nodos e do *cluster Hadoop* como um todo, obtendo resultados específicos e sem favorecimento a nenhum dos algoritmos. O próximo capitulo irá trazer os resultados dos experimentos aqui descritos, expondo os testes de comparação de desempenho em cada arquivo que os algoritmos processarem.

4 RESULTADOS

Os resultados obtidos dos algoritmos paralelos, são provenientes da utilização dos 3 nodos do *cluster*. Os nodos são sincronizados pelo Apache Hadoop. Os algoritmos MapReduce foram executados neste ambiente por diversas vezes, até obter a melhor configuração de execução. Esta configuração, é feita em relação ao tamanho do arquivo de cada teste e a quantidade de funções *Map* e *Reduce* executadas pelo *Hadoop*.

Desta forma, os algoritmos MapReduce com ordenação no *Map*, independentemente de ser do tipo *Insertion Sort* ou *Merge Sort*, tiveram seu melhor desempenho com maior número de *splits* e menores tamanho de *slices*, a quantidade de *Reduce* não foram relevantes para este tipo de ordenação. Entretanto, para o algoritmo com ordenação na função *Reduce*, o tamanho dos *slices* não tem grande influência sobre a performance, neste caso a quantidade de funções *Reduce* fez a maior diferença. Em contrapartida, quanto maior quantidade de funções *Reduce*, maior são as incidências de erros do *Hadoop* para esta tarefa.

a	back	care	en	great	In	no	pulse	Tess	was
a	back	care	en	great	in	No	pulse	Tess	was
a	back	care	en	great	in	no	pulse	Tess	was
a	back	care	en	great	in	No	pulse	Tess	was
a	back	care	en	great	Kilimane	no	pulse	that	was
a	back	care	en	great	Kilimane	no	pulse	that	was
and	back	care	en	Harville	Kilimane	not	resource	that	was
and	be	curves	exists	Harville	Kilimane	not	resource	that	was
and	be	curves	exists	Harville	Kilimane	not	resource	that	we
and	be	curves	exists	Harville	Kilimane	not	resource	the	We
and	be	curves	exists	Harville	Kilimane	of	resource	the	we
and	be	curves	exists	her	Kilimane	of	resource	the	we
and	be	curves	exists	her	les	of	returned	the	we
and	been	curves	falling	her	Les	of	returned	the	we
and	been	direction	falling	her	les	of	returned	thought	we
and	been	Direction	falling	her	les	of	returned	thought	we
assisted	been	direction	falling	her	les	of	returned	thought	we
assisted	been	direction	falling	her	Les	out	returned	thought	you
assisted	been	direction	falling	hoped	les	out	schoot	thought	you
assisted	been	direction	falling	hoped	make	out	schoot	vinegar	you
assisted	Business	direction	falling	hoped	make	out	schoot	vinegar	you
assisted	business	direction	falling	hoped	make	out	schoot	vinegar	you
assisted	business	direction	Fig	hoped	make	out	schoot	vinegar	you
assisted	business	dream	Fig	I	make	patrols	seated	vinegar	you

Figura 4.1 - Amostra de arquivo ordenado

Fonte: Do autor

Os experimentos realizados mostraram que quanto menor o tamanho do arquivo de entrada, *slice*, em relação ao arquivo original, maior será o número de pedaços gerados e consequentemente maior o número de tarefas *Map*. Da mesma forma, quanto maior o número de *splts,* a etapa de *Reduce* já é iniciada mesmo sem terminar a etapa de *Map*. Também é possível parametrizar o número de tarefas *reduces* da execução MapReduce, de forma independente do tamanho do arquivo. Isto ocorre, devido ao fato que os *slices* são processados paralelamente nos *cores* dos nodos do *cluster Hadoop*.

Sendo assim, um número de *slices* proporcionais a quantidade de *cores*, apresentam melhora de performance nos algoritmos MapReduce. Entretanto, um número exagerado de *slices* acaba sobrecarregando os *cores* e dificultando o processo de junção destes pedaços. Nos experimentos realizados neste trabalho, foram executados testes para identificar o melhor ganho entre o tamanho de arquivo e processamento realizado. O *cluster* utilizado neste trabalho, possui 24 *cores* (divididos em 3 processadores, sendo 1 processador por nodo), porém o *Hadoop* disponibiliza 23 *cores* para o processamento paralelo, desta forma os arquivos foram divididos de maneira que gerassem em torno de 23 *slices*, pois com estas proporções foram diagnosticados os melhores desempenhos.

No caso da execução dos algoritmos de ordenação sequenciais, os testes foram realizados no nodo mais robusto do *cluster*. Foram medidos os tempos de execução de cada algoritmo em 3 execuções e a quantidade de memória consumida também foi mapeada. As execuções dos algoritmos sequenciais foram realizadas no servidor 3, o qual possui 32 GB de memória RAM e 38 GB de espaço em disco. Os resultados dos testes nos algoritmos de ordenação, realizados no cluster estão apresentados nas próximas seções.

4.1 Experimento com o algoritmo *Insertion Sort* sequencial

O algoritmo de ordenação *Insertion Sort* sequencial por ter complexidade de $O(n^2)$ acaba sendo muito mais lento do que as outras opções de algoritmos de ordenação, tratados neste trabalho. Este fato, dificultou os testes nos arquivos de 100 MB e 1 GB, tornando inviável a utilização destes arquivos na avaliação de desempenho. Testes realizados em um arquivo de 5 MB com 982.634 palavras levaram mais de 8 horas.

O problema não está ligado diretamente ao tamanho do arquivo, mas sim a quantidade de palavras especificamente. Isto ocorre devido à forma como este algoritmo faz sua ordenação. O *Insertion Sort* deve percorrer todos os itens e a cada *loop* deve comparar a palavra em questão com as palavras já percorridas, uma a uma, até que seja localizada a

posição exata da mesma. Desta forma, o algoritmo vai ficando cada vez mais lento à medida que aumentam a quantidade de palavras percorridas, pois com ele aumentam a quantidade de laços para encontrar a posição certa da palavra.

O teste no algoritmo de ordenação *Insertion Sort*, foi realizado em cima de um arquivo menor, com tamanho de 1 MB e cento e 183.082 palavras. Apesar do arquivo ser relativamente menor ao do arquivo de 5 MB do teste anterior e muito menor que o realizado nos testes dos outros algoritmos, este algoritmo de ordenação de forma sequencial possui o pior desempenho de todos em relação ao tempo, com uma média de 14 minutos e 35 segundos. O consumo de memória ficou relativamente baixo devido ao tamanho do arquivo, ficando com uma média de 20 MB entre todas as execuções. Os detalhes das três execuções do algoritmo, estão na tabela 4.1.

Tabela 4.1 - Detalhes dos resultados do *Insertion Sort* sequencial

Tamanho do arquivo	Quantidade de palavras	Tempo da execução 1	Tempo da execução 2	Tempo da execução 3	Tempo médio de execução
1 MB	183.082	00:14:30	00:14:43	00:14:31	00:14:37

Fonte: Do autor

O *Insertion Sort* apresenta um melhor desempenho com uma menor quantidade de dados, conforme experimentação realizada. Quando executado paralelamente no MapReduce, este algoritmo de ordenação demonstrou uma velocidade de execução muito superior à sua versão sequencial. Os resultados se equivaleram ao do algoritmo de ordenação *Merge Sort*, que tem uma complexidade inferior.

4.2 Experimento com o algoritmo *Insertion Sort* paralelo

O algoritmo de ordenação *Insertion Sort* paralelo, com a função de ordenação no *Map,* utilizou o mesmo método de ordenação deste algoritmo no formato sequencial. Foram realizadas três execuções do algoritmo nos arquivos de 100 MB e 1 GB. O monitoramento dos seus desempenhos na interface do Apache Hadoop é ilustrada na figura 4.2. Não foi necessária nenhuma adaptação no algoritmo, pois rodando paralelamente o algoritmo não vai ordenar um número tão grande de palavras quanto ao *Insertion Sort* sequencial, desta forma não vai enfrentar o problema relatado na seção anterior.

Figura 4.2 - Interface WEB do Apache Hadoop

Fonte: do autor

Desta forma, para obter o melhor desempenho deste algoritmo de ordenação, foi realizada uma bateria de testes, onde foram modificados o tamanho dos *slices,* para provocar um determinado número esperado de tarefas *Map*, obtendo o menor tempo de execução para os arquivos de tamanhos 100 Mb e 1 GB, individualmente. Os detalhes da configuração usada para rodar o *Insertion Sort* MapReduce se encontram na tabela 4.2. Também se encontram os tempos médio de execução das funções *Map* e *Reduce* nas três execuções deste algoritmo. Como a execução da versão sequencial do *Insertion Sort* só foi possível com o arquivo de 1 MB, também foi acrescentado às tabelas de comparativo de execução, tabelas 4.2 e 4.3, os tempos de execuções do *Insertion Sort* usando este arquivo de tamanho reduzido.

Tabela 4.2 - Configuração do *Hadoop* no *Insertion Sort* MapReduce

Tipo de Configuração	Arquivo de 1 MB	Arquivo de 100 MB	Arquivo de 1 GB
Tamanho do *slice*	1 MB	10 MB	20 MB
Tarefas *Map*	1	12	67
Tarefas *Reduce*	1	1	1
Tempo Médio *Map*	27 Seg	0,45 Min	1,35 Min
Tempo *Reduce*	15 Seg	1,19 Min	12,38 Min

Fonte: Do autor

Os testes realizados no *Insertion Sort* paralelo indicaram que para o arquivo de 100 MB o tamanho do *slice* que mais se adequou foi de 10 MB, gerando 12 *slices* e consequentemente 12 tarefas *map*. Apesar do *cluster* possuir 23 *cores*, foram utilizados 12 *slices*, e consequentemente paralelizado em 12 *cores*, pois ao se trabalhar com *slices* muito

pequenos ocorre que o número de tarefas *map* que são abortadas aumenta e o *Hadoop* deve repetir estas tarefas, desta forma o tempo total de execução acaba não compensando. Para o arquivo de 1 GB, o tamanho de arquivo de entrada inicial escolhido foi de 20 MB, gerando 67 *slices* e consequentemente 67 tarefas *Map*. Como este algoritmo possui o processamento no *Map*, um maior número de tarefas *Map* escalonaria melhor os processos, pois estariam distribuídas pelos *cores* do *cluster*. Em contrapartida, o processo de *Reduce* acaba gerando um gargalo, pois possui somente uma tarefa levando em torno de 12 minutos para rodar. Como é necessário gerar um arquivo de saída, neste caso, esta etapa acaba comprometendo o tempo total do processo como um todo. Os resultados das execuções deste algoritmo se encontram na tabela 4.3.

Tabela 4.3 - Detalhes dos resultados do *Insertion Sort* MapReduce

Tamanho do arquivo	Quantidade de palavras	Tempo da execução 1	Tempo da execução 2	Tempo da execução 3	Tempo médio de execução
1 MB	183.082	00:00:44	00:00:37	00:00:45	00:00:42
100MB	22.331.674	00:01:47	00:01:53	00:01:49	00:01:50
1GB	237.942.884	00:13:11	00:13:45	00:13:36	00:13:31

Fonte: Do autor

Os tempos médios de execuções, acabaram ficando em 42 segundos para o arquivo de 1 MB, 1 minuto e 50 segundos para o arquivo de 100 MB e 13 minutos e 31 segundos para o arquivo de 1 GB. Estes tempos, mesmo com o maior arquivo, acabou não chegando ao tempo do arquivo de 1 MB do algoritmo sequencial, que foi de 14 minutos e 35 segundos. Isto demonstra a eficácia do MapReduce com algoritmos de complexidade alta.

4.3 Comparação entre o *Insertion Sort* sequencial e paralelo

O algoritmo de ordenação *Insertion Sort*, em seu formato tradicional, tem uma complexidade alta, O(n^2), o que dificultou muito os testes em arquivos grandes, tornando impossível realizar alguns deles devido à demora para finalizar o processo de ordenação. Na tabela 4.4 estão os tempos médio das três execuções do algoritmo em cada um dos arquivos nela informado e a versão do algoritmo correspondente.

Nota-se um ganho muito grande em relação ao tempo de desempenho do algoritmo paralelo em comparação ao sequencial, isto se deve a estratégia do algoritmo sequencial a utilizar-se de muitos *loops* para comparar todos os valores, no caso de uma lista de palavras

muito grande este algoritmo percorre grande parte lista várias vezes para obter uma lista totalmente ordenada. O algoritmo paralelo, como ordena uma parte menor do arquivo original por vez, acaba não permitindo *loops* muito longos, o que o que resulta nos ganhos observados deste experimento.

Tabela 4.4 - Comparação entre a versão sequencial e paralela do *Insertion Sort*

Tamanho do Arquivo	Insertion Sort Sequencial	Insertion Sort MapReduce
1 Megabyte	00:14:35	00:00:42
100 Megabytes	-	00:01:06
1 Gigabytes	-	00:13:31

Fonte: Do autor

Embora a complexidade do algoritmo continue a mesma na forma paralela, o tempo de execução diminui muito devido ao fato de que o formato sequencial não aproveita da melhor forma todo os *cores* do processador. Além disso, o algoritmo *Insertion Sort* no formato paralelo com MapReduce, além de explorar melhor todos os *cores* do processador, utiliza também os nodos disponibilizado no *cluster*. O maior fator de redução no tempo de execução está relacionado aos pedaços do arquivo gerados pelo *Hadoop*. Como ele ordena uma quantidade reduzida de dados, o algoritmo paralelo para a ser muito mais eficiente.

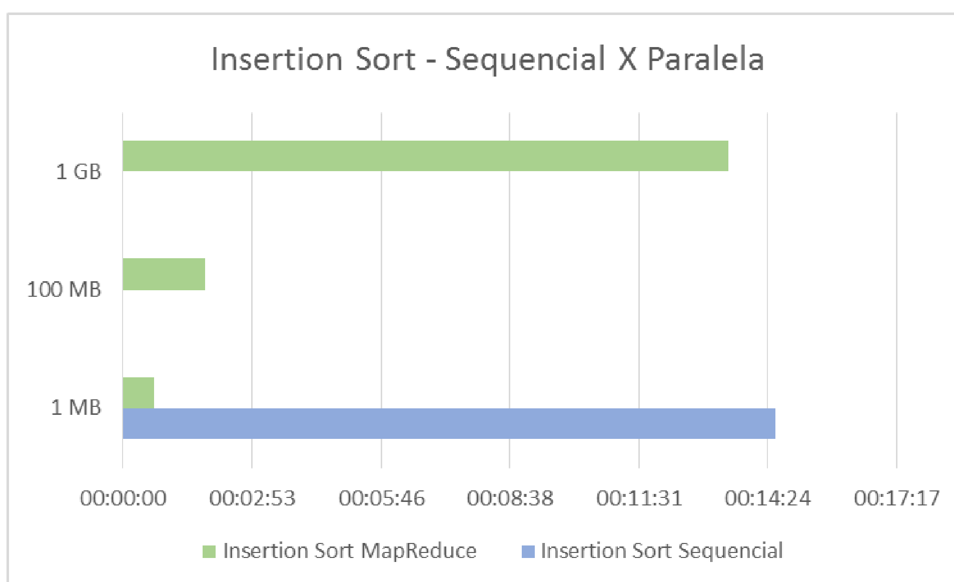

Figura 4.3 - Gráfico de tempos de execução do *Insertion Sort*

Fonte: Do autor

A figura 4.3 mostra que o maior tempo, dentre todos os tipos de execuções do *Insertion Sort*, é o da versão sequencial no menor arquivo de todos, 1 MB, e demonstra também a disparidade entre os tempos de execução neste mesmo arquivo, entre as versões paralela e sequencial. O algoritmo sequencial foi tão ineficiente neste caso que a execução no arquivo de 1 MB acaba sendo maior até mesmo que o arquivo de 1 GB. Neste experimento, pode-se notar que um algoritmo com complexidade $O(n^2)$ teve o seu tempo de execução reduzido ao ser paralelizado com o MapReduce, trabalhos futuros poderão explorar como reduzir o tempo de execução de outros algoritmos com complexidade alta sendo paralelizado com o modelo de programação MapReduce.

4.4 Experimento *Merge Sort* sequencial

O algoritmo de ordenação sequencial do *Merge Sort*, apesar de não ser projetado para ser rodado paralelamente, por ser um algoritmo baseado em funções recursivas, acaba sendo melhor escalonado pelo servidor e obtendo resultados semelhantes a sua versão paralela. Por ter complexidade de $O(n \log n)$ trouxe resultados muito superiores aos do algoritmo *Insertion Sort* sequencial, que tem complexidade maior $O(n^2)$.

O consumo de memória deste algoritmo de ordenação também foi observado neste experimento. Para que seja possível ordenar cadeias de caracteres em um arquivo, por um algoritmo de ordenação convencional, é necessário primeiramente ler o arquivo e posteriormente armazenar as cadeias de caracteres desejadas na memória. Ou seja, transformar as palavras do arquivo em um *array* de *strings*.

Na execução do experimento, foi observado que o *Merge Sort* sequencial, utiliza muita memória para rodar, decorrência do uso de funções recursivas que ficam armazenadas na pilha de execução. Ao rodar o arquivo com tamanho de 1 GB notou-se que o mesmo utilizou em média 10 GB de RAM nas três execuções do algoritmo. Esta medição, foi realizada a partir da diferença entre a memória livre no início da execução do algoritmo e ao término, utilizando-se para a medição o recurso nativo no Java chamado *Runtime*.

Sendo assim, este comportamento observado de alocação excessiva de memória é minimizado quando o algoritmo de ordenação roda no formato paralelo, já que o MapReduce acaba utilizando o sistema de arquivos HDFS para alocar espaço para execução. A experimentação realizada com o algoritmo de ordenação *Merge Sort* sequencial em relação ao

tempo de execução, está detalhada na tabela 4.5, apresentando o tempo de cada execução do algoritmo agrupado por tamanho de arquivo.

Tabela 4.5 - Detalhes dos resultados do *Merge Sort* sequencial

Tamanho do arquivo	Quantidade de palavras	Tempo da execução 1	Tempo da execução 2	Tempo da execução 3	Tempo médio execução
100MB	22.331.674	00:00:51	00:00:51	00:00:51	00:00:51
1GB	237.942.884	00:11:43	00:11:45	00:11:45	00:11:44

Fonte: Do autor

Neste contexto sequencial, este algoritmo de ordenação é muito superior ao *Insertion Sort*, mesmo sem a paralelização do algoritmo *Merge Sort*, foram possíveis fazer testes nos mesmos arquivos utilizados com os algoritmos de ordenação paralela, diferente mente do *Insertion Sort* sequencial, sendo viável fazer a comparação de desempenho entre as versões paralelas e sequenciais em condições de processamento mais próximas.

4.5 Experimento *Merge Sort* paralelo

A mesma metodologia de experimentação realizada com o algoritmo de ordenação *Merge Sort* sequencial, assim como os mesmos tamanhos de arquivos, foi utilizada para o algoritmo paralelo, com a função de ordenação no *Map*. Também foram realizadas três execuções e os resultados de desempenho foram obtidos pela interface do Apache Hadoop, como no caso do *Insertion Sort*.

A execução do algoritmo de ordenação *Merge Sort* paralelo foi realizada com o MapReduce utilizando estratégias diferentes para os dois arquivos. No caso do arquivo menor, de 100 MB, foi utilizando 10 MB para os arquivos de entrada, forçando o *Hadoop* a criar 12 tarefas de *Map*. Para o arquivo de 1 GB foi escolhido um tamanho de arquivo de entrada de 20 MB, fazendo que o MapReduce gerasse 67 tarefas *Map*. A tabela 4.6 detalha as configurações utilizadas para executar o *Merge Sort* paralelo.

Tabela 4.6 - Configuração do *Hadoop* no *Merge Sort* MapReduce

Tipo de configuração	Arquivo de 100 MB	Arquivo de 1 GB
Tamanho do *slice*	10 MB	20 MB
Tarefas *Map*	12	67
Tarefas *Reduce*	1	1

Tempo Médio *Map*	0,47 Min	1,36 Min
Tempo Médio *Reduce*	1,44 Min	12,46 Min

Fonte: Do autor

Da mesma forma que ocorreu com o *Insertion Sort* paralelo, foi notado um gargalo na função *Reduce*. Para que o algoritmo de ordenação possa ordenar com sucesso o arquivo de entrada de dados, é necessário gerar somente uma saída e para isto é necessário utilizar apenas uma tarefa de *Reduce*. A tarefa de ordenação, que fica na etapa de *Map*, executou em um tempo muito bom com uma média de 1 minuto e 36 minutos, tempo muito superior a versão sequencial. Os tempos referentes as três execuções do algoritmo no ambiente *Hadoop*, estão apresentados na tabela 4.7, agrupados por tamanho de arquivo.

Tabela 4.7 - Detalhes dos resultados do *Merge Sort* MapReduce

Tamanho do arquivo	Quantidade de palavras	Tempo da execução 1	Tempo da execução 2	Tempo da execução 3	Tempo médio da execução
100MB	22.331.674	00:01:56	00:01:57	00:01:50	00:01:54
1GB	237.942.884	00:13:40	00:13:31	00:13:15	00:13:29

Fonte: Do autor

Pode-se notar a proximidade dos tempos em relação ao algoritmo *Insertion Sort* paralelo, obtendo uma média de 13 minutos e 29 segundos no arquivo de 1 GB e 1 minuto e 54 segundos no algoritmo de 100 MB. No algoritmo *Merge Sort* não foi apresentado os tempos em relação ao arquivo de 1 MB, como no algoritmo anterior, pois o *Merge Sort* conseguiu executar ambos os arquivos propostos, de 100 MB e 1 GB.

4.6 Comparação entre o *Merge Sort* sequencial e paralelo

A execução do *Merge Sort* sequencial obteve vantagem em relação a sua versão paralela nos dois arquivos testados. Isto se justifica pelo fato de que o *Hadoop* executa alguns processos, como dividir o arquivo e gravação no disco, que acabam deixando o processo mais lento e sendo inferior a versão sequencial que é mais direto que sua versão MapReduce. O comparativo dos tempos de execução do *Merge Sort* sequencial e paralelo estão detalhados na tabela 4.8, onde se pode notar esta ligeira vantagem do sequencial sobre o paralelo.

Tabela 4.8 – Comparação entra a versão sequencial e paralela do *Merge Sort*

Tamanho do Arquivo	Merge Sort Sequencial	Merge Sort MapReduce
100 Megabytes	00:00:51	00:01:54
1 Gigabytes	00:11:44	00:13:29

Fonte: Do autor

Um importante ponto a ser observado, foi o uso excessivo de memória notado no algoritmo sequencial (aproximadamente 10 GB para o arquivo de 1 GB). Este problema não foi notado no algoritmo MapReduce, pois ele utiliza o disco para compartilhar o processamento. Devido a esta constatação, pode ser inviável rodar este algoritmo sequencial em um arquivo maior, tendo um cenário semelhante ao encontrado no algoritmo *Insertion Sort* sequencial. A tendência, é que quando utilizado arquivos maiores, tendendo há *terabytes,* o algoritmo MapReduce seja mais eficaz. A figura 4.4 mostra em forma de gráfico, a vantagem em relação ao tempo de execução

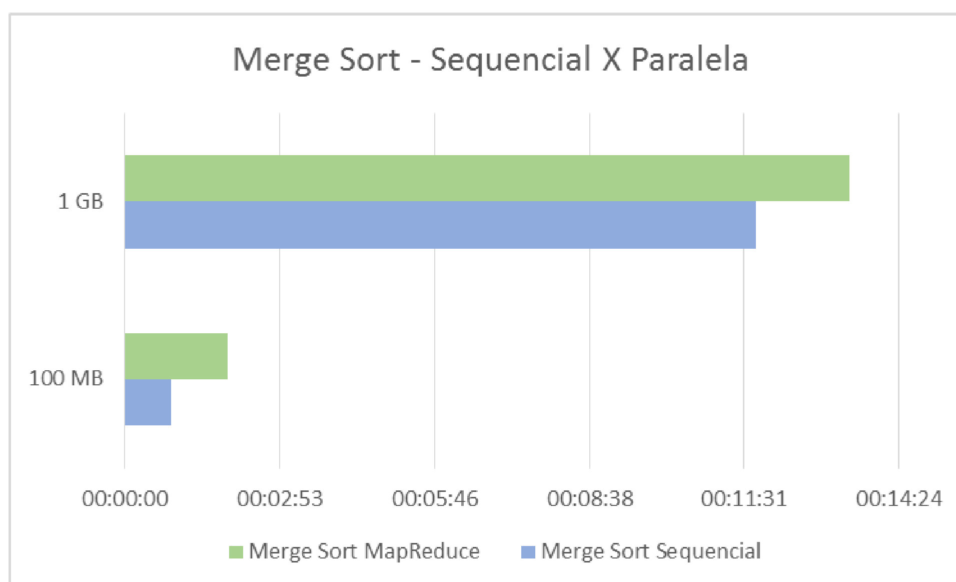

Figura 4.4 - Gráfico de tempos de execução do *Merge Sort*

Fonte: Do autor

Em contrapartida, pode-se fazer uma comparação somente dos tempos de ordenação propriamente dito. No caso dos algoritmos sequenciais, o tempo de execução para ordenação é contado desde o momento de ler o arquivo, grava-lo como uma lista em memória até efetivar a ordenação, não está sendo contado o tempo de gerar um arquivo final a partir desta lista ordenada em memória. O algoritmo paralelo com MapReduce, executa uma pré-ordenação na etapa *Map* e finaliza a ordenação na etapa de *shuffle*, a etapa *Reduce* serviria para unir os *slices* gerados por estas duas etapas em um arquivo final, sendo assim ele não

estaria presente na contagem de tempo de ordenação. Os tempos de comparação entre os algoritmo *Merge Sort* sequencial e paralelo sem o *Reduce* estão descritos na tabela 4.9.

Tabela 4.9 - Comparação entre *Merge Sort* sequencial e MapReduce sem o *Reduce*

Tamanho do Arquivo	Merge Sort Sequencial	Merge Sort Map
100 Megabytes	00:00:51	00:00:47
1 Gigabytes	00:11:44	00:01:36

Fonte: Do autor

Pode-se concluir que, comparando somente o tempo médio do processo do *Map*, o algoritmo paralelo se torna mais rápido em ambos os arquivos, conforme está apresentado em formato de gráfico na figura 4.5. Entretanto, um algoritmo de ordenação que não se possa obter a saída correspondente ao seu propósito de ordenar, não teria finalidade alguma em termos de utilização para algo específico.

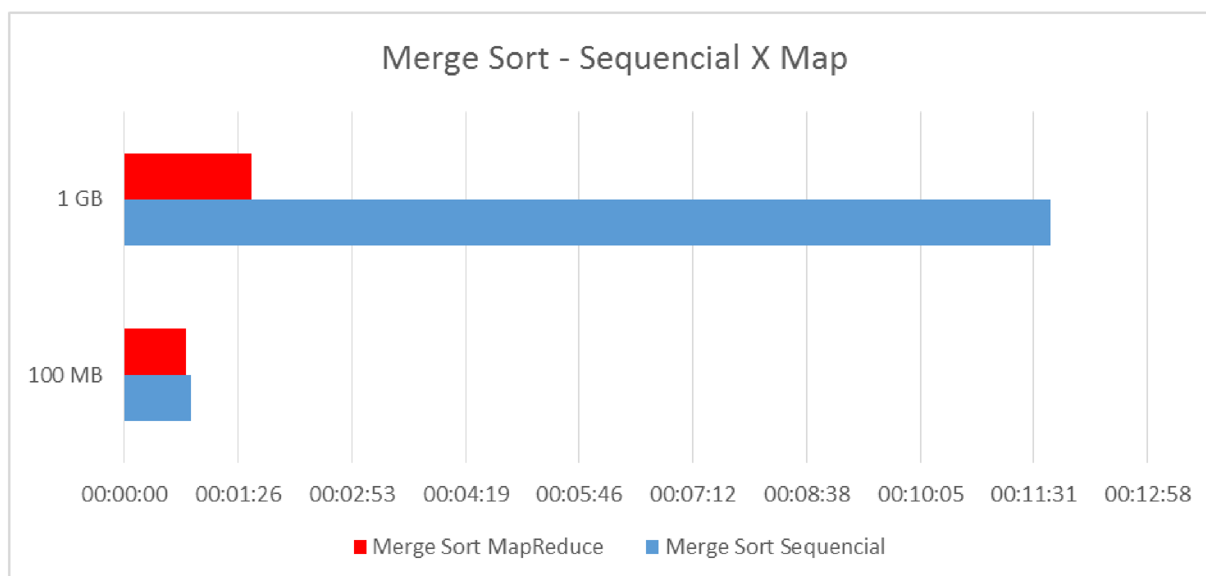

Figura 4.5 - Gráfico de tempos de execução do *Merge Sort* na função *Map*

Fonte: Do autor

O *Merge Sort*, na sua versão convencional (sequencial), é um dos algoritmos de ordenação mais rápidos devido a sua complexidade ser O(log n). Entretanto é conhecido por utilizar muita memória (GARCIA, 2013). Esta constatação pode abrir a possibilidade para trabalhos futuros, onde possam ser explorados até qual tamanho de arquivo o *Merge Sort* possa ser mais eficiente do que sua versão MapReduce.

4.7 MapReduce com ordenação no *Map* com mais de um *Reduce*

As execuções anteriores, utilizando algoritmos de ordenação paralelos, por meio do modelo de programação MapReduce, apontaram um gargalo na etapa de *Reduce.* Tento em vista um *cluster*, com 3 nodos e 24 *cores*, esta função poderia ser paralelizada configurando no *Hadoop* um aumento no número de *reduces*, resultando em um ganho de desempenho considerável em ambos os tipos algoritmos de ordenação. Este ganho de desempenho está detalhado na tabela 4.10 para o algoritmo de ordenação *Insertion Sort* e na tabela 4.11 para o algoritmo de ordenação *Merge Sort.* As tabelas mostram o tempo de execução do algoritmos em relação aos arquivos com tamanho de 100 MB e 1 GB.

Tabela 4.10- Resultados do *Insertion Sort* MapReduce com 23 *Reduces*

Tamanho do arquivo	Quantidade de palavras	Tempo da execução 1	Tempo da execução 2	Tempo da execução 3	Tempo médio da execução
100MB	22.331.674	00:01:05	00:01:14	00:01:05	00:01:08
1GB	237.942.884	00:03:11	00:03:25	00:03:13	00:03:16

Fonte: Do autor

Tabela 4.11 - Resultados do *Merge Sort* MapReduce com 23 *Reduces*

Tamanho do arquivo	Quantidade de palavras	Tempo da execução 1	Tempo da execução 2	Tempo da execução 3	Tempo médio da execução
100MB	22.331.674	00:01:08	00:01:05	00:01:06	00:01:06
1GB	237.942.884	00:03:33	00:03:41	00:03:53	00:03:42

Fonte: Do autor

O tempo de execução mais relevante ocorreu quando foi executado o arquivo de 1 GB. O tempo médio para o caso do *Insertion Sort* foi de 3 minutos e 16 segundos, muito superior ao tempo de 13 minutos e 31 segundos referente a execução deste mesmo algoritmo com somente uma tarefa de *Reduce*, conforme exibido na tabela 4.10. O tempo referente ao arquivo de 1 GB do algoritmo *Merge Sort* foi de 3 minutos e 42 segundos, da mesma forma, levou muito menos tempo do que a sua versão com uma tarefa *Reduce*, com 13 minutos e 29 segundos, exibido com mais detalhes na tabela 4.11.

Neste contexto, pode-se notar que houve pouca diferença no tempo de execução do arquivo de tamanho 100 MB, mesmo assim houve uma vantagem no tempo médio de quase 50 segundos da execução do algoritmo com 23 *reduces* para a execução com 1 apenas. Embora uma vantagem pequena, projeta-se que quanto maior o tamanho do arquivo

processado a vantagem, em relação ao tempo de execução, se torna cada vez mais significativa a medida que aumenta-se o número de tarefas *Reduce*.

Os ganhos obtidos no tempo de execução, tiveram as configurações que estão detalhas nas tabelas 4.12 e 4.13, para os algoritmos *Insertion Sort* e *Merge Sort*, que mostram a quantidade de tarefas *Map* e *Reduce*, juntamente com o tempo médio de execução das duas funções referentes os arquivos de 100 MB e 1 GB. Estas configurações foram obtidas através de uma bateria de testes para conseguir o melhor desempenho quanto ao tempo de execução.

Tabela 4.12 - Configuração do *Hadoop* no *Insertion Sort* MapReduce

Tipo de configuração	Arquivo de 100 MB	Arquivo de 1 GB
Tamanho do *slice*	20 MB	60 MB
Tarefas *Map*	6	23
Tarefas *Reduce*	6	23
Tempo Médio *Map*	0,63 Min	1,63 Min
Tempo Médio *Reduce*	0,45 Min	1,67 Min

Fonte: Do autor

Tabela 4.13 - Configuração do *Hadoop* no *Merge Sort* MapReduce

Tipo de configuração	Arquivo de 100 MB	Arquivo de 1 GB
Tamanho do *slice*	20 MB	60 MB
Tarefas *Map*	6	23
Tarefas *Reduce*	6	23
Tempo Médio *Map*	0,62 Min	1,7 Min
Tempo Médio *Reduce*	0,44 Min	1,71 Min

Fonte: Do autor

Foi constatado que utilizando o mesmo número de tarefas para as etapas *Map* e *Reduce*, se obteve um tempo de execução muito menor ao obtido quando utilizado somente uma tarefa de *Reduce,* esta diferença está disposta no gráfico da figura 4.6.

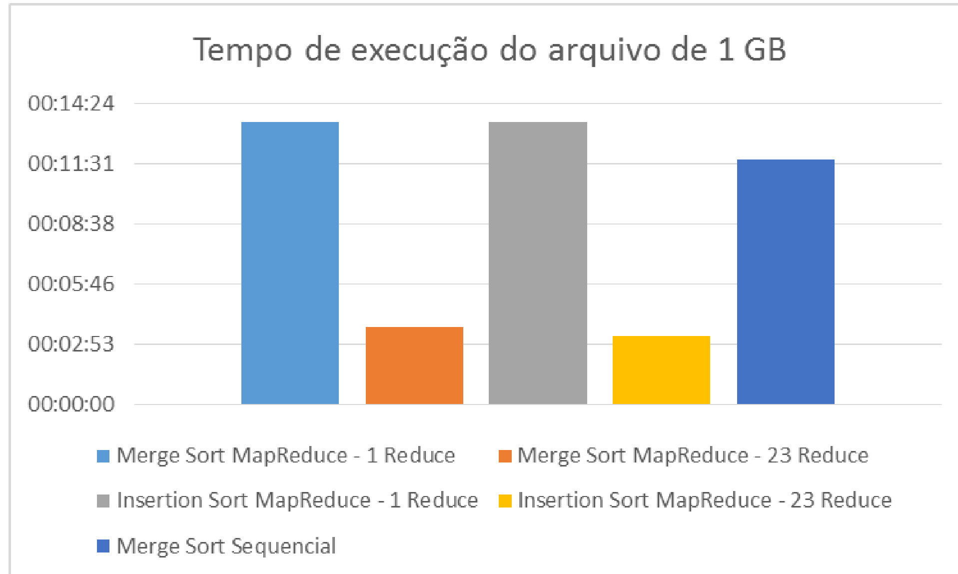

Figura 4.6 - Gráfico do tempo de execução do arquivo de 1 GB

Fonte: do Autor

A comparação entre as duas formas de execuções, de ambos os tipos de algoritmos de ordenação, detalhada na figura 4.6, juntamente com o tempo do *Merge Sort* sequencial, todos referente ao tempo de ordenação no arquivo com tamanho de 1 GB. Para os testes com mais de uma tarefa *Reduce*, foram utilizadas 3 execuções de cada algoritmo, assim como realizado com os outros algoritmos paralelos e sequenciais.

Embora o ganho tenha parecido expressivo quanto ao arquivo de 1 GB, ter muitas tarefas *Reduces* implica em ter vários arquivos de saída. O *Hadoop* acaba gerando 23 arquivos de saída, no caso do arquivo de 1 GB, conforme ilustra a figura 4.7. Este número de saídas de arquivos é gerado conforme a configuração do número de tarefas *Reduce*. Neste caso foram geradas 23 tarefas de *Reduce* e consequentemente foram gerados 23 arquivos de saída.

```
hduser@servidor3:~$ hadoop fs -ls /OrderMR/outputMerge100mb
Found 24 items
-rw-r--r--   1 hduser supergroup          0 2015-10-31 11:43 /OrderMR/outputMerge100mb/_SUCCESS
-rw-r--r--   1 hduser supergroup    8414299 2015-10-31 11:43 /OrderMR/outputMerge100mb/part-r-00000
-rw-r--r--   1 hduser supergroup   20799416 2015-10-31 11:43 /OrderMR/outputMerge100mb/part-r-00001
-rw-r--r--   1 hduser supergroup    9591454 2015-10-31 11:43 /OrderMR/outputMerge100mb/part-r-00002
-rw-r--r--   1 hduser supergroup    8824326 2015-10-31 11:43 /OrderMR/outputMerge100mb/part-r-00003
-rw-r--r--   1 hduser supergroup   10590189 2015-10-31 11:43 /OrderMR/outputMerge100mb/part-r-00004
-rw-r--r--   1 hduser supergroup   16243517 2015-10-31 11:43 /OrderMR/outputMerge100mb/part-r-00005
-rw-r--r--   1 hduser supergroup    9393750 2015-10-31 11:43 /OrderMR/outputMerge100mb/part-r-00006
-rw-r--r--   1 hduser supergroup   14936532 2015-10-31 11:43 /OrderMR/outputMerge100mb/part-r-00007
-rw-r--r--   1 hduser supergroup   10110587 2015-10-31 11:43 /OrderMR/outputMerge100mb/part-r-00008
-rw-r--r--   1 hduser supergroup    8347281 2015-10-31 11:43 /OrderMR/outputMerge100mb/part-r-00009
-rw-r--r--   1 hduser supergroup   11957182 2015-10-31 11:43 /OrderMR/outputMerge100mb/part-r-00010
-rw-r--r--   1 hduser supergroup    8291819 2015-10-31 11:43 /OrderMR/outputMerge100mb/part-r-00011
-rw-r--r--   1 hduser supergroup   10817394 2015-10-31 11:43 /OrderMR/outputMerge100mb/part-r-00012
-rw-r--r--   1 hduser supergroup   11988314 2015-10-31 11:43 /OrderMR/outputMerge100mb/part-r-00013
-rw-r--r--   1 hduser supergroup   14209085 2015-10-31 11:43 /OrderMR/outputMerge100mb/part-r-00014
-rw-r--r--   1 hduser supergroup   10399795 2015-10-31 11:43 /OrderMR/outputMerge100mb/part-r-00015
-rw-r--r--   1 hduser supergroup    8759609 2015-10-31 11:43 /OrderMR/outputMerge100mb/part-r-00016
-rw-r--r--   1 hduser supergroup    9215308 2015-10-31 11:43 /OrderMR/outputMerge100mb/part-r-00017
-rw-r--r--   1 hduser supergroup    9488732 2015-10-31 11:43 /OrderMR/outputMerge100mb/part-r-00018
-rw-r--r--   1 hduser supergroup   12935177 2015-10-31 11:43 /OrderMR/outputMerge100mb/part-r-00019
-rw-r--r--   1 hduser supergroup    7555266 2015-10-31 11:43 /OrderMR/outputMerge100mb/part-r-00020
-rw-r--r--   1 hduser supergroup    7870194 2015-10-31 11:43 /OrderMR/outputMerge100mb/part-r-00021
-rw-r--r--   1 hduser supergroup    8075462 2015-10-31 11:43 /OrderMR/outputMerge100mb/part-r-00022
```

Figura 4.7 – Lista de arquivos de saída com várias tarefas *Reduce*

Fonte: do Autor

Este comportamento ocorre porque à medida que o *Map* vai terminando as suas execuções, o *shuffle* já começa a agrupar as chaves geradas e repassa ao *Reduce*. Desta forma, as tarefas de *Reduce* que estão livres, obtém as chaves já geradas de forma aleatória e iniciam o processo de juntar os pedaços do arquivo para gerar o arquivo final. Como cada tarefa *Reduce* gera um arquivo de saída, os arquivos passam a agrupar pedaços ordenados, porém não de uma forma sincronizada, gerando vários arquivos ordenados conforme ilustrado na figura 4.8. Estes arquivos estão ordenados de A até Z, porém não é gerado um só arquivo final com todo conteúdo ordenado.

Arquivo 1

A	care	en	her	make	pulse
a	curves	en	hoped	make	schoot
a	curves	exists	hoped	no	that
and	direction	falling	hoped	no	that
and	Direction	falling	in	no	thought
back		great	in	of	vinegar

Arquivo 2

A	be	en	her	resource	the
a	care	falling	Kilimane	returned	we
and	care	falling	Kilimane	that	
and	care	Harville	make	that	
and	curves	Harville	not	the	
back	curves	her	not	the	

Arquivo 3

a	care	exists	hoped	make	pulse
assisted	care	exists	in	make	resource
assisted	care	great	Kilimane	no	returned
be	direction	Harville	Kilimane	no	schoot
be	Direction	Harville	les	of	that
been	en	her	make	of	thought

Arquivo 4

and	care	her	in	resource	
and	care	her	Kilimane	resource	
and	curves	her	Kilimane	returned	
back	curves	in	make	returned	
back	exists	in	not	Zoo	
care	falling	in	not		

Arquivo 5

A	care	exists	her	les	returned
a	care	falling	her	les	schoot
and	curves	great	hoped	not	thought
and	curves	great	in	not	zoo
and	direction	her	in	not	
back	Direction	her	les	not	

Arquivo 6

A	care	en	Kilimane	pulse	was
and	care	great	Kilimane	schoot	was
back	care	Harville	no	schoot	Zoo
back	curves	Harville	no	schoot	
back	curves	her	no	that	
back	en	hoped	of	vinegar	

Figura 4.8 – Exemplos de arquivos de saída com várias tarefas *Reduce*

Fonte: do Autor

Utilizar várias tarefas de *Reduce*, apesar de ter reduzido o tempo de execução, não gerou um arquivo final corretamente, por isso não pode ser considerado na análise de desempenho. Trabalhos futuros poderão explorar uma forma de ordenar estes vários arquivos gerados, unindo-os e ordenando novamente de uma forma eficiente para resultar em somente um arquivo final totalmente ordenado, pois dados pré-ordenados são ordenados de forma mais rápida (ASTRACHAN, 2003).

4.8 Ordenação com MapReduce com ordenação no *Reduce*

As seções anteriores trataram do algoritmo de ordenação no *Map*, ou seja, o algoritmo de ordenação, seja ele *Merge Sort* ou *Insertion Sort*, estava inserido na dentro da função *Map* do MapReduce. Uma das propostas de algoritmos de ordenação paralela com MapReduce, propostas por este trabalho, é utilizar o algoritmo de ordenação inserido na função *Reduce*. Desta forma, a função *Map* não teria a finalidade de fazer qualquer uma ordenação, mas sim de gerar chaves contendo a primeira letra de cada palavra encontrada e como valor o restante da palavra. Assim, o *shuffle* ordenarias as chaves e repassaria ao *Reduce*, onde este faria a ordenação de todas as palavras que começassem com uma determinada letra. O processo desta proposta de algoritmo está detalhado na figura 4.9, que expõe a disposição das palavras em cada etapa do MapReduce.

Arquivo
Be
Dream
Exists
Been
Curves
Business
Assisted
And
Em
A
Care

Map	
Chaves	Valores
B	e
D	ream
E	xists
B	een
C	urves
B	usiness
A	ssisted
A	nd
E	n
A	
C	are

Shuffle	
Chaves	Valores
A	
A	ssisted
A	nd
B	een
B	e
B	usiness
C	urves
C	are
D	ream
E	xists
E	n

Reduce	
Chaves	Valores
A	
A	And
A	Assisted
B	Be
B	Been
B	Business
C	Care
C	Curves
D	Dream
E	En
E	Exists

Figura 4.9 - Processo MapReduce para ordenação no *Reduce*

Fonte: Do autor

Sendo assim, para que este algoritmo funcionasse de forma paralela teriam que ser gerados várias tarefas de *Reduce*, tantas quanto o número de chaves geradas pelo *Map* para realizar o processamento entre vários nodos do *clustes*. Este formato ficou muito prejudicado em função de alguns problemas enfrentados quanto ao processamento paralelo, divisão de tarefas de *Map* e a geração de vários arquivos de saída.

A etapa *Map* foi realizada com sucesso, testes realizados mostraram que foram gerados chaves com a primeira letra de cada palavra, sendo ela letras números ou caracteres especiais. Para gerar um único arquivo deveria ser realizado somente uma tarefa *Reduce*, porem desta forma, o algoritmo de ordenação executaria o processo de ordenação sequencialmente, devido a tarefa de *Reduce* rodar somente em um nodo. Desta forma, também ocorreu o problema relatado com o algoritmo sequencial do *Insertion Sort*, onde o mesmo demorava muito tempo para ordenar, tornando o teste inviável.

Foi necessário então, criar várias tarefas de *Reduce*, onde foi constatado o primeiro problema. O número de chaves geradas pelo *Map* seria essencial para gerar o mesmo número de tarefas *Reduce*, para garantia que ele ordenaria um número menor de registro e geraria arquivos únicos para cada chave gerada, por exemplo, uma tarefa de *Reduce* ordenaria todas as palavras que iniciassem com a letra “A” e gerassem um arquivo somente com este grupo de palavras. O problema neste caso, está no número de tarefas chaves gerados pelo *Map*, que só é

possível se obter em tempo de execução. Para prosseguir com o experimento, foi estimado um número de 40 *Reduces*, desta forma o número seria maior que o chaves geradas pelo *Map*.

Sendo assim, foi enfrentado dificuldades quanto ao uso de memória no servidor quando executado o *Merge Sort*. Apesar de estar ordenando um número menor de palavras, algumas letras possuíam muitas palavras subsequentes, além do alto número de palavras repetidas presentes nos arquivos, o que mesmo assim acabou gerando um tempo muito alto nas execuções.

Arquivo 1		
curves	of	out
curves	of	out
curves	of	out
curves	of	out
curves	of	out
curves	of	out

Arquivo 2		
back	be	Business
back	be	Business
back	been	Business
back	been	Business
back	been	Business
back	been	Business

Arquivo 3		
Kilimane	Les	In
Kilimane	Les	In
Kilimane	Les	In
Kilimane	Les	In
Kilimane	Les	In
Kilimane	Les	In

Arquivo 4		
make	make	no
make	make	No
make	not	no
make	not	No
make	not	no
make	not	No

Arquivo 5		
A	A	and
a	a	and
a	a	and
A	and	assisted
a	and	assisted
a	and	assisted

Arquivo 6		
pulse	pulse	returned
pulse	pulse	returned
pulse	resource	returned
pulse	resource	returned
pulse	resource	returned
pulse	resource	returned

Arquivo 7		
that	Tess	thought
that	Tess	thought
that	Tess	thought
that	thought	thought
Tess	thought	thought
Tess	thought	thought

Arquivo 8		

Arquivo 9		
falling	falling	Harville
falling	great	Harville
falling	great	Harville
falling	great	Harville
falling	great	Harville
falling	great	Harville

Figura 4.10 - Arquivos de saída da algoritmo de ordenação paralelo no *Reduce*

Fonte: do Autor

Outro problema enfrentado, foi em relação aos arquivos de saída. O MapReduce não garante a ordem que as tarefas de *Reduce* irão executar. Por este motivo, alguns arquivos de saídas acabavam sendo gerados com as letras “A” e “C” no mesmo arquivo e também alguns arquivos acabavam não processando nenhuma chave e sendo gerados sozinhos. A figura 4.10 ilustra como ficaram os arquivos de saída utilizando este processo, nele é possível notar alguns arquivos vazios, alguns com mais de uma letra e também alguns estão corretos.

Como este experimento não gerou resultados finais esperados, no caso um arquivo de com palavras totalmente ordenadas, este formato de algoritmo não foi inserido na comparação

de desempenho. Tendo em vista o cenário gerado neste experimento, trabalhos futuros poderão ser realizados em cima deste formato de algoritmo, explorando formas de gerar um o mesmo número de tarefas *Reduces* conforme o número de chaves geradas. Ou como garantir que as tarefas *Reduces* processem, individualmente, somente uma chave. Também poderão explorar como agrupar algumas palavras, para evitar ordenar palavras repetidas e ordenar mais rapidamente, ou até mesmo aumentar o número de letras para gerar as chaves no *Map* para obter um grupo menor de palavras para o processo do *Reduce*.

4.9 Considerações Finais do Capítulo

O modelo de programação MapReduce demostrou ser uma poderosa ferramenta para o processamento paralelo. Como foi demonstrado com o algoritmo *Insertion Sort* que em sua versão sequencial foi inviável a execução, na sua versão paralela com MapReduce chegou muito próximo, ou algumas vezes foi superior, a um algoritmo de complexidade menor, no caso o *Merge Sort*.

Sendo assim, para chegar a estes resultados, foram necessárias diversas execuções do algoritmo MapReduce, utilizando diferentes tipos de configurações, como tamanhos de *slices* para gerar diferentes números de tarefas *Map* e quantidade de tarefas *reduces*, tendo o devido cuidado com o arquivo final de saída, que pode ser gerado de forma que não seja útil, conforme o tipo de necessidade. A partir dos testes realizados, foi constatado que quanto maior o número de *maps* ou *reduces* maior o número de tarefas abortadas, para os tamanhos de arquivos testado neste trabalho. Para a realização destes testes, foi mais prático ter um arquivo menor, que levasse menos tempo de execução para identificar a necessidade de tamanho de arquivo e número de *slices*.

Entre os teste realizados, o *Merge Sort* sequencial acabou sendo o algoritmo que teve a melhor média de tempo de execução. Entretanto, foi este algoritmo que utilizou mais memória do servidor, podendo tornar inviável o seu uso em um tamanho de arquivo muito grande em um único servidor. A pior média de tempo de execução foi observada no algoritmo *Insertion Sort* paralelo seu uso em arquivos muito grande, com muitos itens para ordenar acabou não sendo possível de ser executado devido ao seu tempo passar de horas para uma amostra de arquivo de 5 MB, sua execução acabou sendo feita em um arquivo de 1 MB.

A tabela 4.14 detalha todos os tempos médios de execução de todos os algoritmos de ordenação e todos os arquivos testados neste trabalho. Como a diferença de tempo de execução foi muito pequena, não se pode afirmar que o algoritmo *Insertion Sort* ou *Merge*

Sort é mais eficiente. Os tempos de execução dos algoritmos MapReduce não tiveram muita diferença de tempo de execução sendo comparado com a execução do mesmo tamanho de arquivo.

Tabela 4.14 - Tempos de análise de desempenho geral

Algoritmo	Arquivo de 1 MB	Arquivo de 100 MB	Arquivo de 1 GB
Insertion Sort Sequencial	00:14:37	-	-
Insertion Sort Paralelo	00:00:42	00:01:50	00:13:31
Merge Sort Sequencial	00:00:01	00:00:51	00:11:44
Merge Sort Paralelo	00:00:41	00:01:54	00:13:29

Fonte: Do autor

A figura 4.11, mostra a comparação de tempo de execução entre todos os algoritmos de ordenação, em relação ao arquivo com tamanho de 1 MB. Neste caso, o *Insertion Sort* sequencial levou mais tempo que os outros e o *Merge Sort* foi mais rápido levando em torno de um segundo para executar.

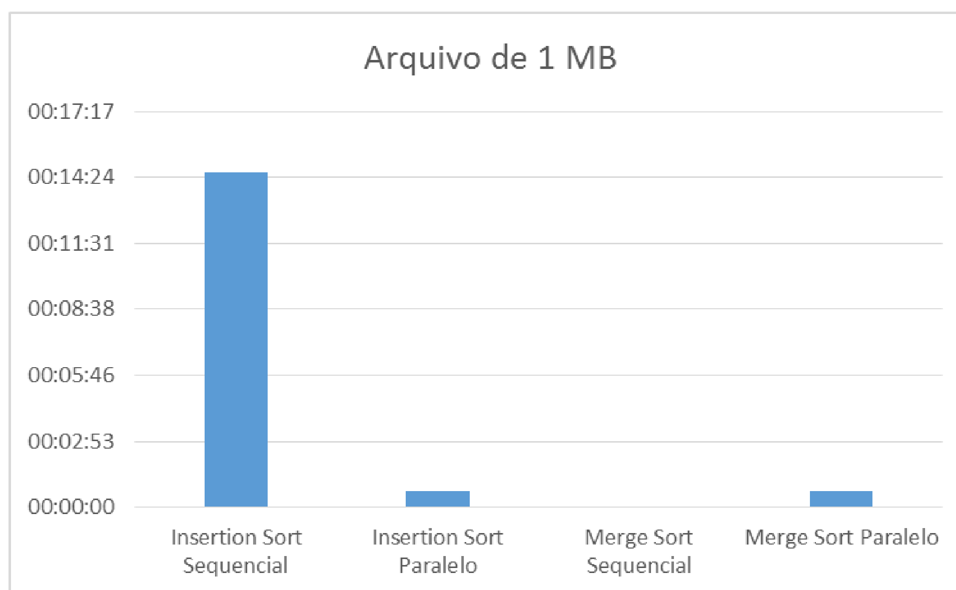

Figura 4.11 - Comparação entre os algoritmos de ordenação no arquivo de 1 MB

Fonte: do Autor

As figuras 4.12 e 4.13 mostram os tempos de execuções dos algoritmos de ordenação *Insertion Sort* paralelo, *Mergse Sort* sequencial e paralelo, quanto aos arquivo de 100 MB e 1 GB. Nestes gráficos fica evidente o melhor desempenho do algoritmo *Merge Sort* sequencial quanto ao tempo de execução.

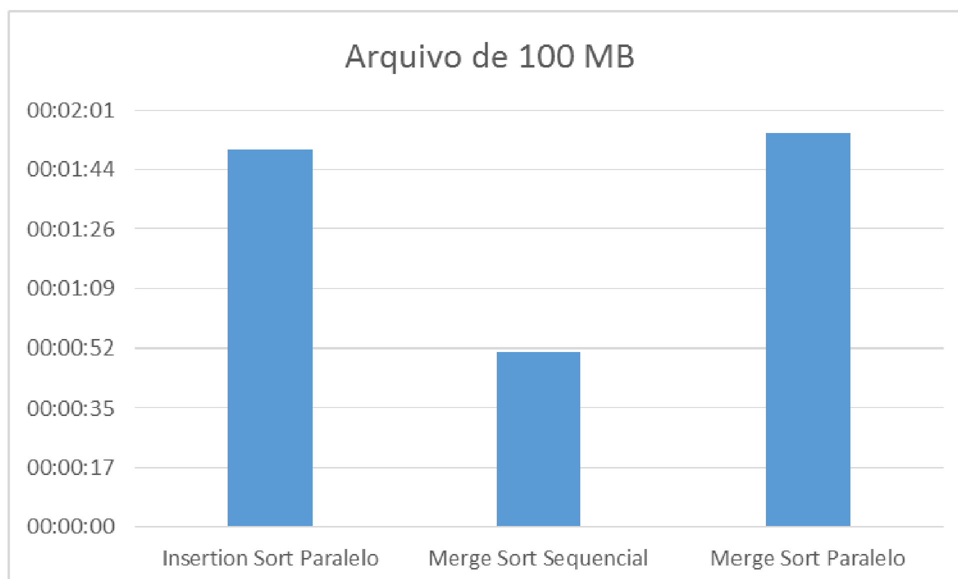

Figura 4.12 - Comparação entre os algoritmos de ordenação no arquivo de 100 MB

Fonte: do Autor

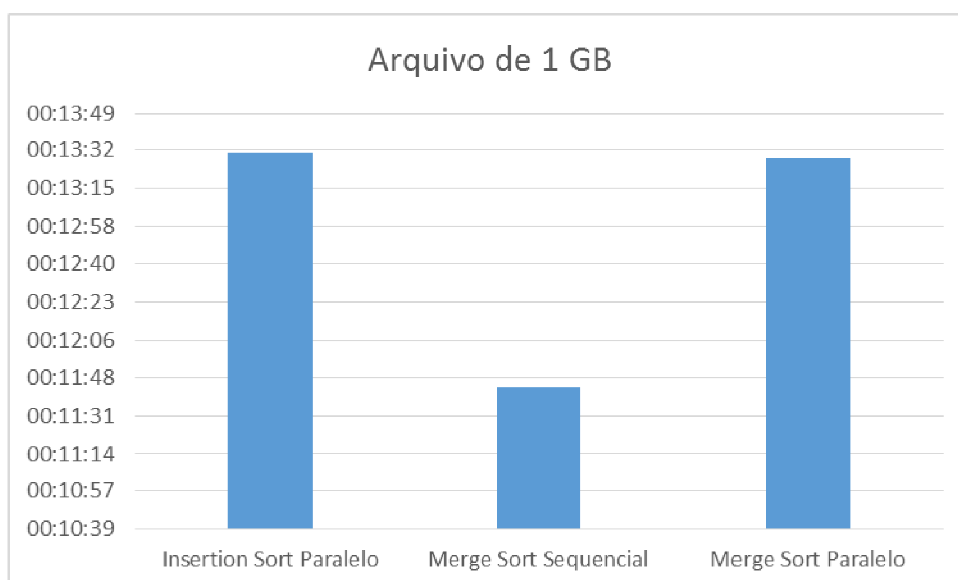

Figura 4.13 - Comparação entre os algoritmos de ordenação no arquivo de 1 GB

Fonte: do Autor

Os experimentos realizados neste capítulo mostram resultados interessantes perante ao desempenho dos algoritmo de ordenação com diferentes complexidades. Como visto anteriormente, algumas propostas de algoritmos tiveram insucesso, não chegando em um resultado contundente. Apensar disso, estes resultados abrem espaço para trabalhos futuros, conforme evidenciado nas seções anteriores.

CONCLUSÃO

Pode-se afirmar que o MapReduce é um ótimo modelo de programação para se trabalhar com paralelismo. Por ser um modelo de fácil entendimento, programadores inexperientes poderão desenvolver aplicações de forma paralela em *cluster*. Sendo assim, o Apache Hadoop também irá facilitar este desenvolvimento, pois o *framework Hadoop* facilita a manipulação de dados em ambientes de *cluster* de computadores com a utilização do modelo de programação MapReduce, possibilitando que as áreas relacionados a mineração de dados e manipulação de grandes massas de dados sejam melhor exploradas.

A partir dos trabalhos correlatos descritos neste trabalho, é possível identificar a eficácia do MapReduce, quando o objetivo é manipular grandes quantidades de informações. Através das pesquisas bibliográficas relacionadas aos algoritmos de ordenação, fica evidente a melhora significativa de performance que recebem ao serem executados em paralelo. O objetivo deste trabalho é fazer uma avaliação de desempenho relacionando o ganho de desempenho descritos nos algoritmos de ordenação, com o modelo de programação paralela MapReduce.

Foi realizado uma avaliação de desempenho entre os algoritmos de ordenação *Insertion Sort* e *Merge Sort*, cada um deles no formato sequencial e paralelo utilizando o modelo de programação MapReduce. Os testes realizados por estre trabalho, fizeram uso do laboratório fornecido pela Universidade Feevale, na sala 204 do Prédio Verde, com um ambiente Apache Hadoop instalado e configurado para um *cluster* com três nodos. Os mesmos algoritmos de ordenação foram usados nas versões sequenciais e paralelas, o que proporcionou integridade aos experimentos realizados. Os testes efetuados com os algoritmos sequenciais foram executados de forma tradicional no servidor 3, que é o servidor mais robusto do *cluster*, já os testes realizados com MapReduce fizeram uso de todo o *cluster* e utilizaram os mesmos algoritmos de ordenação, inclusive com o mesmo código fonte inserido neste modelo de programação de duas formas, sendo elas dentro da função *Map* ou dentro da função *Reduce*.

Os resultados dos algoritmos executados, demonstraram o melhor desempenho quanto ao tempo de execução, quando utilizado o algoritmo *Merge Sort* sequencial, este algoritmo possui complexidade baixa, O(n log n), superando o tempo de execução dos algoritmos paralelos. Em contrapartida, este algoritmo exigiu muita memória do servidor, tornando o *Merge Sort* inviável com arquivos maiores. O algoritmo *Insertion Sort,* que possui

complexidade O(n²), não pode ser executado com os arquivos propostos inicialmente, sendo eles de tamanho 100 MB e 1 GB, devido ao seu tempo de execução ser muito alto, os testes neste algoritmo foram feitos em cima de um arquivo de tamanho reduzido, de 1 MB. Apesar de ser executado com um arquivo muito menor que os outros, o *Insertion Sort* sequencial obteve o tempo médio mais alto de todos.

Os algoritmos de ordenação que tiveram a proposta de inserir o mecanismo de ordenação na função *Reduce* do MapReduce, não obtiveram sucesso, isto devido as várias tarefas de *Reduce* que tiveram de ser configuradas na sua execução para existir paralelismo. O maior problema neste caso, foi devido as tarefas *Reduce* não estarem sincronizadas com as chaves criadas na etapa *Map*. A tarefas *reduces* geravam vários arquivo de saída que ordenavam os valores dentro das chaves, porém os agrupamentos de chaves não eram gravadas individualmente nos arquivos de saída, gerando chaves desordenadas dentro dos arquivos. Esta situação, não permitiu a criação de um único arquivo final totalmente ordenado e foi desconsiderado na avaliação de desempenho final. Trabalhos futuros poderão ser realizados em cima desta proposta de algoritmo, explorando formas de gerar um o mesmo número de tarefas *Reduces* conforme o número de chaves geradas pela função *Map*, assim como gerar arquivos finais individualmente por chave.

Os algoritmos MapReduce com o mecanismos de ordenação na função *Map*, tiveram sucesso nas suas execuções, porém acabaram obtendo um gargalo na função *Reduce*, o que acabou demando tempo excessivo para executar todo o processo em menos tempo que o algoritmo *Merge Sort* sequencial. Trabalhos futuros poderão explorar até qual tamanho de arquivo este algoritmo sequencial continua sendo mais efetivo que sua versão paralela. Este gargalo observado nesta execução, pode ter o seu tempo de execução reduzido quando este número de tarefas é aumentado. Porém, desta forma são gerados vários arquivos de saída ordenados, o que não resulta em um arquivo totalmente ordenado. Trabalhos futuros poderão explorar uma forma de ordenar estes vários arquivos gerados, obtendo um arquivo final totalmente ordenado.

A avaliação de desempenho, resultante deste trabalho mostrou que o MapReduce tende a ser mais efetivo quando utilizado em um arquivo muito grande, na casa do *terabytes.* Isto devido aos tempos dos processos que o *framework Hadoop* executa, para que seja possível trabalhar com grandes massas de dados. Apesar de nem todos os algoritmos propostos por este trabalho não terem obtido êxito, foram abertos espaços para que outros trabalhos possam partir destes estudos para realizar novos testes e obter novos resultados.

A principal contribuição deste trabalho, foi referente ao algoritmo de ordenação *Insertion Sort,* que de forma paralela com MapReduce conseguiu atingir médias de tempo de execução semelhantes aos tempos do *Merge Sort* que possui uma complexidade muito menor. Ao ser utilizado na função *Map* e consequentemente utilizando paralelismo, o *Insertion Sort* obteve ganhos de desempenho interessantes, devido ao fato de estar inserido em um contexto que divide os dados de entrada em partes menores e são nestes pedaços que este algoritmo realiza o seu processamento. Ao processar uma quantidade menor de dados, este algoritmo foi executado muito mais rápido do que se fosse executado em no mesmo arquivo de tamanho total.

O *Insertion Sort* tradicional, tem um desempenho melhor quando é executado em cima de trechos menores. Desta forma, outros algoritmos que tenha um cenário semelhante a ele podem ser explorados de forma que sejam adaptados ao MapReduce, baseado em uma quantidade de dados menor e ao mesmo tempo obtendo uma execução paralela, com toda a integridade proporcionada pelo Apache Hadoop.

REFERÊNCIAS BIBLIOGRÁFICAS

APACHE. **Welcome to Apache Hadoop**. Disponível em <http://hadoop.apache.org/>. Acesso em: 10/03/2015.

ASTRACHAN, Owen. Bubble sort: An Archaeological Algorithmic Analysis. SIGCSE Bull, 2003.

BARBOSA, Marco AC; TOSCANI, Laira; RIBEIRO, Leila. **Ferramenta para a Automatização da Análise da Complexidade de Algoritmos.** SBIE2000–XI Simpósio Brasileiro de Informática na Educação/SBC. Anais..., n. Maceió, 2000.

BOLINA, Camilo. **Avaliação do Framework Mapreduce para Paralelização do Algoritmo Apriori**. Universidade Federal de Lavras, n. Lavras, p. 70, 2013.

BOLINA, A. C., PEREIRA, D. A., ESMIN, A. A. A., PEREIRA, M. R. **MapReduce Apriori (MRA) : Uma Proposta de Implementação do Algoritmo Apriori Usando o Framework MapReduce.** Universidade Federal de Lavras (UFLA), n. Lavras, 2013.

CLOUDERA. **About Us**. <http://www.cloudera.com/content/cloudera/en/about.html> Acesso em: 09/05/2015.

COSS, Rafael. **What is the Hadoop Distributed File System (HDFS)**. IBM, 2012. <http://www-01.ibm.com/software/data/infosphere/hadoop/hdfs/>. Acesso em: 02/05/2015.

DA ROSA, Luciano; ANTONIAZZI, Luiz Rodrigo. **Métodos De Ordenação De Dados: Uma Análise Prática I.** XVII Seminário Interinstitucional De Ensino Pesquisa E Extensão, 2014.

DEAN, J.; GHEMAWAT, S. **MapReduce**: Simplified Data Processing on Large Clusters. Communications of the ACM, SIGMOD '07. v. 51, n. 1, p. 1–13, 2008.

FILHO, Celso Luiz Agra de Sá. **Processamento se dados em larga escala na computação distribuída**. Universidade Católica De Pernambuco, 2011.

GARCIA, Adriano M. **Analisando o Desempenho da Paralelização no Algoritmo de Ordenação Mergesort In-place**. Universidade Federal do Pampa (UNIPAMPA), n. Alegrete, RS, 2013.

GONDA, Luciano. Algoritmos BSP/CGM para Ordenação. **Universidade Federal de Mato Grosso do Sul**, v. Campo Gran, p. 80, 2004.

LOPES, Raul H C. **Ordenacao Interna de Sequencias**. Universidade Federal do Espírito Santo (Ufes), 2015.

MURTA, Cristina Duarte; RAQUEL, Mariane; GONÇALVES, Silva; PINHÃO, Paula De Morais. **Implementação e Avaliação de Algoritmos de Ordenação Paralela em MapReduce.** WSCAD-SSC 2013 - XIV Simpósio em Sistema Computacionais, n. Belo Horizonte, MG, Brasil, p. 18, 2013.

NESELLO, P.; FACHINELLI, A. C. **Big Data**: O Novo Desafio para Gestão. Revista Inteligência Competitiva, 2014.

OLIVEIRA, A. L. F. DE; SOUZA, U. DOS S. **Algoritmos Paralelos De Ordenação**. Universidade Federal do Rio de Janeiro, n. Niterói, RJ, Brasil, p. 70, 2008.

OLIVEIRA, D. **Big data:** o desafio de garimpar informações. Computerworld, p. 22, 2012.

PRADO, José Augusto Soares. Análise Experimental do Quicksort Probabilístico com Gerador de Números Pseudo-Aleatórios Penta-Independente. Universidade Federal do Paraná., n. Curitiba, 2005.

RUSSOM, P.; IBM. **Big data analytics**. TDWI Best Practices Report, Fourth Quarter, p. 38, 2011.

SILVA, Sérgio Francisco da. Realimentação de Relevância via Algoritmos Genéticos Aplicada a Recuperação de Imagens. Universidade Federal de Uberlândia, 2007.

TAURION, Cezar. **Você realmente sabe o que é Big Data?.** Disponível em <https://www.ibm.com/developerworks/mydeveloperworks/blogs/ctaurion/entry/voce_realmente_sabe_o_que_e_big_data?lang=en>. Acesso em: 10/03/2015

VELLOSO, Fernando. Informática: **Conceitos Básicos**. [S.I], Campus Elsevier, 2014.

VIEIRA, M. R.; FIGUEIREDO, J. M.; LIBERATTI, G.; VIEBRANTZ, A. F. M.. **Bancos de Dados NoSQL:** conceitos, ferramentas, linguagens e estudos de casos no contexto de Big Data. Simpósio Brasileiro de Bancos de Dados - SBBD 2012, n. 1, p. 1–30, 2012.

WHITE, Tom. **Hadoop: The definitive guide.** O'Reilly Media, Inc., 2010.

ZIKOPOULOS, P.; EATON, C.; DEROOS, D. **Understanding big data**. [s.l], McGraw Hill. p. 166, 2012.

ZIVIANI, Nivio. **Projeto de Algoritmos:** com Implementações em Pascal e C. Thomson, 2004.